

**Tízéves az
ELTE Eötvös József Collegium
Informatikai Műhelye**

**TÍZÉVES AZ
ELTE EÖTVÖS JÓZSEF COLLEGIUM
INFORMATIKAI MŰHELYE**

2004 – 2014



Budapest, 2014



EMBERI ERŐFORRÁSOK MINISZTERIUMA

A kiadvány az Emberi Erőforrások Minisztériuma támogatásával készült.

Az Informatikai Műhely megalapításának 10 éves évfordulója alkalmából 2014. február 21-én megtartott tudományos konferencia előadásainak anyaga. A cikkek szerzőinek anyagaiból szerkesztette: Csörnyei Zoltán

© Csörnyei Zoltán, 2014

ISBN 978-615-5371-30-1



Felelős kiadó:

Dr. Horváth László,

az ELTE Eötvös József Collegium igazgatója

A nyomdai munkákat a Pátria Nyomda Zrt. végezte

1117 Budapest, Hunyadi János út 7.

Felelős vezető: Fodor István vezérigazgató

Köszöntő

Elcsépelet, kiüresedő kifejezés az „interdiszciplinaritás”, amely ma egy-egy, magára valamennyit is adó tudományos intézmény, kutatócsoport vagy tehetséggondozó szervezet (szakkollégium) cégérének kötelező, elmaradhatatlan eleme. A „tudományszakok közötti” valódi együttműködés (*inter disciplinas*) azonban ritka, mint a fehér holló, kivételes jelenség. Az Eötvös József Collegium évszázadnál régebbi dögész – filosz párbeszéde, az akadémián elkülönülő tudományterületek képviselőinek egymást lelkesítő, állandó kíváncsisága, a magyar szellemtörténet és tehetséggondozás semmihez sem hasonlítható nagyszerű teljesítménye. A különböző szakok találkozása azonban csupán szellemi kaland marad, ha nem indul ki belőle közös gondolat, olyan kutatás, amelyben a résztvevők együtt kereshetik az őket foglalkoztató kérdések megoldását. Az Informatika műhely tíz évvel ezelőtti létrehozása épp ezt, a legnemesebb célt hivatott szolgálni. A sors kedvezése volt, hogy e kezdeményezésnek elindítója lehettem (*quorum pars fui*), ott sáfárkodhattam a Collegium szelleméhez méltó bővítés felett, amikor a nemrég önállósult Informatikai Kar vezetőjével – azóta az Eötvös József Collegiumért emlékérem birtokosával (2012) – Kozma László Dékán úrral, illetve Horváth Zoltán és Csörnyei Zoltán Tanár urakkal az Eötvös-collegista informatikus hallgatót „kigondoltuk”. A Collegium érdekében eltervezett és több-kevesebb sikerrel véghezvitt tettek közül – saját, szűkebb szakterületemre is gondolva – ezt tartom az egyik legjelentősebb eredménynek. A terv, a kezdeményezés azonban hamar elsorvadt volna, ha a működéshez nincsenek kivételesen kedvező feltételek: a Kar töretlen támogatása, a műhelyvezető ragaszkodása és munkája és a kiváló diákok jelenléte. Most, amikor a tízéves működés alkalmából gratulálok az Eötvös József Collegium Informatika műhelyének, tulajdonképpen köszönetet mondok. Köszönöm a Kar vezetésének, Csörnyei Zoltán Tanár úrnak, a műhely alapító vezetőjének, *spiritus rector*ának és az Eötvös-collegista informatikusnak, hogy tudásukkal, munkájukkal, igaz collegista kötődésükkel szolgálták a Collegiumot, és dicsőséget szereztek mindannyiunknak. Kívánom, hogy az a szellemi frissesség, amely az informatikus hallgatók collegiumi „beköltözésével” áthatott minket, hosszú ideig megmaradjon és tovább hasson.

Horváth László
igazgató

Tartalomjegyzék

Horváth László	
Köszöntő	5
Csörnyei Zoltán	
Az Informatikai Műhely első 10 éve	9
Bencs Ferenc, Englert Péter, Gévay Gábor	
Programozási versenyek	18
Lócsi Levente	
Az infrastruktúra fejlődése	24
Cséri Tamás	
Egyszer volt, hol nem volt, volt egyszer az Üvegszál	35
Balassi Márton	
Nagy méretű gráfok feldolgozása Hadoop és Giraph rendszerek segítségével	49
Bodó Zalán	
Gépi tanulás gráfokkal	61
Bozó István, Tóth Melinda	
Erlang folyamatok és a köztük lévő kapcsolatok felderítése	79
Czigola Gábor	
Egy startup anatómiája	93
Gilián Zoltán	
Képregisztráció a frekvenciatartományban	108
Góbi Attila, Kozsik Tamás, Vezér Boglárka	
Aspektusok Scalához	118
Grósz Tamás, Tóth László	
Mély neuronhálók a magyar nyelvű beszédfelismerésben	139

Horváth Gábor, Kozár Gábor, Szűgyi Zsolt	
Típusbiztos szkriptnyelvek generálása funkcionális beágyazott	
nyelvekből.....	155
Kaposi Ambrus	
Bevezetés a homotópia-típuselméletbe	166
Kovács Györgyi	
A fordítóprogramokról.....	197
Kovács Lehel István	
Fotorealistikus 3D grafika: számítógéppel generált tájak	208
Kovács Máté	
Webes alkalmazások tesztelésének automatizálása.....	222
Kovács Péter	
A Collatz-sejtés matematikai és informatikai megközelítéseinek	
elemzése.....	235
Kozma László	
A programok helyességéről	248
Köllő Hanna	
A programkód és a konfigurációs kód közötti határ	277
Lócsi Levente	
Racionális függvényrendszerek és alkalmazásuk a jelfeldolgozás	
területén	286
Lővei Péter	
Az EIT ICT Labs Master School	305
Manninger Mátyás	
Funkcionális programozásban használt lencsék vizsgálata	
Ágdában	310
Nádor István	
Önszervező kritikus rendszerek az idegtudományban	320
Novák Ádám	
Konszenzus szekvenciaillesztések hatékony előállítására	330
Páli Gábor János	
Rendszerszintű fejlesztés funkcionális nyelven	353
Porkoláb Zoltán	
C++ template metaprogramozást támogató eszközök.....	373

Szita István	
Megerősítéses tanulás.....	395
Végh Zoltán	
Egy pályaelhagyó vallomása	411
Névmutató.....	415

Az Informatikai Műhely első 10 éve

Csörnyei Zoltán

Eötvös József Collegium

csz@inf.elte.hu

Az Informatikai Műhely 2004. január 8-án alakult meg. Az új műhely munkájának beindításához különösen sok segítséget kaptunk a Collegium Klasszika-Filológia Műhelyének vezetőjétől, *Horváth László* tanár úrtól. Az Informatikai Műhely Szabályzatát *Takács László*, az Eötvös József Collegium igazgatója és *Csörnyei Zoltán*, a műhely vezetője február 15-én írta alá, és ezen a napon fogadták el a Szabályzatot a Műhely tagjai is.

A Szabályzat első két pontja meghatározza a Műhely céljait:

- Az Eötvös József Collegium nagymúltú hagyományainak, célkitűzéseinek megfelelően a 2004-ben alapított Informatikai Műhely a tudós, önálló kutatásaikkal is kitűnő, hivatásukat szerető és szívvel-lélekkel végző informatikusok képzését tekinti feladatának.
- Az Informatikai Műhely annak a felismerésnek a szellemében fogant, miszerint a XXI. századi informatikus képzésben feltétlenül szükséges a legújabb elméleti és gyakorlati eredmények megismerése, és a modern informatika egységes, átfogó ismerete.

A Szabályzat további néhány pontja a Műhely működésének formai kereteit adja meg, így ez alapján kértük fel a Műhely tiszteletbeli elnökének *Lovász László* tanár urat, aki a felkérést elfogadta.

Alapításakor a Műhelynek csak négy tagja volt, a létszám folyamatosan emelkedett, jelenleg a Műhelynek a doktorandusz hallgatókkal együtt 22 egyetemista tagja van. A Szabályzat határozza meg azt is, hogy a Műhely tagja nem csak az informatikus collegista egyetemi hallgató, hanem a Műhely tagjának tekintjük a Műhely vezetőjét és az itt tanító

tanárainkat is. Örömmel látjuk köreinkben a nem-informatikus hallgatókat is, volt már matematikus hallgatónk, és most tagja a műhelynek egy bölcsész és egy fizikus hallgató is.

A Műhely vezetését a vezető tanáron kívül a Műhely collegista titkára is képviseli, az elmúlt tíz évben a műhelygyűléseken *Végh Zoltánt* (2004–2007), *Sztupák Szilárd Zsoltot* (2007–2009), *Csérei Tamást* (2009–2013) és *Manninger Mátyást* (2013–) választottuk meg titkárnak.

A Szabályzat meghatározza az első két pontban kitűzött cél elérésének alapvető módját:

- Az Informatikai Műhely munkájának célja az informatika és az informatikával kapcsolatos szakú collegista hallgatók képzésének kiegészítése. Ebből egyértelműen következik, hogy a collegistáknak többet kell teljesíteniük nem collegista évfolyamtársaikhoz képest.

IKP-9096/EC	Szita István	Statisztikai módszerek a gépi tanulásban
IKP-9098/EC	Csörnyei Zoltán	Modern elméletek az informatikában I.
IKP-9099/EC	Csörnyei Zoltán	Modern elméletek az Informatikában II.
IKP-9110/EC	Lővei László	Funkcionális programozási nyelvek II.
IKP-9120/EC	Csörnyei Zoltán	Mobil rendszerek elmélete
IKP-9134/EC	Tejfel Máté	Helyességbizonyító eszközök alkalmazása funkcionális programok esetén
IKP-9159/EC	Lócsi Levente	Digitális publikációs technikák
IKP-9178/EC	Kozma László	Komponens alapú rendszerek verifikációja
IKP-9179/EC	Diviánszky Péter	Funkcionális programozási nyelvek (Haskell)
IKP-9180/EC	Tóth Melinda – Bozó István	Funkcionális programozás 2 (Erlang)
IKP-9202/EC	Diviánszky Péter	Agda I.
IKP-9203/EC	Páli Gábor	Haladó Haskell
IKP-9215/EC	Kozsik Tamás	Programfejlesztés Scala-ban
IKP-9220/EC	Diviánszky Péter	Agda II.
IKP-ASZ1E	Fekete István	Algoritmusok és adatszerkezetek I.
IKP-ASZ2E	Fekete István	Algoritmusok és adatszerkezetek II.
IKP-2APF1E	Tejfel Máté	Párhuzamos folyamatok

1. táblázat. Az Informatikai Műhely meghirdetett órái

1. Tanulmányi eredmények

A többletteljesítmény kereteit a Műhely Szabályzatának további pontjai adják meg. Ezek között szerepel, hogy a collegisták a szemesz-

terenként az Eötvös Collegiumban meghirdetett előadások, speciális kollégiumok, szemináriumok közül legalább egy, szakjuknak vagy szakjaiknak megfelelő tárgyat kötelesek felvenni és abból levezsgázni.

Az Informatikai Műhely tagjai szemeszterenként változó előadás-, speciális kollégium- és szeminárium-kínálatból választhatnak. Az elmúlt években a Műhelyben az 1. táblázatban felsorolt órákat hirdettük meg, például a 2012/2013 tanév őszi félévében ezek közül 8 előadás futott párhuzamosan. Ezeket az órákat az ELTE Informatikai Karának tanszékei saját tanegységüknek ismerik el.

Az Informatikai Műhely tagjainak munkáját jól jelzi, hogy az elmúlt években tagjaink 33 *Köztársasági Ösztöndíjat* kaptak (2. táblázat).

	04	05	06	07	08	09	10	11	12	13
Balassi Márton	.	.	.	.	.	.	■	.	.	.
Cséri Tamás	.	.	.	■	■	.	■	.	.	.
Englert Péter	.	.	.	.	.	.	■	■	.	■
Gévay Gábor	.	.	.	.	.	.	■	■	■	■
Gilián Zoltán	.	.	.	.	.	■	.	.	.	.
Horváth Gábor	.	.	.	.	.	.	.	.	.	■
Kovács Györgyi	.	.	.	.	.	.	.	■	■	■
Kovács Péter	.	.	.	.	■	■	.	■	.	.
Laki Balázs	.	.	.	.	.	■	■	.	.	.
Leskó Dániel	.	.	.	.	.	■	.	.	.	.
Lócsi Levente	■	■	■	■	.	.	.	.	.	.
Manninger Mátyás	.	.	.	.	.	.	.	.	.	■
Nádor István	.	.	.	.	.	.	.	.	■	■
Parragi Zsolt	.	.	.	.	■	.	.	.	.	.
Szijjártó Beáta	.	.	.	.	■	.	.	.	.	.
Sztupák Sz. Zsolt	.	.	■	■	.	.	.	.	.	.

2. táblázat. Köztársasági Ösztöndíj

Kar Kiváló Hallgatója kitüntetést 23 alkalommal vett át collegistánk (3. táblázat), három tagunk, *Lócsi Levente* (2008), *Laki Balázs* (2010), *Gilián Zoltán* (2012) a Kar Legkiválóbb Hallgatója címet is megkapta, sőt *Tóth Melinda* tanárnő 2009-ben, egyetemistaként szintén megkapta a Kar Legkiválóbb Hallgatója kitüntetést. Így elmondhatjuk, hogy az

eddig kiadott öt *Legkiválóbb* kitüntetésből négy „itt van” a Collegiumban.

	04	05	06	07	08	09	10	11	12	13
Balassi Márton	.	.	.	.	.	.	.			.
Cséri Tamás	.	.	.	.	.	.			.	.
Englert Péter	.	.	.	.	.	.	.		.	.
Gévay Gábor	.	.	.	.	.	.	.	.		
Gilián Zoltán	.	.	.	.	.	.		.		.
Horváth Gábor	.	.	.	.	.	.	.	.	.	
Kovács Máté	.	.	.	.	.			.	.	.
Kovács Péter	.	.	.	.	.	.	.		.	.
Laki Balázs	.	.	.	.	.	.		.	.	.
Leskó Dániel	.	.	.	.	.	.		.	.	.
Lócsi Levente	.	.					.	.	.	.
Parragi Zsolt	.	.	.	.			.	.	.	.
Sztupák Sz. Zsolt	.	.	.				.	.	.	.

3. táblázat. A Kar Kiváló Hallgatója

Eddig egy tagunk, *Szita István* szerzett PhD fokozatot, a tavaszi félévben lesz a következő védés, és jelenleg négy doktorandusz hallgatónk van. Három tagunk, *Lócsi Levente*, *Kószegi Judit* és *Leskó Dániel* az ELTE Informatikai Karán tanársegéd. 2011-ben Lócsi Levente tanulmányi eredményei és közösségi munkája elismeréseképpen *Eötvös Collegiumért* emlékérmét kapott.

Az OTDK konferenciákon kettő I. díjat, egy II. díjat, kettő III. díjat és négy különdíjat kaptunk. Műhelyünk tagja, *Szita István* korábban, 2003-ban *Pro Scientia Aranyérmet* is kapott.

Műhelyünk tagjai mindig sikeresen szerepelnek az Eötvös Konferencián, amely már két alkalommal is elnyerte az Év Tudományos Rendezvénye díjat. Itt hazai és határon túli szakkollégiumok hallgatói tartanak előadásokat kutatási témájukban elért eredményeikről. A konferencia anyagából Adsumus címmel a Collegium tanulmánykötet jelentet meg.

A konferencián az Informatikai Műhely önálló Informatikai Szekcióval 2006-ban jelent meg, az elmúlt nyolc év alatt a konferenciákon a szekcióban 63 előadás hangzott el. A konferenciákon a szekció elnöke *Kozma László* dékán úr volt. A Collegium a dékán úrnak 2011

szepemberében a Collegium érdekében kifejtett áldozatos munkájáért, nagylelkű támogatásáért, az Informatikai Kar és a Collegium közötti szoros kapcsolat kialakításáért *Eötvös Collegiumért* emlékermet adományozott.

Ebben a tanévben már harmadszor rendeztük meg nagy sikerrel középiskolásoknak az Eötvös Tanulmányi Versenyt, és 2013 januárjában az ELTE Informatikai Karára igyekvő utolsó éves középiskolásoknak már másodszor szerveztünk Tehetségtábort.

A műhely tagjai programozási versenyeken is képviseltetik magukat. Tagjaink előkelő helyezéssel szoktak szerepelni a Sapia-ECM versenyen, az ACM verseny magyarországi fordulóján, és például a Challenge24 nemzetközi programozó verseny döntősei között is találunk műhelytagokat.

2. Műhelyközi kapcsolatok

A Műhely tagjai rendszeres résztvevői a Collegium *TTK-s estjeinek*, az Informatikai Műhely tagjai több alkalommal tartottak itt előadást kutatási eredményeikről.

Az első igazán nagy közös munka a HypereiDoc projekthez kapcsolódott. Az Eötvös Collegium Bollók János Klasszika-filológia, Orientalisztika és Informatikai Műhely együttműködésének eredménye a *HypereiDoc* nevű szoftver elkészítése volt. A program egy szövegszerkesztő, amely a szövegtöredékek digitális könyvtári feldolgozását segíti, és lehetővé teszi a legkülönbébb jegyzetek, járulékos adatok megjelenítését is. A begépelt szövegből a program követhetővé teszi a különböző kiolvasási javaslatokat, kiegészítéseket és értelmezéseket, és a tudományos szövegkiadói elvárásoknak megfelelő dokumentumot állít elő.

A HypereiDoc továbbfejlesztése volt annak a projektnek a célja, amelyben a Collegium nyelvi műhelyeivel közösen kialakított elvárásoknak megfelelő, elsősorban középkori szövegek kiadásához szükséges szerkesztői munkát végző program készült el.

A Collegium levéltárosa és a Történelem Műhely szakmai irányításával alakult meg a *Registrator* csoport, amely a Collegium levéltári anyagának digitalizálását és feldolgozását végezte el. Szabó Miklósnak, az Eötvös Collegium néhai igazgatójának kéziratok hagyatékát is ez a csoport archiválta és tette közzé a Collegium honlapján. A speciális

informatikai háttér kialakításában és a konkrét digitalizálási munkában az Informatikai Műhely 4 tagja vett részt.

3. Belföldi és nemzetközi kapcsolatok

Jó kapcsolatot tartunk fenn a társ-szakkollégiummal, az ELTE *Bolyai Kollégiumával*. Az elmúlt évben például a két kollégium vezetője kölcsönösen tartott a társ-szakkollégiumban egy-egy előadást kutatási témájának eredményeiről.

Az elmúlt években vendégül láttuk *Vámos Tibort*, a SZTAKI igazgatóját és *Vesztergombi György* professzort, a CERN munkatársát egy-egy előadás tartására. Ezek az előadások azt a célt is szolgálták, hogy collegistáink bekapcsolódhassanak az ezekben az intézményekben folyó tudományos munkákba. Meghívtuk *Járai Antal* tanár urat is, aki a legmodernebb mikroprocesszorokkal, azok szerkezetével, tervezési elveivel, működésével, korlátaival és nagy hatékonyságú számításokra történő felhasználásával ismertette meg a collegistákat.

Egyik évben *Novák Ádám*, Műhelyünk öregdiákja is tartott egy előadást oxfordi bioinformatikai kutatási eredményeiről. Beszélt kutatócsoportjuk munkájáról, valamint ismertette az oxfordi kutatási lehetőségeket, és élménybeszámolóján keresztül megismerhettük egy angol egyetemváros inspiráló légkörét is.

Az Eötvös Collegium Informatikai Műhelye elsőként, már 2005-ben együttműködési megállapodást kötött a kolozsvári *Babeş-Bolyai Tudományegyetem Farkas Gyula Szakkollégiumával*, amelynek eredményeképpen tanárokat fogadtunk egy-egy előadásra a Collegiumban, ennek keretében látogatott el hozzánk *Kása Zoltán* professzor és *Soós Anna*, aki most a kolozsvári tudományegyetem rektorhelyettese. A Farkas Gyula Szakkollégium vezetője, *Bodó Zsolt* és *Róth Ágoston* tartott két alkalommal egyhetes előadássorozatot collegistáinknak. Az Eötvös Konferenciákon a Farkas Gyula Szakkollégiumból eddig 8 hallgató vett részt. Tőlünk egyhetes előadásokkal a Műhely vezetője és 4 hallgató járt Kolozsváron. Ezt a kapcsolatot 2006-ban kibővítettük a Collegium és a *Kolozsvári Magyar Egyetemi Intézet* együttműködésére.

A *Sapientia Erdélyi Magyar Tudományegyetem Kiss Elemér Szakkollégiumával* 2010-ben kötöttünk együttműködési megállapodást, amelynek keretében az Eötvös Konferenciákon 3 hallgató vett részt. Az Informatikai Műhely 2 tagját 2013-ban látta a Szakkollégium vendégül

egyhetes előadássorozat megtartására.

A Collegium a *Szegedi Tudományegyetem Eötvös Loránd Szakkollégiumával* is kötött együttműködési megállapodást, az Informatikai Műhely egy Workshop keretében ellátogatott ide és előadást is tartott a szegedi Informatika Műhelyben.

2012-ben kapcsolat épült ki a *Dunaújvárosi Főiskola Kerpely Antal Computers Szakkollégiumával* is azon célból, hogy az együttműködési megállapodás alapján a szakkollégiumaink tagjai kölcsönös látogatásokkal, tanfolyamokkal, konferenciákon előadásokkal fejlesszék informatikai ismereteiket. Az Informatikai Műhely rendszeresen részt vesz a Főiskola által szervezett országos Szakkollégiumi Találkozózn, és például a Főiskoláról 2013-ban két hallgató is tartott előadást az Eötvös Konferencián.

4. A közösség

Kikapcsolódásként félévente műhelykirándulást szervezünk. Ősszel egy várost látogatunk meg (voltunk már például Egerben, Szegeden, Esztergomban, Komáromban, Szentendrén, Vácott), tavasszal pedig a természetben teszünk egy sétát (például a Sukoró és Lillafüred környéki erdőkben, a Mátrában, a Pilisben és a Börzsönyben is jártunk már).

Befejezésül néhány kép a kirándulásokról:



1. ábra. Dobogókőn, 2004-ben



2. ábra. $1024 = 2^{10}$ méter magasan, Magyarországon



3. ábra. Egerben, a borospince meglátogatása után



4. ábra. Szegeden 2007-ben



5. ábra. Vácott 2013-ban

Programozási versenyek

Bencs Ferenc, Englert Péter, Gévay Gábor

Eötvös József Collegium*

{hun.fertoo, engi.peti, ggab90}@gmail.com

1. A versenyek

A különféle programozási versenyek lehetőségek adnak informatikusoknak, hogy összemérjék felkészültségüket, tudásukat és programozási készségeiket. Ezek főleg azon hallgatók számára érdekesek, akik érdeklődnek az algoritmusok világa és a matematika iránt. A Műhely több jelenlegi és volt tagja is aktív versenyző, röviden ismertetnénk pár ismertebb programozási versenyt, valamint a műhelytagok ezekhez kapcsolódó eredményeit.

Az egyetemistáknak szólóak között talán legismertebb az *ACM programozási verseny*. Itt háromfős csapatokban, egyetemüket képviselve indulnak a fiatal informatikusok és matematikusok. Mivel a három versenyző egyetlen számítógépen osztozik, így az elméleti ismeretek és a programozási gyakorlat mellett a csapatmunka is elengedhetetlen a sikeres szerepléshez. A helyi forduló eredménye alapján az ELTE két legjobban szereplő csapatát küldi a regionális fordulóra, ahol az Informatikai Műhely tagjai is szép számban képviseltették magukat, az utóbbi hat évben pontosan hatan: *Cséri Tamás, Englert Péter, Gévay Gábor, Gilián Zoltán, Parragi Zsolt, Sztupák Szilárd Zsolt*, és a Collegium Matematika-Fizika Műhelyéből *Bencs Ferenc* és *Szalkai Balázs*.

A *Budapesti Műszaki Egyetem* minden évben megrendezi saját nemzetközi programozási versenyét. Szintén háromfős csapatokat várnak, azonban nem kikötés, hogy a csapat tagjai egy egyetemen tanuljanak,

*Bencs Ferenc (Mat-Fiz. Műhely) 2009–, Englert Péter 2009–, Gévay Gábor 2010–

sőt, egyetemi hallgatónak sem kell lenniük. Akkor is indulhat valaki, ha tanulmányait már rég befejezte. Az indulók mellett a felhasználható eszközök tekintetében is sokkal kötetlenebb a verseny, és az átlaghoz képest nagyobb arányban tartalmaz gyakorlatias hozzáállást igénylő feladatokat. Mára igazi nemzetközi versennyé nőtt, a 24 órás döntőbe jutó harminc csapat között tavaly, 2013-ban mindössze négy magyar csapat szerepelt. Ebből a négy csapatból kettő állt az Eötvös Collegium jelenlegi és volt tagjaiból. Az egyik csapatot *Cséri Tamás*, *Parragi Zsolt* és *Sztupák Szilárd Zsolt* alkotta, a másik csapat tagjai voltak *Bencs Ferenc*, *Englert Péter* és *Gévay Gábor*.

Marosvásárhelyen a *Sapientia Erdélyi Magyar Tudományegyetem* rendez minden évben programozási versenyt, ahol romániai és magyarországi egyetemek hallgatói mérik össze tudásukat. A versenyen egy, az Informatikai Műhely és a Matematika-fizika Műhely tagjaiból álló csapat kétszer is részt vett, és először harmadik, másodszor első helyezést érték el. Mindkét alkalommal *Bencs Ferenc*, *Englert Péter* és *Gévay Gábor* alkotta a csapatot.

Ízelítőként a versenyek stílusából, következzen két feladat, amelyeket ezen a versenyen tűztek ki.

2. Első feladat: sokszögek

Van n azonos hosszúságú pálcikánk. Hányféleképpen rakhatunk össze ezekből valahány szabályos háromszöget, négyzetet és szabályos hatszöget? Minden pálcikát fel kell használnunk, és két eset akkor számít különbözőnek, ha valamelyik sokszögből különböző számú keletkezik. n legfeljebb 2 000 000 000, továbbá az eredményt modulo 666013 kell kiszámolni. A program tesztetenként 1 másodpercig futhat.

2.1. Megoldás

A feladat jellegéből sejthetjük, hogy valamiféle rekurzióval oldható meg. Most ezen rekurzió megtalálásához generátorfüggvényeket fogunk használni. Egy a_n sorozat generátorfüggvényén azt az algebrai objektumot értjük, ami $f(x) = \sum_{n \geq 0} a_n \cdot x^n$. Az ilyen objektumok halmazán

bevezethetünk összeadást és szorzást, ahol a szorzást pontosan úgy végezzük, mint amit a polinomoknál megismertünk. Így ezen műveletekkel

a generátorfüggvények halmaza egy gyűrűt fog alkotni. (Megj.: az invertálható elemek épp azok, amelyeknek konstansa nem nulla.) Például a csupa-1 sorozat generátorfüggvénye $\frac{1}{1-x} = \sum_{n \geq 0} x^n$. Egy nevezetes tétel

az, hogy az a_n sorozat akkor és csak akkor tesz eleget lineáris rekurzióknak, ha a generátorfüggvénye $\frac{p(x)}{q(x)}$ alakú, ahol $p(x)$ és $q(x)$ relatív prím polinomok, $\deg(p(x)) < \deg(q(x))$. Sőt ha $q(x) = 1 - t_1x - t_2x^2 - \dots - t_kx^k$, akkor a_n eleget tesz az $a_n = t_1a_{n-1} + \dots + t_ka_{n-k}$ rekurzióknak ($n \geq k$).

A továbbiakban tekintsük azon A_n, B_n, C_n sorozatokat, amelyek megmondják, hogy n darab pálcikából hányféleképpen lehet csak háromszögekből, négyszögekből, illetve hatszögekből álló képet készíteni. Vegyük észre, hogy ezen sorozatok igen egyszerűek, ugyanis mindhárom sorozatban csak 0-k és 1-esek szerepelnek, méghozzá A_n -ben minden harmadik, B_n -ben minden negyedik, C_n -ben minden hatodik 1-es. Nem nehéz látni, hogy ezen sorozatok generátorfüggvényei sorra $f = \frac{1}{1-x^3}$, $g = \frac{1}{1-x^4}$, illetve $h = \frac{1}{1-x^6}$.

Most gondoljunk abba bele, hogy hányféleképpen lehet n pálcika seítségével csak háromszöget és négyszöget tartalmazó képet készíteni. Ez pontosan $\sum_{k=0}^n A_k B_{n-k}$ (az alapján összegezzünk, hogy hány pálcikát használtunk fel háromszögeépítésre). De vegyük észre, hogy ezen sorozat generátorfüggvénye éppen $f \cdot g$.

Hasonló megfontolással kapjuk, hogy a feladatban keresett sorozat generátorfüggvénye

$$f \cdot g \cdot h = \frac{1}{1 - x^3 - x^4 - x^6 + x^7 + x^9 + x^{10} - x^{13}},$$

amiből azonnal le tudjuk olvasni a definiáló lineáris rekurziót:

$$a_n = a_{n-3} + a_{n-4} + a_{n-6} - a_{n-7} - a_{n-9} - a_{n-10} + a_{n-13} \quad n \geq 13.$$

További probléma, hogy n túl nagy ahhoz, hogy a sorozat összes n -edik előtti elemét egyenként meghatározzuk. Eszünkbe juthat, hogy a lineáris rekurziók átírhatók mátrixos alakba. A rekurzió rendje 13, így egy 13×13 -as mátrixra lesz szükség, amellyel megszorozva a sorozat 13 szomszédos eleméből álló vektort, megkapjuk az 1-gyel nagyobb indexű pozíción kezdődő szomszédos 13 elemet. A mátrix első sorában lesznek a rekurzió együtthatói (amelyekkel skalárisan szorozva 13 szomszédos elemet, megkapjuk a következő elemet), a főátló alatti átlóban pedig

csupa 1-es lesz és így a vektor többi eleme visszafelé elcsúszik egy pozícióval. Ebből már megkaphatjuk $\mathcal{O}(\log n)$ időben az n -edik elemet, hiszen az első 13 elem vektorát megszorozva a mátrix $(n - 1)$ -edik hatványával (amit a gyorshatványozás algoritmusával kiszámolhatunk) az n -edik pozíción kezdődő 13 szomszédos elem áll elő.

3. Második feladat: Rubik-kocka

Hányféle állapotba forgathatjuk a $2 \times 2 \times 2$ -es Rubik-kockát, ha csak 180° -os forgatásokat engedünk meg?

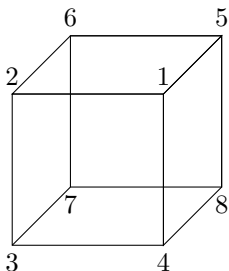
3.1. Megoldás

Válasszuk ki a Rubik-kocka valamelyik kis kockáját, és figyeljük meg, hogy milyen helyekre juthat el, és milyen állásokban. Egy 180° -os forgatás során egy kis kocka átkerül az egyik lapnak az átlósan szemben lévő csúcsába. Így, a kocka lapátlóin közlekedve négy helyre juthat el, amelyek egy tetraéder négy csúcsának felelnek meg. Továbbá vegyük észre, hogy mind a négy helyen csak egy-egy-féle állásban lehet (azaz, ha tudjuk egy kis kocka helyét, akkor nem kell foglalkoznunk azzal, hogy saját magán belül hogyan fordult el).

Ez alapján már adódik is, hogy a kérdést le tudjuk írni úgy, hogy 8 elemnek milyen permutációi érhetők el az oldalak forgatásával. Tehát adott egy alaphalmaz (8 elem összes permutációja) egy csoportthatással (amit az oldalforgatások generálnak), és az a kérdés, hogy mekkora a kocka alapállásának a pályája (orbitja).

A fentebbi gondolatmenetet már le is tudjuk programozni: képzeljük el azt a gráfot, aminek a csúcsai az alaphalmaz elemei, és minden csúsból indul ki 6 db él, amelyek megfelelnek egy-egy generátorelemnek. Azaz, lesz a kódban 6 db függvényünk, amelyek bemenetként kapják egy sorrendjét az $1, 2, \dots, 8$ számoknak (ami kódolja a kocka egy állapotát), és előállítják egy oldal forgatásának megfelelő átrendezését. Így a kezdőállapotból indított gráfbejárással megkaphatjuk és megszámlálhatjuk azokat az állapotokat, amelyek előállíthatók.

Most egy másik megoldási utat is szeretnénk mutatni, ami lényegében az *Orbit-stabilizátor tétel*en alapul, ami azt mondja ki, hogy egy csoportthatás esetén a csoport mérete megegyezik egy elem orbitja és stabilizátora (az a részcsoporthatás, amelynek minden eleme az adott



1. ábra. A kis kockák számozása

elemet helyben hagyja) méreteinek szorzatával. A továbbiakban a kocka állásaira, mint a 8 elem permutációjára gondolunk, így elegendő az előbb említett tételt rekurzívan használni.

A könnyebb olvashatóság végett az 1. ábra számozását fogjuk követni. Először is vizsgáljuk meg, hogy az 1-es helyen lévő kocka mely pozíciókba juthat el. Könnyen láthatjuk, hogy ez összesen 4 helyet takar (beleértve azt, ahol eredetileg is van). Ezek után a 2-es kocka 4 helyre juttatható el úgy, hogy az 1-es kockát a helyén tartjuk. Hasonlóan összeszámolva azt, hogy a 4-es kocka hány helyre juttatható el, míg az 1-es és 2-es kocka a helyén marad, 3-at kapunk. Végezetül az 5-ös kockát csak 2 helyre tudjuk elvinni úgy, hogy az 1-es, 2-es és 4-es kocka a helyén marad. Viszont a továbbiakban minden kocka a helyén marad, ha az 1-es, 2-es, 4-es és 5-ös kockákat a helyükön tartjuk. Így a keresett eredmény $4 \cdot 4 \cdot 3 \cdot 2 = 96$.

4. Utószó

A fentiek szép példái azoknak a versenyfeladatoknak, ahol a matematikai ismeretek is fontosak a sikerhez. Emellett azonban szükség van gyors és precíz programozási készségre is, mivel általában sok feladat szokott lenni. Egy feladatra átlagosan kb. 25–40 perc jut, így a rendelkezésre álló időbe nem fér bele sok hibakeresés.

A verseny az ACM versenyekhez hasonlóan zajlott. Háromfős csapatok vettek részt, egy számítógépet lehetett használni. Tizenegy feladat volt kitűzve, öt óra állt rendelkezésre. A tökéletesen megoldott

feladatok egy pontot értek, az azonos pontszámot elérő csapatok között pedig az időeredmény döntött. A legsikeresebb versenyünkön tíz feladatot tudtunk megoldani.

Marosvásárhely – a Bolyaiak városa – patinás, szép város. Az egyetem épülete kellemes környezetben, egy domboldalon található. A szervezők nagyszerű körülményeket teremtettek a verseny lebonyolításához.

Az infrastruktúra fejlődése[‡]

Lócsi Levente*

Eötvös József Collegium**

locsi@inf.elte.hu

A 2000-es évek elejére az informatika egyre nagyobb térhódítását, egyre szélesebb körű elterjedését élhettük meg. Ez tükröződött a Collegiumban is: míg az évezred elején kb. 10 személyi számítógép ki tudta elégíteni a collegisták igényeit, addig ma már szinte nincs olyan collegista, akinek ne lenne saját laptopja.

Elmondhatjuk, hogy a Collegiumnak az informatikai rendszer tekintetében sikerült megfelelnie a növekvő igényeknek, ha az elmúlt 10 évben nem is mindig a legszínvonalasabb, de legalább kielégítő szolgáltatást nyújtva.

1. Rendszergazdák

A Collegium számítógépes hálózatának és rendszerének üzemeltetését mindenkor a bentlakó collegisták oldották meg. A rendszergazdaságot mindig olyan tettekre kész és hozzáértő collegisták vállalták el – általában egyszerre több rendszergazda is működött –, akik tudtak tenni és tettek is a rendszer fenntartása, sőt fejlesztése érdekében is, akik saját ötleteiket és elképzeléseiket is szívesen beépítették a rendszerbe. Akár éjszakákon át is bütykölték a szervereket, programjaikat, vagy építették a kábelhálózatot. Megérteni és megoldani a Collegiumban rendszeresen felmerülő informatikai problémákat, megküzdzeni a technika ördögével, vagy csak észrevenni a kihúzott kábeleket, ez sem kis feladat.

[‡]Lócsi Levente [1]-ben megjelent anyagának rövidített, kissé átdolgozott változata.

*ELTE Informatikai Kar

**2003–2011

Tevékenységük hozzávetőleges idejét is jelezve, a Collegium rendszer-gazdái a következők:

Horváth Péter		
Mizera Ferenc		
Stefán István	–	2004
Novák Ádám	–	2004
Romsics Bence	2004	– 2007
Simon Győző	2004	– 2005
Czigola Gábor	2005	– 2007
Sztupák Sz. Zsolt	2007	– 2009
Parragi Zsolt	2007	– 2009
Ölvedi Tibor	2009	– 2011
Cséri Tamás	2010	– 2013
Hapák József	2011	–
Kovács Máté	2013	–

A következő fejezetekben még utalunk rájuk, hogy különböző tetteiknek is emléket állítsunk: kinek milyen honlapok, milyen szolgáltatások kialakítása, vagy éppen megszüntetése fűződik a nevéhez.

2. Gépterem

Sokáig fontos szerepet játszott a collegisták internetelésében, levelezésében, közösségi életében, mára azonban már teljesen elvesztette létjogosultságát a *számítógépterem*.

Valamikor régen a gépterem az alagsorban, a 018-as teremmel szemben lévő folyosó végén volt berendezve, de erre sokáig csak a rengeteg fali csatlakozó, dugaszoló aljzat emlékeztetett. 2009 végéig a gépterem a második emelet közepén, a fölépcsőházzal szemben üzemelt, öt-hat PC és egy nyomtató volt itt. Sokan rendszeresen látogatták a termet éjszakai munka, levelezés, játék, internetezés, társalgás céljából. Még az is, akinek egyébként saját számítógépe volt. A bejárók is hasznát vehették. A terem lányszárny felőli része pedig (azon túl, hogy a Mednyánszky Dénes Könyvtár és Levéltár állományának egy része is itt állt) a szerverterem, avagy a „*Stella*-szoba” volt, egy vékony térelválasztóval – és ügyesen elhelyezett könyvespolcok hátlapjával – elkülönítve a terem többi részétől, lakatra zárható ajtóval. Lényegében ez volt a hálózat, az informatikai rendszer szíve.

Az ELTE Informatikai Kara 2006-ban nagy segítséget nyújtott az elromlott collegiumi gépek pótlásában, és a későbbi években a kari leltárból kikerülő, de még jól használható gépek közül 30-at adományozott a Kar a Collegiumnak. Ezeknek nagy hasznát láttuk az infrastruktúra fejlesztésében, és a gépek egy része a *Registrator* csoport munkájában is szerephez jutott. 2011-ben további tucatnyi gépet kaptunk az Informatikai Kartól.

Az elromló gépek, a növekvő információéhség és a kapott új gépek miatt felmerült a gépterem egyfajta kibővítése, néhány további közös használatú számítógépnek a folyosókon való elhelyezése. Ez szinte csak a lányszárnyon, a második emeleten, a lépcsőfeljáró mellett valósult meg, itt kapott helyet a fénymásoló, a nyomtató, később a szkennel is. Eközben – minthogy egyre többeknek lett saját számítógépe – a gépterem elvesztette jelentőségét. A titkárság és a diákbizottság véleménye egyaránt közrejátszott abban, hogy végül a gépterem megszűnt és a szerverterem leköltözött a TMK-műhely mögé, helyükön pedig megnyílt a Társalgó.

3. A rendszer

A következőkben vázlatosan bemutatjuk a Collegium szervertermének hardveres és szoftveres fejlődését, alakulását.

Sokáig a *Stella* volt az egyetlen szerver, ezen futott minden szolgáltatás, a honlapoktól a levelezésig, talán a tűzfal funkciójának a kivételével. A Stellán UNIX rendszer működött (korábban FreeBSD, később Debian Linux, majd Gentoo). Levéve egy kis terhet a *Stella* válláról, a nyomtatás kiszolgálása évekig egy *EC130* nevű szerver feladata volt.

Akkoriban a fájlmegosztás úgy nézett ki, hogy volt három-négy collegista, akinek a gépén rengeteg zene, film volt, tőlük lehetett a belső hálózaton keresztül átmásolni anyagokat. Talán a legfontosabb szereplők ezen a téren 2004 táján Szita István *Enif*, Hamar Gergő *Dhamy* és Romsics Bence *Inertia* nevű gépe voltak.

Később több gép is a rendszer szolgálatába lépett, például egy időben *Bela* és *Belane* vették át a fájlszerver és a felhasználói könyvtárak tárolásának a szerepét. Volt, hogy a rendszergazdák saját gépeiket is kölcsönadták egy-egy funkció betöltése érdekében, ilyen volt például *Shamsiel*, akinek távozása 2009-ben nagy port kavart, mivel az akkor

kiköltöző Sztupák Sz. Zsolt és Parragi Zsolt rendszergazdák, miután a Collegium vezetésével nem tudtak megegyezni a rendszer áráról és tulajdoni viszonyairól, elvitték magukkal a rendszerüket.

Tűzfalszerverként szolgált még *Arakiel* és *Turiel* is, egy újabb szerver pedig az *Elias* nevet kapta. Volt egy időszak, hogy *Tóbiás* – Ölvedi Tibor rendszergazda laptopja – is látott el szerverfeladatokat.

A szolgáltatások egy része 2008 táján átvándorolt *Windows* alapokra: *Active Directory*, *LDAP* autentikáció stb. A lényeg az volt, hogy egy-egy géptermi gépre bejelentkezve az egyszeri collegista mindig ugyanazt kapja: akármelyik géphez is ül le, a saját háttérkép, asztal, dokumentumok beállításait lássa viszont. A saját felhasználói könyvtárainkat is egy másik protokoll szerint kellett és kell ma is elérnünk.

Operációs rendszerek tekintetében lépést tartottunk a *Microsoft* fejlesztéseivel, a géptermi gépeken a *Windows XP* után a *Windows Vista* és a *Windows 7* is megjelentek.

2010 tavaszán volt egy nagy szerverbeszerzés, akkor egy komolyabb számítógépet sikerült telepíteni, a „négysszázezer forintos szervert”.

Az új idők technikáit követve nálunk is megjelent a *virtualizáció* alkalmazása. Ez azt jelenti, hogy egy-egy feladat ellátását nem szükséges sem egy meglévő – már más feladatot ellátó – kisebb gépre bízni, sem egy újabb gépet összerakni ebből a célból, hanem a feladat ellátására kiszemelt „új gép” megjelenhet úgy, mint egy nagyobb teljesítményű szerveren futó program: egy virtuális számítógép. Ennek a technikának is számos előnye van. Például egy ilyen virtuális szerver látja el jelenleg az X-es honlap üzemeltetésének a feladatát, és a *Stella* is virtualizált módon élt tovább. A rendszer üzemeltetéséhez tartozott egy, majd két nyomtató, fénymásoló és egy szkennert fenntartása is.

4. Internet, levelezés

A collegisták számára 2007-től kezdve a Collegiumon belüli internethasználatához szükségessé vált a regisztráció és a „nethasználati díj” befizetése. Korábban a szolgáltatás ingyenes volt. A nem regisztrált felhasználók szűrését úgy lehetett megoldani, hogy a felhasználónak a kommunikáló számítógép és eszközeinek egyértelmű azonosítóját kódját (a „MAC”-kódot) regisztrációkor meg kellett adni.

A Collegium internetes szolgáltatásai közé tartozott és tartozik ma is az elektronikus levelezés. Ennek változásai közül az egyszerű felhasználók

talán csak arra emlékeznek, hogy sokszor nem működött, majd egy idő után ismét helyreállt. Mégis, néhány bekezdésben érdemes összefoglalni a főbb eseményeket a levelező rendszerrel kapcsolatban.

Kezdetben az egyszerű, de mégis nagyszerű *pine* programot használtuk sokan a *Stellára* bejelentkezve, parancssorból, karakteres felületen. Régi történet, hogy a dögészek sokszor csak odamentek az egyik terminálhoz és gyorsan elintézték a levelezést, míg a filoszok sorban álltak a többi géphez. Ugyanis ők nem értettek a *UNIX* rendszerek kezeléséhez, hanem csak grafikus felületen keresztül tudtak dolgozni. Történt ekkor, hogy egy agresszívabb bölcész boxkesztyűvel megfenyegette az akkori rendszergazdát, azt követelve, hogy mutassa meg neki is, hogy ez hogy működik, ne kelljen mindig sorban állnia. Máskor állítólag egy-egy gépet úgy védtek meg az illetéktelen – bölcész – használóktól, hogy bejelentkezéskor ki kellett számolni egy 3×3 -as mátrix determinánsát is.

Néhány éve a *Horde* nevű webes levelezőklienst üzemelték be a rendszergazdák, amelynek 2009-től kezdve egy újabb verziója működik. Két merevlemezen is tárolódnak a collegiumi levelezés anyagai, ám igen kellemetlen, de előfordult, hogy mindkettő egyszerre tönkrement.

2010 tavaszán a valamivel modernebb és komolyabb *Microsoft* rendszer, az *Outlook Web Access (OWA)* használata is felmerült, de ekkor ez az ötlet csak a tesztelési fázisig jutott.

5. Honlapok

Ami talán kívülről és belülről egyaránt jól látható és fontos része a collegiumi informatikai infrastruktúrának – bár csak a jéghegy csúcsa –, az a különböző honlapok kérdésköre. Ezek egyszerre hivatottak szolgálni a Collegiumon belüli, a collegisták közötti (néha csak belső, néha nyilvános) kommunikációt, valamint a Collegium megjelenését és virtuális reprezentációját a külvilág, a világháló felé.

Az évek során több honlap is megjelent, különféle funkciókat ellátva, az egy-két oldalból álló weblapoktól egészen a komplett portálokig. Vegyük most sorra ezeket, így is áttekintve a rendszergazdák és segítők munkáját, felelevenítve a Collegium webes megjelenésének történetét is.

5.1. Aktuális és Forum Collegii

A Collegiumban töltött első néhány évemben még az a honlap és fórum működött, amelyet pár évvel korábban Horváth Péter hozott létre. Talán még van, aki emlékszik az Aktuális és a Forum Collegii narancssárgás-barnás árnyalataira, nem is beszélve az ott folytatott heves vitákról. Az Aktuális a közérdekű hírek nyilvánosságra hozásának, a Forum Collegii pedig a viták folytatásának színtereként üzemelt. A belső kommunikáció mellett ez a rendszer egyben a Collegium külső megjelenését is szolgálta. Ugyanazon az egy szerveren (a neve Stella volt) működött a levelezés, valamint a collegisták saját anyagaikat is ide tehették fel.¹ Kiegészítve a nyomtatószámlákat kezelő belső rendszerrel (a collegisták befizethettek kisebb összegeket a rendszergazdáknál, amelyet utána nyomtatásra használhattak fel), ez az összeállítás sokáig jól szolgált.²

Hamarosan eljött az idő, amikor a Collegium igényei túlmutattak azon, amit ez a rendszer nyújthatott: egyre több kisebb szervezet, műhelyóra, tanárok kívántak egyre több és több információt egyszerűen megosztani; problémát jelentett továbbá a Forum Collegii-n egyre hevesebb és sokszor névtelen (vagy álnéven elkövetett) hozzászólások nyilvános volta is – erről Közgyűlések jegyzőkönyvei is megemlékeznek. (Az egyik szoba lakói például az „Ötvös Csöpi” álnév mögé rejtőzve közzétették gondolataikat a fórumon.) A 90-es éveket idéző dizájn, valamint a fejlődő webes eszközök és szabványok is hozzájárultak ahhoz, hogy tovább kelljen lépünk.

5.2. A Wiki hajnala

A váltás 2006-ban következett be – Romsics Bence és Czigola Gábor rendszergazdasága idején –, amikor több fronton is áttértünk elterjedt, szabadon használható, ingyenes technológiákra. A Collegium webes megjelenését egy Wiki rendszer adta (a Wikipedia mintájára), az online eszmecsere pedig egy PHPBB nevű fórummotor helyi installációján nyugodott (ilyet is sok helyen láthatunk). A wikis rendszer máig is él és

¹Régen jóval fontosabb volt, hogy a Collegium szerverén az ember rendelkezett egy jó nagy tárhellyel. Akkoriban ugyanis nem volt pendrive, gmail, a floppyk picik voltak és a saját merevlemezek drága kapacitásával is spórolni kellett.

²Történeti kiegészítés: 2001-ig működött egy belső levelező rendszer – hasonlóan a mostani „mindenki” levelezőlistához –, az úgynevezett „local disputa”.

üzemel,³ többször frissítették, és a felhasznált technológiák is fejlődtek.

Ez a rendszer igen sok előnnyel rendelkezett: könnyen szerkeszthető minden collegista – avagy regisztrált felhasználó – által (legalábbis az informatikusok szerint), és levéltári szempontból is kellemes: bármikor visszakereshető bármely lapnak egy korábbi változata is, nem vesznek el adatok. Az új fórum is kényelmesen használható lett, korszerű; szabályozhatóvá vált, hogy mi látható a kívüljáró felé, mi nem, bárki hozzászólhat-e egy adott témához, vagy csak a belépett felhasználók.

Persze ezzel a rendszerrel kapcsolatban is – már üzemelése helyezése előtt – felmerültek kételyek, például: megjelenését tekintve nem elég szép, nem elég reprezentatív, így nem méltó a Collegiumhoz. Általában sem szokás egy wikit hivatalos intézményi honlapként használni. Nem akadt azonban collegista, aki egy szebb honlap elkészítésére vállalkozott volna.

5.3. Az „új” Aktuális

Sztupák Szilárd Zsolt (avagy SztupY) és Parragi Zsolt 2007-ben átvette a rendszergazdaságot. Ők ketten vállalkoztak arra, hogy az egész collegiumi honlap- és fórumrendszert saját fejlesztésű, egységes szerkezetbe ültetik át. Hamarosan meg is született termékük, az *Aktuális*, amely valóban igen szépen kezelte a fórum kihívásait, a nyomtatószámlát, a felhasználók adminisztrációját stb. Még külön jelezhető volt az is, ha egy hozzászólás csak bizonyos műhelyek (TTK, magyar, TMK stb.) számára volt érdekes.

Azt azonban nem sikerült elérniük, hogy a wikit is leváltsák. Talán nem is baj, ugyanis a rendszergazdák távozásukkor a fórumrendszert is magukkal vitték.

5.4. A Facebook-tól az EIR-ig

A wikis honlap tehát megmaradt, viszont nem volt fórum: ismét egy PHPBB alapú rendszer beállításának feladata hárult akkor Kovács Mátéra mint informatikus collegára, ez az újrakelesztett fórum azonban nem volt hosszú életű. Valahogy ebben az időben, egy értelmes, megszokott, stabilan működő fórum hiányában eléggé átalakultak a collegisták fórumozási szokásai. Hamarosan megalakult a Facebook-on

³A wikis honlap elérhető például ezen a linken: <http://wiki.eotvos.elte.hu/>.

az „Eötvös Collegium” zárt csoport (ezt tehát nem olvashatja, és nem is írhat ide bárki, csak a tagjai, akik pedig a collegisták közül kerülnek ki). Sok információ – méltóbb hely híján – itt kezdett el terjedni, és a hozzászólások hossza, stílusa, valamint azok strukturáltsága is alkalmazkodott a közösségi oldal által adott lehetőségekhez. Figyelembe veendő azonban, hogy nem minden collegista található meg a Facebook-on.

Ekkortájt nyílt lehetőség arra, illetve vált bevett szokássá (a régi Aktuális funkcióját pótlendő) a „mindenki” levelezőlistára küldeni a közérdekű üzeneteket. (Ennek a levelezőlistának a tagja minden – főleg jelenlegi – collegista. Az ide küldött üzeneteket a tagok e-mailben kapják meg.)

Azonban ezek nem helyettesítenek egy jól működő fórumot, nem nyújtanak alkalmas teret a vitákra. . .

A rendszergazdaságot Ölvédi Tibor vitte tovább. Honlapok tekintetében ő is maradandót alkotott, több fontos információs oldal (például az általános leírásokat tartalmazó oldal, az internet-regisztrációt kezelő oldal, az új collegiumi újság honlapja) kiépítésén túl az EIR (Egységes Információs Rendszer) megvalósítása fűződik nevéhez. Alapvetően ezen a rendszeren keresztül érhető el most mindenféle leírás a rendszerről, itt lehet regisztrálni, felhasználónevet, internet-hozzáférést igényelni, a szobabeosztást megtekinteni, a diákbizottság által szervezett programokról olvasni stb. Ezeken túlmenően elkészült a leendő felvételizőket megcélzó blog is. A rendszergazdai munkába Cséri Tamás, majd Hapák József is bekapcsolódott. Később teljesen átvették a koordinálást.

A Facebook fellendülése előtt jóval népszerűbb volt a magyar iWiW rendszer. Itt is megjelent már 2007-ben az Eötvös Collegium azzal a fő céllal, hogy ebből az irányból is elérhessük a leendő felvételizőket, a végzős középiskolásokat. Ugyanekkor jött létre az Estike felhasználó is az iWiW-en.

5.5. Az X-es honlap

2009 végén kezdődtek meg a tárgyalások a győri X-Meditor céggel, amelynek a munkája már az ELTE BTK egyes honlapjain ismert volt. Ők kaptak a Collegiumtól megbízást egy látványos, jól használható honlap elkészítésére pályázati pénzből, kb. egymillió forint értékben. (Elsődleges feladat a pályázati követelményeknek való megfelelés volt,

másodlagos kérdésnek bizonyult, hogy valódi, szép, felhasználóbarát honlap legyen.)

Hamarosan – 2010 késő tavaszán – el is készült a dizájn és néhány kiegészítő modul, amelyet a cég már meglévő tartalomkezelő rendszeréhez hozzá kellett írni. Sokáig tesztelték, próbálgatták, a végső üzembe helyezését és beállítását a Collegiumban pedig jómagam vettem a vállamra 2011 nyarán, persze az akkori rendszergazdák segítő támogatásával. Jelenleg ez (az „X-es honlap”) üzemel, mint a Collegium hivatalos honlapja.⁴

2011 szeptemberében a diákbizottság és az egyes műhelyek is hozzáfoghattak a szerkesztéséhez. Remélhetőleg hamarosan nemcsak formája, hanem tartalma révén is igényes collegiumi honlap működhet, és talán a biztonsági mentések rendszere is kialakul.

6. Kábelezés és az üvegszál

A Collegiumban, a lakószinteken a folyosón sétálva a tablók és a régi képek mellett a mindennapi látvány elemei lettek a kusza kábelek (amelyek az internet-hozzáférést biztosítják a szobákban), továbbá a furcsa megoldások a hálózati eszközök rögzítésében, az érintésvédelmi technikákban. A Collegium – elsősorban a lakószintek – kábelezése is fontos pontja a helyi informatikának.

2003-ban, amikor még informatikai vonatkozásban alig 10 gép képes volt kielégíteni a Collegium igényeit, egy egyszerű, fekete koaxkábel (koaxiális kábel) futott végig a szinteken, a padlón szépen hozzásimulva a falhoz, leágazva egy-egy szobába, a hagyományos (BNC) csatlakozókkal teremtve meg a kapcsolatot az egyes számítógépekkel. Mivel ez egy soros kapcsolású rendszer, ha bármelyik ponton megszakad a kapcsolat, a teljes szint internetelérése leáll. Erre persze volt is példa bőségesen, ilyenkor a hosszadalmas, szobákon végigkopogtatós keresés és tesztelgetés árán állt csak helyre a rendszer.

Sokszor voltak felújítások ebben a tekintetben is. Ezek az átépítések egyébként nagyszerű közösségi eseményeknek is bizonyultak. Bár a feladat nagyobb része és levezénylése a rendszergazdák kezében volt, sok segítő collegánál megfordult ilyenkor a kalapács (hogy a kábelcsatornákat összerakja, a kábeleket az ajtófélfához rögzítse), a

⁴Kevésbé megszokott elnevezésű címe: <http://honlap.eotvos.elte.hu/>.

fúró (hogy a switch-eket a falra tegye) és a krimpelőfogó (hogy megfelelő méretű kábeleket gyártson, és a végükre szerkessze a csatlakozót). És persze ilyenkor a lányszárnyon is be kellett jutni minden szobába és barátságosan elmagyarázni, hogy mi is történik.

Egy collegisták által kiépített hálózatnak lehetnek hátrányai. Nem mindig ragaszkodunk például az érintésvédelmi előírásokhoz sem. (Nem veszélyes eszközökről van szó, inkább ezeket kell védeni az érintéstől.) Előfordult az is, hogy egy switch-et – mivel a célra kiszemelt fali dobozba nem fért bele – pusztán a belőle induló kábelek tartottak. Persze így nem maradhatott, doboznak lennie kell, ezért került bele a készülék egy neves pizzéria nagy pizzájának dobozába. A kábelek még így is elbírták. Továbbá az sem túl elegáns megoldás, hogy egy kisebb köteg kábel három folyosói kép között szalomozva érje el rendeltetési célját.

Külön érdemes volt megfigyelni és csodájára járni, milyen kalandos úton jutott el a hálózati kapcsolat az Estikébe. A 016-ba a 8-eres kábelnek csak 4 stratégiaiilag fontos ere fért le a nyíláson, ám a többin úgysem megy jel. A teremből egy átalakító a régi koaxkábelén továbbította az adást, amely kábel pedig kiment az ablakon, majd a Collegiumot körbefutó aknarendszerbe távozott. Innen pedig a pincszint valamelyik ablakán jutott be ismét az épületbe, majd további egy-két vezetéken és átalakítón keresztül érkezett meg az Estikébe. Már az egyszerű lekövetése sem volt könnyű ennek a megoldásnak – gondoljunk bele, hány kulcsot kellett felvenni és mennyit sétálni –, hát még milyen fantázia kellett e megoldás kiötléséhez és megvalósításához! (Az Estikébe internet egyébként főleg a „zenegéphez” kell, tudniillik, ahhoz a számítógéphez, amiről zenét lehet lejátszani. Van még egy másik állandó számítógép odalent, a – nevezzük most így – „regisztrációs gép”, amelyen pedig kb. 2005 óta a Czigola Gábor által írt „regisztrációs program” fut.⁵)

Az áttekintett időszakot végigkísérte a vágyakozás az *üvegszál* után, amellyel nagyon gyors hálózati kapcsolat valósulhat meg, ezzel is lehetővé téve a collegisták számára – többek közt – minél több online tananyag hatékony elérését. *Az üvegszál történetéről ennek a könyvnek Cséri Tamás által írt következő cikke szól* [2].

⁵Történeti kiegészítés: Az Estike program első verzióját Perge István írta 2001-ben. A retroérzéshez hozzá tartozott, hogy egy hibás monitor volt akkoriban odalent, ami a képernyő jobb felén széthúzta a képet, ami miatt a kép szélső 5 cm-e hiányzott. A program erre a monitorra volt optimalizálva, így a hibás monitoron nagyszerűen nézett ki, és szolgált ki többéves pályafutása során sok-sok elégedett Estike-vendéget.

7. Összefoglalás

Elmondhatjuk, hogy sikerült lépést tartanunk az informatika fejlődésével, az egyre növekvő igényekkel. Informatikai rendszerünk folyamatosan fejlődött, és bár lépten-nyomon felmerültek problémák, voltak elégedetlen hangok a szolgáltatásokkal kapcsolatban (például „lassú a net”, „nem megy a nyomtató”, nem beszélt *kedvesen* a rendszergazda), most működik a *kolínet*, és ha nem is kifogástalan, de legalább elfogadható, hála a rendszergazdák kitartó munkájának. Az üvegszálnak köszönhetően villámgyors lett a hálózat, a honlapok is ízlésesen, harmonikusan és céljuknak megfelelően szolgálnak, és – ráadásul – kábelkötegek sem kerülgetik a folyosón kifüggesztett képeket.

Hivatkozások

- [1] Lócsi L., Informatika a Collegiumban (2003–2011), *ECCE I.*, ELTE Eötvös József Collegium, 2013, pp. 351–374.
- [2] Cséri T., Egyszer volt, hol nem volt, volt egyszer az Üvegszál, *Tízéves az Eötvös József Collegium Informatikai Műhelye*, 2014.
- [3] Kovács M., *A jelenlegi állapot*, szóbeli közlés, 2014.

Egyszer volt, hol nem volt, volt egyszer az Üvegszál

Cséri Tamás*

Eötvös József Collegium**

cseri9@gmail.com

1. Hol nem volt

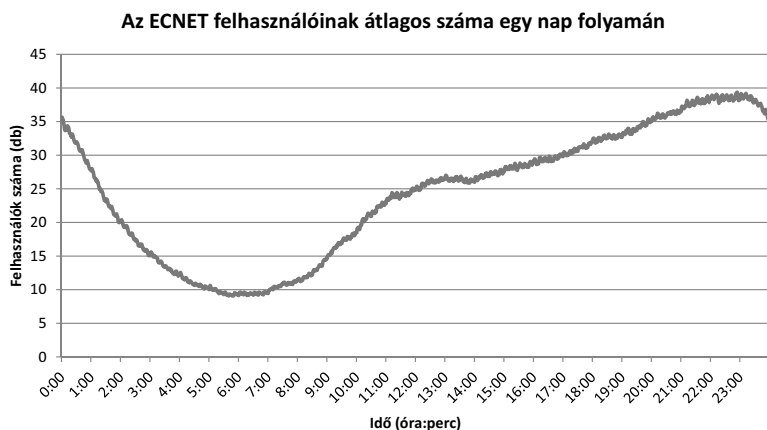
1.1. Miért volt szükség az üvegszálás internetre?

Amikor 2006-ban bekerültem a Collegium falai közé, már látszódott, hogy a meglévő ADSL-vonal az épület internetellátását nem tudja megfelelő módon biztosítani. Ugyan az idő nagy részében megfelelő sebességgel lehetett böngészni, de csúcsidőben – ami a Collegiumban nem a megszokott időben volt (lásd az 1. ábrán) – alig lehetett elérni a weboldalakokat.

A mérést 2007. december 11. és 2008. április 14. között végeztem a szerver címfeloldó gyorsítótárának (arp cache, a `/proc/arp` fájl) figyelésével. A Collegium teljes infrastruktúráját számba vettem, ezért a minimális 9 értékbe az éjjel-nappal üzemelő vezeték nélküli routerek is beleszámítanak. A géptermi gépek hosszabb inaktivitás után alvó üzemmódba kerültek, ezért nem kerültek éjjel a statisztikába. Aki laptopján egyszerre használta a vezetékes és a vezeték nélküli kapcsolatot, az kétszer szerepel a mérésben. Szintén figyelemre méltó, hogy a mérés során tapasztalt legmagasabb érték 72, tehát ebben az időben a collegisták jelentős része még nem rendelkezett számítógéppel.

*ELTE Informatikai Kar

**2006–2013



1. ábra

A lassúságnak több oka is volt. Egyrészt, annak következtében, hogy az interneten egyre nagyobb méretű tartalmak jelentek meg – gondoljunk például a YouTube 2005-ös indulására, amikortól általánossá váltak a videók az interneten – megnőtt a felhasználók sávszélesség-igénye. Másrészt, amikor az ADSL-vonalat kitalálták, még úgy gondolták, hogy a felhasználók a tartalmakat elsősorban az internetről fogják saját gépeikre tölteni, nem pedig fordítva, ezért a letöltési sávszélesség nagyobb volt, mint a feltöltési sávszélesség.

Az internet fejlődése azonban más irányt vett. A felhasználók egyre több tartalmat (képeket, videókat) akartak megosztani, ami jelentős feltöltési sebességet igényelt. Ezen felül elterjedtek a fájlcserélő hálózatok, ahol különböző, esetenként más forrásból nem megszerezhető tartalmakhoz juthatunk hozzá úgy, hogy nemcsak letöltjük, hanem cserébe más, ugyanazon tartalom iránt érdeklődő felhasználóknak tovább is adjuk, feltöltjük.

A Collegium épületében ezen kívül üzemelnek szerverek is, amelyeken külső felhasználóknak szánt tartalmak találhatók, például a Collegium honlapja, vagy az egyes collegisták személyes tárterülete. Amikor a külső felhasználó letölt valamit, azt a collegiumi hálózaton működő szervernek fel kell töltenie, tehát ez is a Collegium feltöltési sávszélességét veszi

igénybe. Így fordulhatott elő, hogy egy-egy nagy fájl letöltése, vagy egy „mindenki” levelezőlistára küldött nagy mellékletet tartalmazó levél kézbesítése nemcsak rendkívül sok időbe telt – a „mindenki” levelezőlistának tagja a mindenkori bentlakó és bejáró collegisták mellett igen nagyszámú egykori hallgató is –, hanem olykor percekre megbénította a teljes Collegium internetelérését.¹

1.2. Elképzelések az üvegszál bekötésére

A Collegiumba kerülésem után hamar hallottam a pletykát, hogy a BME egy üvegszál kábele, rajta nagyon gyors, 100 megabites internettel, itt halad el az épülettől nem messze, a Ménesi út túloldalán.

Elbeszélések során hallottam, hogy már 2003-ban is történtek próbálkozások a bekötésre, amit egy pályázatból finanszírozták volna. A kábel bekötését viszont nem tudták megoldani. A kábel ugyanis a T-cégcsoport tulajdonában volt, aki viszont nem tudta átvezetni az út alatt, mert az út alatti kábelcsatornájuk megtelt.² Emiatt útbontás lett volna szükséges, ezt viszont az önkormányzat nem engedte.

A szélessávú internetelés megvalósítható lett volna az ELTE Lágymányosi Campusa felé kialakított mikrohullámú kapcsolaton keresztül is. Ez a módszer sokkal kevésbé megbízható, mint a vezetékes megoldás, azonban mégsem olyan hihetetlen, mint ahogy első hallásra tűnik: a Kőrösi Csoma Sándor Kollégium a mai napig így csatlakozik az internetre. (Ott lakók elbeszélései alapján a stabilitásra vonatkozó félelmek nem voltak alaptalanok.)

Az üvegszál bekötésének gondolata folyamatosan napirenden maradt. Parragi Zsolt és Sztupák Sz. Zsolt rendszergazdasága idején konkrét lépések is történtek: egészen az árajánlatig jutottak, de éppen akkor sajnos nem volt a beruházásra megfelelő anyagi keret.

¹Ezt elkerülendő, a Collegium Ménesi úti épületének centenáriuma megjelent Lustrum kötetet például nem a Collegium szerverén, hanem a központi ELTE-s üzemeltetésű Caesar szerveren tettük közzé – innen sokkal gyorsabban letölthető, és nem befolyásolja a Collegium sávszélességét.

²A magas kihasználtságot magyarázhatja, hogy az út túloldalán levő telefonközpontból egy-egy kábel jön a Collegium minden egyes szobájába az ott található telefonkészülékbe.

1.3. Belső kábelezés

A collegisták a belső kábelezést először maguk építették ki.³ Habár eszközök és anyagi keret híján ezek nem voltak profi megoldások, folyamatosan haladtak a korrall és a növekvő igényekkel.

2004-ben épült ki az épületben a korábbi technológiákat leváltó UTP-kábelezés. Minden szobába három kábelt kötöttek be az akkori rendszergazdák (kivéve néhány szobát, ahol bölcsészek laktak - legalábbis ezt a választ kaptam indoklasként, amikor egy felsőbbévesnél érdeklődtem, hogy a 233-as szobába miért csak egy kábel vezet).

A 2005-2006-os tanévben Romsics Bence és Czigola Gábor rendszergazdasága alatt kiépült a vezeték nélküli hálózat is kilenc darab Linksys WRT54GL router⁴ felhasználásával. Ez egy rendkívül korszerű beruházás volt - a kollégiumok többségében a mai napig nincs vezeték nélküli internet szolgáltatás.

Az évek folyamán a régi kábelezés egyre használhatatlanabbá vált. A legfőbb problémát a szobákban levő csatlakozók (UTP-aljzatok) jelentették: ezeket a padlótól kb. 20 cm magasságban, az ajtófélfá egyik oldalán alakították ki. Emiatt viszont a szobák többségében legalább egy kábel a küszöb mellett haladt, amibe ha beleakadt a collegista lába, akkor az UTP-csatlakozó erős volta miatt nagy eséllyel nem a kábel, hanem a nehezen szerelhető fali aljzat sérült. Szintén probléma volt, hogy a switch-ek sorba voltak kötve, ezért a szerverteremtől távoli szobák már négy-öt sorba kötött, rossz minőségű switch-en keresztül kapták az internetet - így elég nagy volt a meghibásodás esélye.

2008-ban, Sztupák Sz. Zsolt és Parragi Zsolt rendszergazdasága alatt elkezdődött a rendszer teljes felújítása. Ennek során csillagpontos topológiájú hálózat épült ki, amelynek központja a gépteremből (ma: társalgó) fémvázás farostlemez fallal és zárható ajtóval leválasztott szerverterem volt.⁵ Innét ágazott ki a gigabites gerinchálózat, egy-egy

³A témáról részletesebben Lócsi Levente tanulmányában olvashatunk.

⁴Ez a router meglehetősen népszerű termék, mivel nagyon strapabíró és rengeteg egyedi firmware-t készítettek hozzá. A collegiumi karrierjük során is többször cseréltük rajtuk a szoftvert, kezdetben DD-WRT volt, majd Sztupák és Parragi rendszergazdák Tomato programra váltottak, amit azóta csak frissíteniünk kellett. Ezek a routerek nagyon sokoldalúak voltak, a tűzfalszerver meghibásodásakor például ideiglenesen az egyik ilyen eszközön szokott áthaladni a Collegium teljes adatforgalma. A Linksys WRT54GL széria sikerességét mutatja, hogy ez a típus még a cikk írásakor, 2014-ben is kapható a boltokban.

⁵A fal tulajdonképpen egy könyvespolc hátulja volt, mert a gépterem - és így a

vezeték a fiú- és a lányszárnyra, ahol a padláson egy-egy gigabites switch továbbosztotta a jelet öt-öt-felé: egy-egy vezeték a fiú és a lány harmadikra, valamint két-két vezetékpár a fiú és a lány második két pontjára - mindkét ponton a vezetékes és a vezeték nélküli eszközök külön kapcsolattal rendelkeztek. A szerverteremből haladt még egy vezeték az alsó, nem felújított hálózat felé is. A lakószárnyak folyosóin a vezetékes hálózat két-két nagy központi switch-csel és a két kisebb switch-csel rendelkezett, ezekbe volt bekötve szobánként egy kábel. A szobán belül egy kis switch osztotta meg ennek jelét a lakók között. A második emeleti folyosói switch-ek mellett egy-egy vezeték nélküli router kapott helyet.

Az újonnan kiépített vezetékek a folyosókon – a tűzvédelmi előírásoknak megfelelően – kábelcsatornáknak futottak, a szobák ajtait pedig (nem túl esztétikus) gégecsőben közelítették meg.⁶

A kábelfelújítás sajnos csak a lakószintekre terjedt ki, így igen fontos területeken maradt a régi rendszer: a titkárságon, a szeminárium termekben és az Estikében. A Lócsi Levente tanulmányában is említett, a földszint és az alagsor közti két érpárt használó megoldásnak az volt többek közt a hátulütője, hogy olyan 7 megabitnél nagyobb fogalmat nem tudott továbbítani, pedig ekkorra már az ADSL-vonal is 20 megabites, a belső hálózat pedig 100 megabites volt. Az alagsorból az Estikébe vezető úton pedig még az UTP elterjedése előtt használatos BNC-csatlakozós koaxiális kábel is áthaladt a forgalom, amelynek végein egy-egy mindkét típusú porttal rendelkező hub alakította a jelet UTP-jellé.

A következő jelentős esemény a szerverterem költözése volt 2009. október 30-án. A társalgó kialakítása miatt a szerverterem az alagsorba, a TMK műhely, a karbantartói főhadiszállás hátsó termébe kényszerült - itt található ma is. Viszont, mint az előző bekezdésből látszik, a teljes hálózat központja a szerverterem volt. Mivel a csillagpontos rendszer közepe költözött, a padláson levő switch-ekhez és a gépteremhez új kábelt kellett fektetni, illetve az ADSL telefonvonalat is le kellett vezetni az alagsorba. Az új kábelek a padlásról a lányszárny sarkán

szerverterem is - raktárhelyiségként egyúttal otthont adott a Mednyánszky könyvtár állománya egy részének is.

⁶Külön érdekes volt az a több méteres gégecső, amely a folyosó falán levő képek közt kanyarogva vezette be a konyhába az internetet. Annyira szükségesnek nézett ki, hogy Parragi Zsolt remekművét több évig csodálhattuk a falon, és csak a nagyfelújítás előtt szedtem le, hogy kinyerjem belőle a kábelt.

fűtési csövek mellett futottak. A költözésnek volt jó oldala is: miután a korábban a földszintet és az alagsort ellátó régi kábel már teljesen funkcióját veszítette, Ölvédi Tibor egyszerűen kihúzott a szerverteremből egy közvetlen kábelt az alagsor meglévő hálózatához, ezzel növelve az alagsori és az Estikében található hálózat sebességét és megbízhatóságát, s biztosítva, hogy az Estikében újra játszhassa a zenét DJ YouTube.

Nem sokkal a költözés után az ADSL vonal is vérfrissítést kapott. Parragi és Ölvédi rendszergazdák jelezték a szolgáltatónak, hogy gyakran rossz a kapcsolat, ha le van terhelve, eldobja a csomagokat, valamint a sávszélesség is rosszabb lett, mint a költözés előtt. A kikerülő szerelő méréseket végzett, és megdöbbsen, hogy ilyen kábelen egyáltalán működik az internet. Ugyanis a költözéskor a második emeleti szerverteremből az alagsorba költözött az ADSL-modem is, így a bejövő telefonvonal felment az egykori szerverterembe, majd onnan a padláson át még jónéhány tíz méter hosszan vezetett a szerverterembe. Ilyen hosszúságon a kábel már annyi zajt összeszedett, hogy nagy leterheltségnél hibásan működött. A szerelő bekötött egy jobban szigetelt telefonkábel közvetlenül a szerverterembe (és a biztonság kedvéért kicserélte a modemet), és innentől aztán az ADSL modem már stabilan tartani tudta a 20 megabit fölötti sebességet. (25 megabit körül volt a bejövő telefonsínór sebessége, ez az épületen belül kb. 22-re csökkent.) A feltöltés azonban az ADSL technológia miatt továbbra is nagyon lassú maradt. Ez volt az ADSL-en elvégzett utolsó fejlesztés.

2. Hol még csak tervben volt

2.1. A nagy felújítás

Az internet lassúsága nem csak a collegistáknak nem tetszett, Horváth László igazgató úr is elégedetlen volt, különösen a földszinti vezetékeken gyakori „meghibásodások” és az igazgatói irodát is érintő internetszünetek miatt. Persze e leállások kisebb része történt az eszközök tényleges meghibásodása miatt, sokkal gyakoribb volt, hogy valamelyik teremben kihúztak egy switch-et vagy egy kábelt, ami miatt megszakadt az összeköttetés. Még rosszabb volt a helyzet, amikor nem kihúztak, hanem bedugtak egy kábelt, ugyanis ha a rendszerben kör keletkezett, azt az olcsó, otthoni használatra szánt hálózati eszközeink nem tudták kezelni. Ilyenkor az egész épületben elment a net, és

hosszasan kellett keresgélni a hiba forrását. Ezért igazgató úr elhatározta, hogy a 2010-ben a Collegium rendelkezésére álló 200 milliós felújítási támogatásból be kell kötni az üvegszálát, sőt, amennyiben belefér, a belső hálózatot is teljesen, szakszerűen fel kell újítani.

A tervezés 2010 nyarán kezdődött. Üvegszál-ügyben az egyeztetések eljutottak odáig, hogy a rácsatlakozás végre elérhető közelségbe kerüljön, sőt, a korábbi 100 megabit helyett 1 gigabites kapcsolatról születtek tervek.

Időközben azonban a többi felújítás költsége teljesen felemésztette a pénzt. Szerencsére ekkor az ügyünk az ELTE Informatikai Igazgatóságánál már annyira előrehaladott volt, hogy – látván a tényleg tarthatatlan állapotokat – úgy gondolták, érdemes erre áldozni az ELTE hálózatfejlesztési alapjából.

Az ELTE rendszergazdáit az is vezérelte, hogy a Collegium, rákötve az üvegszálra, az ELTENET részévé váljon, és egységesen kezelhető legyen. A legfontosabb további cél a telefonrendszer átalakítása volt: az üvegszálás kábelen keresztül lehetővé vált, hogy a Collegium IP-alapú telefonokon keresztül bekapcsolódjon az ELTE központi telefonhálózatába.

2.2. A végpontok tervezése

2011 januárjában igazgató úr és az ELTE-s rendszergazdák megkérték a Collegium rendszergazdáit, hogy készítsenek terveket a Collegium belső hálózatának felújítására.

A végpontok kialakítására, a finanszírozás mértékétől függően – ekkor még nem tudtuk, mekkora a rendelkezésre álló keretösszeg – Ölvédi Tiborral több tervet készítettünk.

A leggazdaságosabb terv az volt, hogy a felújítás során csak az alagsort építik ki. Azt tudtuk, hogy az alagsori hálózatot mindenképp teljesen fel kell újítani, egyrészt mert ez volt a legrosszabb állapotban, másrészt az ELTE-s rendszergazdák igénye is elsősorban az volt, hogy a dolgozókig eljussanak a telefonokkal és az internettel. Mindenképpen az alagsorba kerültek volna az üvegszálát fogadó eszközök és a telefonos rendszer eszközei is.

Az ELTE-s rendszergazdák managed switch-eket képzeltek el. Egy klasszikus switch úgy működik, hogy az összes beérkező jelet megosztja a kábelek közt, mintha minden bejövő kábel minden másik kábellel össze

lenne kötve. A managed switch-ek ezzel szemben olyan, professzionális hálózatok kiépítéséhez használt eszközök, amelyek végpontjai nemcsak egyszerűen megosztják egymást közt a jelet, hanem ez a megosztás teljes mértékben konfigurálható is: például beállítható, hogy melyik végpont pontosan melyik végponttal legyen összekötve, melyik végponton milyen forgalmat engedélyezünk stb. Az ELTE-s managed switch-ek továbbá egy egységes hálózatot is alkotnak, így egyszerűen központilag irányíthatók. A managed switch-ek magukat a csomagokat is sokkal bonyolultabban kezelik, így ha „körbe kötjük” a rendszert, azt képesek felismerni, ezáltal nem bénul le a kollégium teljes rendszere. Annak, hogy mindeközéig miért nem managed switch-eket használtunk, főként anyagi okai voltak.

A közepes terv a lakószintek felújítását is tartalmazta. Minden szobába egy aljzat került volna, amit egy folyosói managed switch kezelne, és ahogy eddig is, minden szobában egy ott elhelyezett kis switch osztaná tovább az internetet.

A legnagyobb költségvetésű – és végül megvalósult – elképzelés pedig az volt, hogy minden szobába három aljzat kerül (remélhetőleg nem egymás mellé) és minden collegistának lesz saját csatlakozója.

Az ELTE-sek átszámolva a költségeket arra jutottak, hogy az utóbbi terv a köztes megoldáshoz képest nem okoz jelentős költségnövekedést. Mint megtudtuk a kábelezés többletköltsége elhanyagolható, lényegében a plusz két managed switch-et kell megfizetni, amelyekre a nagyobb végpontszám kiszolgálása miatt van szükség. Szobánként egy aljzattal 48 porttal, szobánként két aljzattal kétszer 48 porttal oldható meg egy-egy szárny végpontjainak ellátása.

Apró érdekesség, hogy a tervezés egybeesett egy belépőrendszerítával is. Volt olyan elképzelés is, hogy a collegisták maguk építenek beléptető-rendszert, amelyet a rendszergazdák felügyelnek. Ez egy számítógép (valószínűleg egy saját szoftvert futtató router) lett volna a kapunál, ami vonalkódokat olvasott volna és nyitotta volna az ajtót. A szép megoldás az lett volna, ha van összeköttetése a szerverekkel is. Ennek a tervnek a nyoma az az UTP-aljzat, amely a főbejárat külső és belső ajtaja között, bal oldalt található.

3. Végre célegyenesben

Az üvegszál bekötését azonban még mindig megnehezítette az a körülmény, hogy a kábel a Ménesi út túloldalán haladt. Az ELTE rendszergazdái szerencsére kitalálták a megoldást: a Bibó Kollégium pont az út jó oldalán fekszik, oda be lehet húzni a kábelt, és onnan már csak továbbítani kell a jelet a Collegiumba. Kezdetben voltak ugyan kétségeink ezzel a megoldással kapcsolatban – attól tartottunk, hogy ha a Bibóban áramszünet lép fel, nálunk sem lesz net –, de a tervek egy passzív eszközt javasoltak a Bibó Kollégiumba. Ennek köszönhetően – hacsak a fizikai kapcsolat meg nem szakad – a jel mindenképp megérkezik a Collegiumba. ELTE-s szemszögből óriási előnye ennek a megoldásnak, hogy így a Bibóban lakók is élvezhetik az üvegszálas internet előnyeit. A T-cégcsoportnak azonban még mindig nem volt szabad kapacitása a Ménesi út alatt, de szerencsére volt más megoldás is.

Ugyanis egy másik szolgáltató, a UPC kábeleit is átfutnak az út alatt, sőt a kábeltévé miatt ők is rendelkeztek beállással mindkét épületbe, így lehetséges alternatívának tűnt a két épület közötti távolságot a UPC infrastruktúráján keresztül áthidalni. A UPC szakemberével 2011. február 17-én közös bejárást tartottunk, ekkor ő úgy ítélte meg, hogy semmi akadály a kapcsolatteremtésnek.

3.1. Az új belső gerinchálózat kialakítása

Az új belső hálózat infrastruktúrájának kialakítása a következőképpen zajlott: először is szükség volt egy teremre, ahol el lehetett helyezni egy embermagasságú központi rackszekrényt, ami fogadja a bejövő optikai kábelt, és helyet ad a telefonrendszer központi felszereléseinek: egy speciális, 24 portos PoE switch-nek⁷, valamint egy szünetmentes áramforrásnak, hogy a telefonok áramszünet esetén is működképesek maradjanak. Erre a Collegium bejárati lépcsője alatti, 014-es terem volt a legalkalmasabb.⁸ Ide került az alagsori hálózat központja is: egy 48 portos switch és patch panelek az alagsori hálózat végpontjainak.

⁷A telefonkészülékek az UTP kábelben keresztül kapják az áramot is, ehhez speciális, Power over Ethernet (PoE) szabványt támogató switch szükséges.

⁸Ezzel a helyiséggel az a probléma még fennállt, hogy nem a Collegium, hanem az egykor itt lakó Murgács néni örököseinek tulajdonában volt, akik szerencsére hozzájárulásukat adták a terem ilyen célú használatához.

Ebből a teremből indul ki a két második emeleti rackszekrénybe a gerinckábel, amely az épületen belül is optikai kábel. A második emeleti rackszekrények kisebbek, bennük csupán két-két 48 portos managed switch és patch panel foglal helyet. A szekrények ventilátorokkal aktív hűtést is kaptak. Emiatt azonban hangosak, ezért szobába szerelésük nem volt opció, és sem a fürdő, sem a konyha, sem pedig a mellékhelyiségek előtere nem tűnt az elektronikus eszközök számára vonzó alternatívának, így kerültek a folyosóra.

4. Hol volt

4.1. A nagy felújítás

Az előző fejezetben említett nagy felújítás nemcsak azért érdekes, mert egy viszonylag nagy összeg állt a Collegium rendelkezésére, hanem azért is, mert az elektromos vezetékek cseréjéhez a teljes épületben falbontásra volt szükség. Ennél jobb időzítést elképzelni sem lehetett! Nagyon elegáns megoldásnak tűnt, hogy – ha már úgyis bontani kell a falat – a hálózati kábelek is bekerüljenek a falba.

A konkrét hálózat-felújítási tervek és a pénz viszont addig-addig késlekedtek, amíg az elektromos rendszer felújításának terveibe már nem lehetett bevenni az belső kábelek elhelyezését a falban. A belső hálózat kialakítása így csak azután kezdődött el, miután a befejezték a fal visszaépítését és a festést.⁹

Szerencsére arra volt lehetőség, hogy a hálózati infrastruktúrát kiszolgáló konnektorokat elhelyeztessük az elektromos felújítás idején. Zrupkó Erika, a Collegium gondnoka pontosan tolmácsolta az igényeinket az elektromos rendszer kivitelezői felé, akik a folyosók plafonja közelében konnektorokat építettek ki a WiFi-routereknek.¹⁰ Korábban a folyosókon nem voltak konnektorok, ezért az ott elhelyezett hálózati eszközök áramellátása kreatív megoldásokat igényelt. A második emeleti

⁹Ennél még rosszabb volt a helyzet a tűzjelző rendszer esetében. A tűzjelzőket összekötő kábelnek ugyanis mindössze azért kell kábelcsatornában futnia, mert habár az elektromos rendszer és a tűzjelző egyszerre épült, így lett kiírva a közbeszerzés, és a kivitelező ettől már nem térhetett el. Azért a kooperáció is jelen volt a felújítás során: a tűzjelző rendszer építése során a födémáttöréseket már úgy végezték, hogy az UTP-kábelkötegek is átférjenek.

¹⁰Bár folyosói rackszekrények ekkor még tervekben sem szerepeltek, később a konnektorok azok áramellátását is megkönnyítették.

fiúszárny végén levő WiFi-router kábele például a 232-es szobából volt kivezetve. Bár a szoba lakói általában tudtak erről, mindenesetre néha előfordult, hogy porszívózás vagy a szobát áramtalanító nyári vendég megzavarta az internet-ellátást.

4.2. Épül a strukturális hálózat

Az épületfelújítás utáni félév szeptemberében érkezett el a pillanat, amikor megkezdődött a belső strukturált hálózat kiépítése. A kivitelezésre kiírt közbeszerzési eljárást megnyerő KFKI (a T-cégcsoport tagja, nem a kutatóintézet) végezte. Nagyon gyorsan haladtak – legalábbis a korábbi, collegisták általi kétkezi építési tempóhoz képest mindenképp. Habár az időpont nem volt a legmegfelelőbb, mivel a szerelési munkálatok idején már az épületben tartózkodtak a collegisták, de az ezzel járó kis kellemetlenségeket (néhány kábelcsatorna¹¹ és UTP-aljzat felfúrását) a jobb internet reményében mindenki elviselte. Masszív, időtálló (igaz, picit bumfordi) csatlakozók kerültek a falra – az ELTE rendszergazdák a közbeszerzés kiírásánál erre is ügyeltek.

Már a kábelezés során felkerültek a falra a folyosói rackszekrények és a patch panelek, a switch-ek beszerzése és beszerelése csak később történt meg. 2011. október 20-án kezdtek működni az aktív eszközök a folyosókon – a ventilátorok zümmögő hangja ettől a pillanattól vált részévé a collegiumi mindennapoknak.

4.3. Az üvegszál bekötése

2011. november 4-én, egyik napról a másikra megtörtént az, amire már közel egy évtized óta vártunk: bekötötték a nagysebességű, üvegszálal internetet.

Az új internetkapcsolat először még csak a szerverteremben működött, ezért első feladatunk volt a szervereket áthelyezni az új kapcsolatra. Így a szolgáltatások (DNS, honlapok stb.) stabil internetkapcsolatot kaphattak, és ezáltal is csökkenthettük a régi rendszer leterheltségét.

Időközben, nem sokkal a belső kábelezés kiépítése után, 2011 szeptemberének végén Ölvedi Tibor helyett Hapák József lett az új rendszergazda, akivel elkezdtük a szervereket virtuális gépekké alakítani az

¹¹A felújítás során a kábelek nemcsak a folyosókon, hanem a szobákon belül is kábelcsatornába kerültek.

infrastruktúra megbízhatóságának növelése és karbantarthatóságának egyszerűsítése érdekében. Ez nagyban segített a szolgáltatások folyamatos átállításában is. A helyzetet nehezítette viszont az, hogy a felújítás után nem sokkal (valószínűleg a felújítás által kavart por miatt) elromlott a legnagyobb szerverünk tápegysége.¹² A gép meghibásodása miatt 8 TB tárhely mellett 8 GB memóriától is elestünk, utóbbi az üvegszálalás hálózaton üzemelő új virtuális gépek kialakításánál nagyon jól jött volna.

Az ELTE-s rendszergazdákcal már korábban úgy egyeztünk meg, hogy az IP-címek kiosztását továbbra is a Collegium szerverei végzik majd. A kiosztandó IP-tartományt már a bekötés napján megkaptuk, a pontos részleteket 2011. november 16-án beszéltük meg. A régi hálózaton a felhasználók egy belső IP-t kaptak, mivel az ADSL-kapcsolat csak egy 64 IP-s tartománnyal rendelkezett, így nem volt lehetőségünk arra, hogy minden collegistának külön IP-címet adjunk. Csak a szerverek rendelkeztek saját IP címmel, a külvilág szemszögéből felhasználók egy közös IP-n osztoztak. Az ELTE viszont az üvegszálon keresztül tudott számunkra adni egy 512 IP címből álló tartományt, így lehetővé vált, hogy minden collegista számára egyedi, kívülről elérhető IP-ket osszunk. A rendszert úgy állítottuk be, hogy az IP-címeket felhasználóhoz kötöttük, és emiatt csak félévek között változnak. Ezek az IP címek ráadásul az ELTE IP-cím tartományában vannak, így az egész világ oktatási IP-címkét tekint rájuk, valamint kerülőmegoldások nélkül elérhetjük az ELTE-n megrendelt digitális folyóiratokat és egyéb belső ELTE-s szolgáltatásokat – ilyen például a tanulmányi rendszer adminisztrációs felülete.

2011. november 20-án jött el a pillanat, hogy az üvegszálalás internet már a közvetlenül az ágyam mellett levő fali csatlakozóban is elérhetővé vált. Némi tesztelés után, a rákövetkező héten kezdtük el hirdetni a collegistáknak, hogy a regisztrációs díj befizetése és a számítógép hálózaton való regisztrációja után mindenki számára lehetőség nyílik a fali aljzatokon keresztül a szélessávú internetet használatára.

¹²Amikor megláttuk, hogy mennyibe kerülne cserélni úgy döntöttünk, inkább alternatív megoldások után nézünk. A javíttatás elég lassan haladt, ezért inkább vettünk egy szabványos tápegységet, és miután láttuk, hogy azzal is működik a gép, Hapák József rögzítette a szerver házában az új alkatrészt. A megoldás a mai napig problémamentesen teszi a dolgát.

4.4. Számokban

Az üvegszálás internet sebessége többszöröse lett a korábbi, ADSL-alapú megoldásának (összehasonlításukat lásd az 1. táblázatban). A letöltésünk 50-szeresére, míg feltöltés sebessége még drasztikusabban, 2000-szeresére nőtt.

	Letöltés	Feltöltés
Régen	20 Mbit/sec	0,5 Mbit/sec
Most	1000 Mbit/sec	1000 Mbit/sec
Gyorsulás	50-szeres	2000-szeres
1 Linux telepítő DVD régen	32 perc	21 óra
1 Linux telepítő DVD most	38 másodperc	38 másodperc

1. táblázat. A collegiumi internet le- és feltöltési sebességének összehasonlítása az üvegszálás kapcsolat bekötése előtt és után.

Jól tudjuk, hogy az internet minőségének az adatátviteli sebesség mellett fontos jellemzője a válaszidő (ping) is, azaz az idő, ameddig egy csomag és a rá kapott válaszcsoomag megteszi a gépünk és a szerver közötti távolságot oda-vissza. A válaszidő mérését nagy magyar weboldalak szervereivel érdemes végezni, külföldi szerverek esetén – fizikai korlátok miatt – nyilván nagyobb értéket tapasztalhatunk. A régi hálózat esetén a válaszidő 10-12 ezredmásodperc körül alakult – ez egy tipikus érték ADSL-kapcsolatok esetén. Nagy leterheltségnél azonban ez az érték több száz ezredmásodpercre nőtt. Az üvegszálás kapcsolat esetén a válaszidő 1 ezredmásodperc marad, ami kisebb, mint a régi érték tizede, és ezt az alacsony értéket a rendszer csúcsidőben is tartja.

4.5. A további feladatok

2012. január 10-én a levelezőszerver is átkerült az új hálózatra. A Collegium levelezését továbbra is az épületen belül tároljuk, viszont minden levelünk az ELTE-s központi levelezőszervereken halad keresztül. Ez például olyan előnnyel jár, hogy kisebb eséllyel kerülnek véletlenül a spam mappába a collegiumi szerver által küldött levelek.

2012 szeptemberében került sor az alagsori csatlakozók beüzemelésére. Több felmerülő probléma megoldása után már csak az aljzatok bekötési tervére volt szükség. Az alagsori központban ugyanis, mint

korábban említettem, egy 24 portos PoE switch volt a telefonoknak és egy 48 portos switch a többi aljzatnak. Ez viszont nem volt elég az összes kiépített aljzat (kb. 90 darab, a telefonoknak szánt végpontokkal együtt) ellátásához. A problémát enyhíti, hogy a telefonok hátuljába számítógép csatlakoztatható, valamint hogy a telefonoknak szánt 24 portos switch a telefon nélküli aljzatokat is el tudja látni – igaz ezek csak 100 megabites portok. A végső tervnél arra törekedtünk, hogy minden terembe jusson el a vezetékes net. Mivel az aljzatok a termekbe általában párosával kerültek kiépítésre, ezért ez azt jelentette, hogy a termenkénti két aljzathoz általában az egyikben találhatunk internetet. A be nem kötött aljzatokat piros szigetelőszalaggal ragasztottuk le.

2012. október 10-én tartottuk az üvegszál ünnepélyes átadását. Az igazgató úr által szervezett rendezvényen röviden elmeséltem a bekötés történetét és Varga Klára, az Egyetemi Könyvtár Informatikai és Fejlesztési Osztályának vezetője bemutatta az egyetemi könyvtár által nyújtott digitális folyóiratokat, amelyeket a Collegium épületében már mindenféle kerülő megoldás (stunnel stb.) igénybevétele nélkül elérhetünk.

2013. február 28-tól a regisztrált felhasználók már a vezeték nélküli hálózatot is az üvegszálkapcsolaton keresztül érhetik el. Ehhez egy speciális tűzfalat telepítettünk, amely a régi belső hálózatot összeköti az új külső hálózattal, valamint a forgalom szűrését is elvégzi. A vezeték nélküli hálózaton továbbra is csak néhány port engedélyezett, mert a vezeték nélküli routerek nem bírnák el, ha mindenki kizárólag vezeték nélkül használná az internetet (akkor ugyanolyan lassú lenne, mint régen). Így aki szeretné többi portot is használni, annak a vezetékes hálózatot kell választania.

2013. április 3-tól a teljes forgalom az új hálózaton keresztül megy.

2013. július elején szűnt meg a régi ADSL-kapcsolatunk. Nem sokkal később elvitték a szerverteremből a régi ADSL-kapcsolat eszközeit. Ezzel lezárult egy korszak, azóta egy új, gyorsabb internet áll az Eötvös Collegium tagjainak rendelkezésére.

Nagy méretű gráfok feldolgozása Hadoop és Giraph rendszerek segítségével

Balassi Márton

Eötvös József Collegium*

balassi.marton@gmail.com

Az adatintenzív, elosztott batch feldolgozó keretrendszerek, mint a Google által publikált MapReduce [1] és annak nyílt forráskódú implementációja, az Apache Hadoop [5] csak korlátozott mértékben alkalmasak gráfalgoritmusok párhuzamosítására. Ezek a feladatok MapReduce programok egymás után láncolásával valósíthatók meg, de a keretrendszer architektúrájából kifolyólag felesleges I/O műveleteket eredményeznek. A feladat egy javított megoldását teszi lehetővé a Pregel [10] rendszer, mely speciálisan gráfok feldolgozására készült. Utóbbi képes a teljes algoritmus alatt a klaszter memóriájában tartani a gráf szerkezetét, így lényegesen kisebb futási időt eredményez. Utóbbi Apache implementációja a Giraph [4] projekt.

Jelen tanulmányban rövid bemutatásra kerül mindkét modell, majd a háromszögszámolás algoritmusának implementációja mindkét keretrendszerben és referencia eredményként szekvenciálisan is. Az elkészült programok futási ideje 6 különböző méretű gráfon, a párhuzamosítás különböző mértékei mellett kerül összehasonlításra. Általános megfigyelés, hogy megközelítőleg egymillió csúcsú gráfok esetén már előnyösebb az elosztott megvalósítás a szekvenciálishoz képest, miközben a Giraph rendre jobban teljesít, mint a Hadoop. A programkódok és mérések egy nagyobb lélegzetvételű mérésorozat részét képezik, melyet szerzőtársaimmal publikáltunk [2].

*2009–

1. Elosztott adatfeldolgozás

A feldolgozandó adatmennyiség növekedésével egyre nagyobb szerep jut az elosztott feldolgozó keretrendszereknek az adatintenzív feladatok megoldásában. Ezek tipikusan egy elosztott fájlrendszerből és egy feldolgozó rétegből állnak. A nyílt forráskódú közösség körében általánosan elterjedt elosztott fájlrendszer a HDFS [6], míg a feldolgozó motor rétegre különböző megoldás született, tipikusan eltérő előnyökkel és hátrányokkal. Jelen tanulmányban a gráfalgoritmusok szempontjából egy példán keresztül vizsgáljuk ezeket a megoldásokat.

1.1. A MapReduce programok szerkezete

Ma már klasszikusnak számít az elosztott adatfeldolgozás területén a szavak frekvenciájának kiszámítására vonatkozó `WordCount` példaprogram, egyrészt egyszerűsége, másrészt az invertált indexépítési feladat megoldásakor betöltött szerepe miatt.

Ez a program egy dokumentumhalmazt kap bemenetként, kimenetként pedig minden egyes szóhoz előállítja az dokumentumokbeli összes előfordulásainak számát. Fontos szerep jut annak a ténynek, hogy a MAPREDUCE paradigma kulcs-érték párokon operál. Ennek megfelelően a mind a **Map**, mind a **Reduce** fázisra igaz, hogy bemenetként és kimenetként is kulcs-érték párokat vár. Implementációt tekintve JAVA osztályokkal dolgozunk, így például az 1. ábrán látható módon valósíthatjuk meg az algoritmust.

Ahogy az a pszeudokódban látható a **Map** fázis bemenetként egy dokumentum egy sorát kap meg melyet a HDFS-ről olvasunk, tokenizálja azt, majd az egyes tokenekhez kibocsát egy integer 1 értéket. A MAPREDUCE keretrendszer biztosítja, hogy az azonos kulcsokhoz tartozó értékek egy **Reduce**-hoz kerüljenek¹ [13]. Emiatt a második lépésben képesek vagyunk megszámolni az egyes tokenekhez tartozó összes előfordulást.² Kimenetként minden **Reducer** a HDFS-re ír, mely fájlok összességét transzparensen egy dokumentumként is elérhetjük.

Az algoritmus szerkezetéből következik, hogy a **Reduce** nem kezdődhet el egészen addig, amíg a **Map** fázis teljes egészében be nem fejeződött,

¹Erre a köztes **Shuffling** lépésre gondolhatunk intuitíve elosztott **groupBy**-ként.

²Ez a megvalósítás igen hatékonytalan, mivel feleslegesen sok köztes adattal dolgozik, de példának kiváló.

```

1: class Mapper
2: function MAP((line_id  $id$ , line  $l$ ))
3:   for all term  $t \in$  line  $l$  do
4:     function EMIT(( $t$ , 1))
5:   end class
6:
7: class Reducer
8: function REDUCE((term  $t$ , counts  $[c_1, c_2, \dots, c_k]$ ))
9:    $sum \leftarrow 0$ 
10:  for all count  $c \in$  counts  $[c_1, c_2, \dots, c_k]$  do
11:     $sum \leftarrow sum + c$ 
12:  function EMIT(( $t, sum$ ))
13: end class

```

1. ábra. WordCount példa [9] 23. oldala nyomán

ugyanis a keretrendszer tervezésekor klaszterkörnyezetet vettek figyelembe, nem pedig a grid rendszerek hardvereinek sokszínűségét, mely az egyik lehetséges oka a különböző végrehajtási időnek [7].

Fontos megjegyeznünk, hogy a **Mapper** és a **Reducer** osztályok slave-enként egyszer jönnek létre, így ezzel az enkapszulációval potenciálisan csökkenthető egy slave-en történő párhuzamos végrehajtás esetén a memóriahasználat.

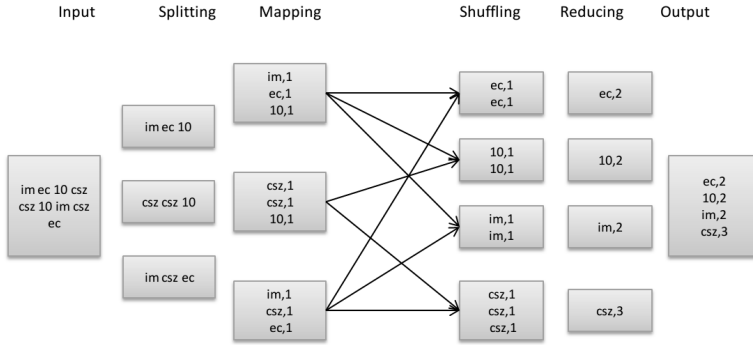
Tekintsük át a következő példán, hogy az algoritmus egyes lépéseiben mi történik egy bemenettel (2. ábra).

Természetesen MAPREDUCE programokat egymás után láncolhatunk – ekkor egy job kimenete a soronkövetkező bemenete lesz –, így bonyultabb feladatok megoldására is alkalmas a rendszer.

2. Háromszögek számának kiszámítása

A továbbiakban nagy méretű irányított gráfok háromszögeinek számát szeretnénk meghatározni elosztott környezetben. A programot megvalósítjuk HADOOP és GIRAPH³ keretrendszerekben, melyek futási eredményeit összevetjük a szekvenciális algoritmusával. A programkódok

³Specifikusan gráffeldolgozó elosztott keretrendszer, előnyeit a fejezet során röviden tárgyaljuk.



2. ábra. WordCount példafuttatás

és a mérések saját munkámat képezik, melyet egy méréssorozat részeként szerzőtársaimmal publikáltunk [2].

2.1. Gráfalgoritmusok elosztott környezetben

Amennyiben gráfalgoritmusokat elosztott környezetben szeretnénk kezelni feltehetjük, hogy az adat puszta mennyiségénél fogva egy-egy fizikai gép nem fog rendelkezni csak a gráf szerkezetének egy részével és jellemzően ritka gráfokról beszélünk, emiatt a hagyományos szomszédsági és incidenciamátrix alapú reprezentációk nem jönnek szóba. A lista alapú reprezentációk közül a szomszédsági lista megfelelő számunkra, mely során egy csúcshoz eltároljuk a szomszédait, például soronként a következő formátumban:

$$node, neigh_1, \dots, neigh_n$$

Ebben az esetben a csúcst kulcsként tudjuk használni a MAPREDUCE keretrendszerben. Így viszont egyetlen gép memóriájában sem jelenik meg a gráf teljes egészében, kénytelenek vagyunk módosítani az algoritmusok megvalósításán is. Például egy mélységi keresés során egy csúcs szomszédainak szomszédairól már nincsen információnk, így ahhoz hogy iteratív algoritmusok futtatni tudjunk a gráfon több jobra

lesz szükségünk. Tulajdonképpen a **Map** fázisban a csúcsok üzenhetnek a szomszédaiknak, mely üzeneteket az egyes csúcsok a **Reduce** fázisban aggregálják.

Ahogy az 1.1. alfejezetben kiemeltém, MAPREDUCE programok egymás után láncolása esetén minden lépésben a HDFS-ről olvasunk és írunk, ezzel jelentős I/O költséget eredményezve. Ennek nagy része felesleges, ugyanis minden lépésben beolvassuk és kiírjuk a gráf szerkezetét, szerencsésebb lenne, ha legalább ezt a speciális adatot memóriában tartanánk. Ebből a feltételezésből indul ki a GOOGLE PREGEL rendszer [10], melynek nyílt forráskódú implementációja APACHE GIRAPH néven elérhető [4]. Utóbbi a gráf memóriában tartásán felül lehetővé teszi, hogy mind a csúcsokat mind az éleket felcímkézzük (akár egészen összetett adatstruktúrákkal, mintegy állapotot adva nekik), programozáskor csak a csúcsok között küldendő üzeneteket kell megvalósítanunk, jelentősen csökkentve munkánkat.

2.2. Szekvenciális referencia implementáció

A keretrendszerek hasznosságának vizsgálata érdekében az algoritmust megvalósítottam szekvenciálisan JAVA nyelven. Itt a korlátozott mélységű keresésre [11] támaszkodva összeszámolom a pontosan 3 mélységben előforduló, a kezdőcsúccsal megegyező azonosítójú csúcsokat. Ekkor minden háromszöget pontosan háromszor számolunk össze.

```

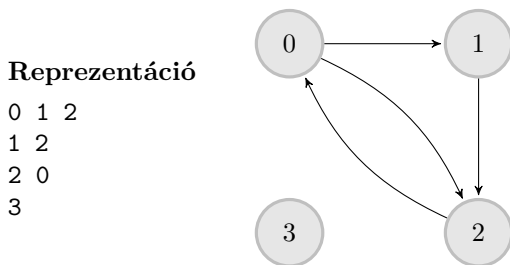
1: class SeqTriangleCounter
2: function COUNTDEPTHINSTS(node s, node t, int n, graph g)
3:   if n = 0 then
4:     if s = t then
5:       return 1
6:     else
7:       return 0
8:   for all term t ∈ line l do
9:     function EMIT(t, 1)
10: end class

```

3. ábra. Korlátozott mélységű keresés

2.3. Háromszögszámolás MapReduce keretrendszerben

MAPREDUCE keret használatával jelentősen elbonyolódik az algoritmusunk, melynek elsődleges oka, hogy csak lokális információnk van a gráf szerkezetéről, ahogyan azt már a 2.1 tárgyaltam. Így két lépésre kell bontanunk az algoritmust. Használjuk szemléltető példaként a következő gráfot:



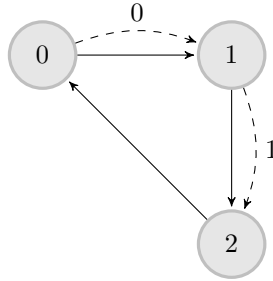
4. ábra. Példagráf az algoritmus szemléltetéséhez

2.3.1. Kétirányú éllista előállítása

Első lépésben elő kell állítanunk a kettő hosszú utakat, ennek egy lehetséges módja, ha minden csúcsban előállítjuk a ki-szomszédian kívül a be-szomszédait is tartalmazó kétirányú éllistát. Az információáramlást csökkentendő elég, ha minden csúcs csak a nála nagyobb azonosítójú szomszédainak küldi el az azonosítóját⁴. A példánkban ez a következőt jelentené, a többletinformáció áramlást szaggatott nyíllal jelöljük (5. ábra).

A pszeudokódot tekintve azonban ezt az ábrán láthatónál némileg bonyolultabb módon érhetjük el, ugyanis kénytelenek vagyunk redundáns módon a gráf szerkezetét is tovább küldeni az új, a fordított éleket tartalmazó információval együtt. Éppen emiatt kénytelenek vagyunk az $\{in, out\}$ halmazból választott jelölőbittel meghatározni, hogy melyik irányú élről van éppen szó.

⁴Minden háromszöget pontosan egyszer fogunk megszámlálni az algoritmus során ezzel a módszerrel.



5. ábra. BiDirectionMap lépés szemléletesen

```

1: class BiDirectionMap
2: function MAP(vertex  $v$ , neighbors  $[n_1, n_2, \dots, n_k]$ )
3:   for all vertex  $n \in$  neighbors do
4:     function EMIT( $v$ , ( $n$ , out))
5:     if  $ID(n) > ID(v)$  then
6:       function EMIT( $n$ , ( $v$ , in))
7: end class

```

6. ábra. BiDirectionMap pszeudokód

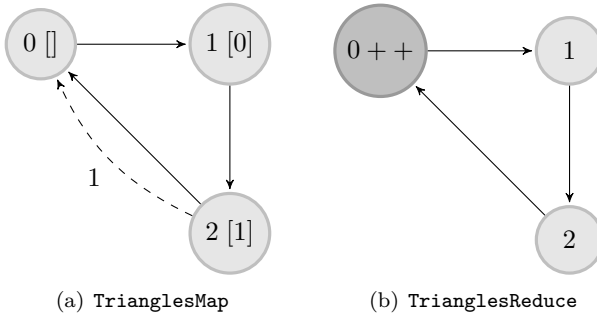
A **Reducer** szerepe mindössze abban merül ki, hogy a kapott éleket a jelzőbit szerint csoportosítva írja a kimenetre.

2.3.2. Háromszögek ellenőrzése

Ebben a lépésben a 2.3.1 fejezetben leírt módon előállított „háromszög jelöltekről” döntjük el, hogy valóban zárt vagy nyílt háromszögek-e. Az ellenőrzendő csúcsokat listaként, az információáramlást nyilakkal vizualizálva a következő feladatot hajtjuk végre a példabemeneten:

A **TrianglesMap** során a csúcsok a kapott *in* jelzőbitű információt továbbküldik, ha az kisebb azonosítót jelöl, mint a sajátuk. Továbbá elküldik a ki-szomszédait saját részükre. A **TrianglesReduce** lépésben a csúcsok megnövelnek egy globális számlálót minden olyan kapott azonosítóra, amely szerepel a szomszédsági listájukban.

Így a globális számlálóba éppen a háromszögek száma kerül



7. ábra. A háromszögek ellenőrzésének lépései

```

1: class TrianglesMap
2: function MAP(vertex  $v$ , edges  $[(n_1, in), (n_2, in), \dots (n_k, out)]$ )
3:   outneighbors := [ ]
4:   info := [ ]
5:   for all edge  $e \in$  edges do
6:     if  $e[2] = out$  then
7:       outneighbors.append( $e[1]$ )
8:     else
9:       info.append( $e[1]$ )
10:  for all vertex  $n \in$  outneighbors do
11:    for all vertex  $i \in$  info do
12:      function EMIT( $n, i$ )
13: end class

```

8. ábra. TrianglesMap pszeudokód

az algoritmus terminálásakor. Megjegyzem ehelyett a lépés helyett használhatnánk egy MAPREDUCE jobot, mely egyszerűen a számlálás programozási tételét valósítja meg, de ezzel csak tovább rontanánk ennek az implementációnak a futási idejét.

2.4. Háromszögszámolás Pregel keretrendszerben

Annak érdekében, hogy elkerüljük a 2.3 fejezetben tárgyalt kellemetlenségek jelentős részét az elosztott gráffeldolgozásra inkább alkalmas

PREGEL keretrendszerben is megvalósítjuk az algoritmust.

Ezen keret mottója, miszerint „gondolkozzunk úgy, mint egy csúcs” jól kiemeli a használatakor alkalmazandó gondolkodási keretet. Mivel a keretrendszer gondoskodik a gráf szerkezetének memóriában tartásáról programunk írásakor csupán a tényleges üzenetküldésekkel kell foglalkoznunk, miközben jelentős I/O terheléscsökkenést érünk el.

2.5. Implementáció

Láttuk a 2. fejezet során, hogy mennyire eltérőek az egyes implementációk egymástól. Amíg a szekvenciális esetben „felülről nézzük” a gráfot, globális információnk van róla, ha elosztottan vizsgálódunk kénytelenek vagyunk lokális információra szorítkozni. Az elosztott lehetőségek közül a GIRAPH modellje lényegesen kényelmesebb gondolkodási sémát biztosít. Ugyan nem feltétlenül mérvadó, de tájékoztatás jelleggel álljon itt az egyes implementációk hossza.

Keret	Implementáció hossza
szekvenciális	42 sor
HADOOP	273 sor
GIRAPH	130 sor

1. táblázat. Az egyes implementációk hossza

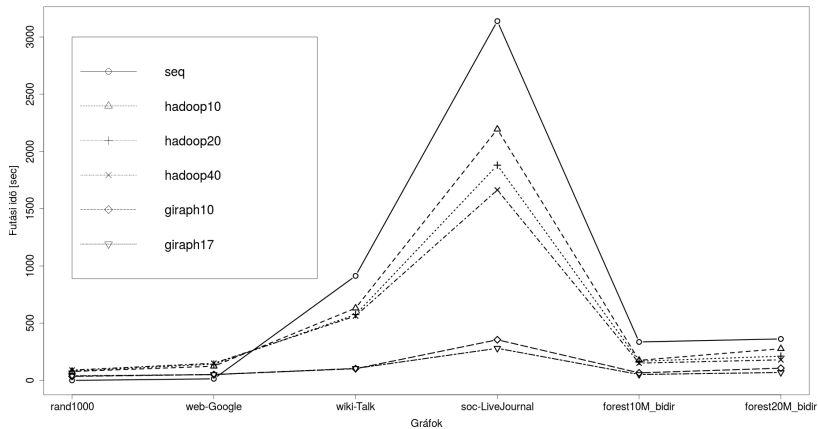
3. Futási eredmények

Az elkészült algoritmusokat a következő gráfokon futtatva vizsgáltam meg. A `rand1000` egy 1000 csúcsú Erdős-Rényi gráf [3], melyet elsősorban tesztelési céllal használtam. Három empirikus szociális hálózatot tekintettem, melyeknél a forrást „SNAP”-ként jelöltem meg, ezek a Stanford Large Network Collection-ből [12] származnak. A két legnagyobb hálózat szintén generált, ezek az úgynevezett Erdőtűz modellel [8] készültek, mely a szociális hálózatokhoz hasonló szerkezetű gráfokat eredményez, igaz esetünkben ezeknél jelentősen ritkábbakat.

Ezen gráfokon referenciaként futtatva a szekvenciális algoritmust, majd a HADOOP és a GIRAPH implementációkat rendre 10, 20, 40, illetve 10 és 17 magon az alábbiakat tapasztaltam.

Gráfnév	Csúcsszám	Élszám	Forrás
rand1000	1000	9440	generált
web-Google	$8,7 \cdot 10^5$	$5,0 \cdot 10^7$	SNAP
wiki-Talk	$2,4 \cdot 10^6$	$5,1 \cdot 10^7$	SNAP
soc-LiveJournal	$4,8 \cdot 10^6$	$6,9 \cdot 10^8$	SNAP
fores10M_bidir	10^7	$2,4 \cdot 10^8$	generált
fores20M_bidir	$2 \cdot 10^7$	$4,8 \cdot 10^8$	generált

2. táblázat. A felhasznált gráfok és paramétereik



9. ábra. Háromszögszámolás futási ideje

Kis méretű adatokra természetesen a szekvenciális megoldás a leghatékonyabb, azonban 10^6 csúcsszámú gráfok esetén már mind a HADOOP mind a GIRAPH implementáció jobban teljesít, mint az egy szálon futó algoritmus. A GIRAPH speciálisan gráffeldolgozó keretrendszer lévén mindig jobban teljesít, mint a HADOOP. Kevésbé nyilvánvaló, de az elosztott keretrendszerek esetén általában található a párhuzamosságnak egy olyan optimális mértéke, mely után már nem javítunk a futási időn újabb magok hozzáadásával, sőt ronthatunk is azon.

4. Összefoglalás

Tanulmányomban a HADOOP és a GIRAPH keretrendszereket vizsgáltam elosztott gráfalgoritmusok megvalósításának céljából. Vizsgálatomat egy igen egyszerű példán, a háromszögszámításon keresztül mutattam be, összehasonlítási alapként megvalósítottam az algoritmust szekvenciálisan is. Azt tapasztaltam, hogy az elosztott algoritmusok lényegileg különböznek a szekvenciális megvalósítástól. Hat különböző méretű gráfon futtatva azt tapasztaltam, hogy egymillió csúcsú gráfok esetén már megéri elosztott módon implementálnunk az algoritmust.

Hivatkozások

- [1] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, 2004.
- [2] P. Englert, M. Balassi, B. Kósa, A. Kiss, Efficiency issues of computing graph properties of social networks, Presented at The 9th International Conference on Applied Informatics, Eger, 2014.
- [3] P. Erdős, A. Rényi, On random graphs, *Publicationes Mathematicae*, 1959, pp. 290–297.
- [4] Apache Giraph Project: Official webpage, 2014. január, <https://giraph.apache.org/>
- [5] Apache Hadoop Project: Official webpage, 2012. november, <http://hadoop.apache.org/>
- [6] Apache HDFS: The hadoop distributed file system: Architecture and design, 2012. november, http://hadoop.apache.org/docs/r0.17.1/hdfs_design.html
- [7] L. Chuck, *Hadoop in Action*, Manning, 2011.
- [8] J. Leskovec, J. Kleinberg, C. Faloutsos, Graph evolution: Densification and shrinking diameters, *ACM Trans. Knowl. Discov. Data*, 1(1) (2007), <http://doi.acm.org/10.1145/1217299.1217301>

- [9] J. Lin, C. Dyer, *Data-Intensive Text Processing with MapReduce*, Morgan & Claypool Publishers, 2010.
- [10] G. Malewicz, M. H. Austern, A. J. C Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: A system for large-scale graph processing, *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, New York, NY, USA, 2010, ACM, pp. 135–146.
- [11] S. Russell, P. Norvig, *Artificial Intelligence – A Modern Approach*, Prentice Hall, 2010, pp. 88–90.
- [12] Stanford University: Stanford large network dataset collection, 2014. április, <http://snap.stanford.edu/data/>
- [13] J. Venner, *Pro Hadoop*, Apress, 2009.

Gépi tanulás gráfokkal

Bodó Zalán

Farkas Gyula Szakkollégium
Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar
zbodo@cs.ubbcluj.ro

1. Bevezetés

A gráfelmélet a matematika és számítástudomány egyik fontos ága. Az első, 1736-ban publikált – a königsbergi hidak problémájával foglalkozó – gráfelméleti tanulmány Leonhard Euler nevéhez fűződik. Azóta a gráfelmélet jelentős tudományággá nőtte ki magát, melynek alkalmazásaival szinte minden tudományterületen találkozhatunk. A sok fontos alkalmazás közül itt mindössze egyet emelnénk ki: a számítógépes hálózatokban a routerek gráfelméleti algoritmusokat használnak annak eldöntésére, hogy milyen *optimális* útvonalon továbbítsák a fogadott csomagot.

Gráfokkal a gépi tanulásban is találkozhatunk. A gráf alapú tanuló algoritmusok bemenetei nem maguk a tanulási adatok lesznek, hanem az adatok felett értelmezett *hasonlósági gráf*, amely alapján sok fontos következtetés vonható le.¹ A tanulmányban bemutatásra kerül a gráf alapú tanuló módszerek egyik központi fogalma, a Laplace-mátrix, melynek röviden ismertetjük néhány fontos tulajdonságát. Ezután három konkrét, gráfokat használó algoritmust mutatunk be: a spektrális klaszterezést, a Laplace típusú regularizált legkisebb négyzetek módszerét és a címkepropagálást. Az algoritmusok ismertetése után

¹Habár jelen tanulmány csak irányítatlan gráfokkal foglalkozik, léteznek irányított gráfokon alapuló módszerek is. Egy példa erre a [10] cikkben bemutatott irányított páros gráfokon alapuló félig-felügyelt algoritmus.

azok működését szemléltetjük kisebb adathalmazokon. A tanulmányban bemutatottak megértéséhez mindössze alapvető matematikai és a gépi tanulással kapcsolatos fogalmak ismerete szükséges.

2. Jelölésrendszer

Jelen tanulmányban az alábbiakban bemutatott jelölésrendszert használjuk. A skalárokat egyszerű római vagy görög betűkkel jelöljük, például a, b, β, λ . A félkövérrrel szedett kisbetűs entitások vektorokat $(\mathbf{u}, \mathbf{x}, \mathbf{y}, \dots)$, a nagybetűsek pedig mátrixokat jelölnek $(\mathbf{A}, \mathbf{L}, \mathbf{X}, \dots)$. Az $\mathbf{A}$ mátrix transzponáltját $\mathbf{A}'$ -vel jelöljük. Az $\mathbf{1}$ vektor az 1-eseket tartalmazó vektort jelöli, $\mathbf{I}$ pedig az egységmátrixot. Ezek méretét nem jelöljük, a kontextus alapján az mindig kiolvasható lesz. A tanulmányban használt $\|\cdot\|$ norma az euklideszi normát jelenti.

A bemutatásra kerülő algoritmusokban címkézetlen és címkézett adatokkal fogunk dolgozni. Címkézetlen adataink klaszterezés esetén vannak, címkézettek pedig osztályozási feladatokban jelennek meg. Adataink d -dimenziós valós vektorok, és ezeket általában $\mathbf{x}$ -szel, illetve adott sorrend esetén $\mathbf{x}_i$ -vel jelöljük, $\mathbf{x}, \mathbf{x}_i \in \mathbb{R}^d$, $i \in \{1, 2, \dots, N\}$. Az adatainkat sorba rendezve, majd egy mátrix oszlopaiba helyezve megkapjuk a $d \times N$ méretű $\mathbf{X}$ adatmátrixot. Címkézett adatok esetén az adatok számát ℓ -lel jelöljük, viszont a bemutatott osztályozó algoritmusok félig-felügyelt módszerek, ami azt jelenti, hogy címkézett adataink mellett címkézetlen adatokat is használunk tanuláskor. Az adathalmaz címkézetlen részhalmazának méretét u -val jelöljük, a teljes tanulási adathalmaz mérete tehát $N = \ell + u$. A tanulmányban csak bináris (azaz kétosztályos) osztályozási feladatokról lesz szó, ezért a címkézett adatokhoz bináris címkék társulnak – ezeket y_i -vel jelöljük, $y_i \in \{-1, 1\}$, $i = 1, 2, \dots, \ell$.

3. Adatgráfok

Az adatok felett egy $G = (V, E, W)$ irányítatlan gráfot definiálunk: az adatpontok alkotják a V halmazt, az E élek pedig az adatokat kötik össze. Az adatokhoz egy hasonlósági mértéket választunk: ez szabja meg, hogy van-e él két pont között, illetve ez adja meg a kötés *erősségét*. A kötés erőssége, vagy az él súlya $(W(\mathbf{x}, \mathbf{z}))$ a két adat hasonlóságát,

közelségét fejezi ki. A hasonlósági mértéktől megköveteljük, hogy pozitív és szimmetrikus legyen: $W(\mathbf{x}, \mathbf{z}) \geq 0$, $W(\mathbf{x}, \mathbf{z}) = W(\mathbf{z}, \mathbf{x})$, $\forall \mathbf{x}, \mathbf{z} \in V$.

Két pont hasonlóságát jelölhetjük a pontok vagy, ha sorba rendezzük a pontokat, akkor a pontok indexeinek segítségével:

$$W(\mathbf{x}_i, \mathbf{x}_j) =: w_{ij}.$$

A $\mathbf{W}$ mátrix a súlyozott szomszédsági mátrixot jelöli, melyet hasonlósági mátrixnak vagy egyszerűen *súlymátrixnak* is nevezünk,

$$\mathbf{W} = (w_{ij})_{i,j=1,\dots,N}.$$

Egy pont fokszámát a szomszédos élek súlyainak összege adja meg, és az $\mathbf{x}_i$ pont fokszámát d_i -vel jelöljük:

$$d_i = \sum_{j=1}^N w_{ij}.$$

A *fokszámmátrix* egy olyan diagonálmátrix, melynek főátlóján a pontok fokszámai találhatók:

$$\mathbf{D} = \text{diag}(\mathbf{W}\mathbf{1}).$$

3.1. Gráfok szerkesztése

Az adatok felett értelmezett gráf megszerkesztésének módja legalább annyira fontos, mint maga a tanuló algoritmus, amit alkalmazni fogunk. A gráf felépítéséhez szükségünk lesz egy hasonlósági mértékre (vagy távolságfüggvényre), illetve egy *vágási* mechanizmusra, amely megmondja, hogy milyen esetekben kössünk, és milyen esetekben ne kössünk össze élel két pontot. A legtöbbször használt hasonlósági függvény a Gauss-féle hasonlóság, amely a következő módon értelmezett:

$$k_G(\mathbf{x}, \mathbf{z}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{z}\|^2}{2\sigma^2}\right),$$

ahol a σ paraméter egyfajta *szomszédsági határt* szab meg: minél nagyobb ez az érték, annál nagyobb hasonlóságot eredményez a függvény távolabbi pontok esetén. Látható, hogy a Gauss-féle hasonlósági függvény pozitív és szimmetrikus, illetve még *normalizált* is, azaz minden

hasonlósági érték a $[0, 1]$ intervallumba esik. Minél nagyobb a függvény által visszatérített érték, az annál erősebb hasonlóságot jelent.

A vágási mechanizmus szempontjából a következő típusú gráfokat különböztetjük meg:

- *k*-legközelebbi szomszéd gráfok: Akkor kötjük össze az $\mathbf{x}_i$ és $\mathbf{x}_j$ pontokat, hogyha $\mathbf{x}_j$ az $\mathbf{x}_i$ *k* legközelebbi szomszédai között van. Ez viszont irányított gráfot eredményez, vagyis a súlymátrix nem lesz szimmetrikus. Ennek szimmetrizálására a következő két módszer közül választhatunk:
 - (i) az $\mathbf{x}_i$ és $\mathbf{x}_j$ pontot akkor kötjük össze, ha $\mathbf{x}_j$ $\mathbf{x}_i$ *k* legközelebbi szomszédja között van, *vagy* fordítva ($\mathbf{x}_i$ $\mathbf{x}_j$ *k* legközelebbi szomszédja között van);
 - (ii) az $\mathbf{x}_i$ és $\mathbf{x}_j$ pontot akkor kötjük össze, ha $\mathbf{x}_j$ $\mathbf{x}_i$ *k* legközelebbi szomszédja között van, *és* fordítva.

E két gráf közül az elsőt egyszerűen *k*-legközelebbi szomszéd gráfnak nevezzük, míg a másodikra a *kölcsönös k*-legközelebbi szomszéd gráf elnevezéssel hivatkozunk.

- ε -szomszédsági gráfok: Egy ε -szomszédsági gráfban akkor kötünk össze két pontot, hogyha azok távolsága kisebb egy előre meghatározott ε küszöbértéknél, vagy hasonlóan, ha azok hasonlósága nagyobb egy előre meghatározott ε küszöbértéknél.²
- teljes gráfok: Ez a legegyszerűbb eset, amikor is nem használunk semmilyen vágást, minden, a hasonlósági mérték által meghatározott súlyt meghagyunk.

A gráfok szerkesztésének módszere nagyban befolyásolja a tanuló algoritmus kimenetét. El kell tehát döntenünk, hogy a fent említettek közül melyik gráfot használjuk: valamelyik *k*-legközelebbi szomszéd gráfot, ε -szomszédsági vagy a teljes gráfot? Ugyanakkor azt is el kell döntetünk, hogy súlyozzuk-e az éleket vagy sem, súlyozás esetén pedig meg kell választanunk a hasonlósági mértéket, illetve annak paramétereit. Ezek tárgyalására itt nem térünk ki, a paraméterek megválasztásának szempontjaihoz ajánljuk a [9] munkát.

²Ebben az esetben a vágások után sokszor elhagyjuk a mérték által meghatározott hasonlóságokat, azaz a súlyozatlan gráfot tekintjük.

4. A Laplace-mátrix

A Laplace-mátrix – amint azt az elkövetkezendő részekben látni fogjuk – fontos szerepet tölt be a gráf alapú tanuló algoritmusokban. Tulajdonképpen három algoritmust fogunk a későbbiekben részletesen bemutatni, és mindhárom algoritmusban fel fog bukkanni ez a mátrix. Ez természetesen nem jelenti azt, hogy minden gráf alapú módszer a Laplace-mátrixot használja, de sok ezen alapszik, még akkor is, hogyha ez első pillantásra nem látható.

A Laplace-mátrix definíció szerint:

$$\mathbf{L} = \mathbf{D} - \mathbf{W},$$

ahol $\mathbf{W}$ és $\mathbf{D}$ a már ismert súly-, illetve fokszámmátrix. A Laplace-mátrix fontosabb tulajdonságai:

1. Minden $\mathbf{f} \in \mathbb{R}^N$ vektorra:

$$\mathbf{f}'\mathbf{L}\mathbf{f} = \frac{1}{2} \sum_{i,j=1}^N w_{ij}(f_i - f_j)^2.$$

2. $\mathbf{L}$ szimmetrikus és pozitív szemidefinit.
3. $\mathbf{L}$ legkisebb sajátértéke 0, és a hozzá tartozó sajátvektor az $\mathbf{1} = [1, 1, \dots, 1]'$.
4. $\mathbf{L}$ N darab nemnegatív valós sajátértékkel rendelkezik, $0 = \lambda_1 \leq \dots \leq \lambda_N$.

A mátrix egy másik érdekes és fontos tulajdonsága, hogy pontosan annyi zérus sajátértéke van, ahány összefüggő komponensből áll a gráf.³ Ha a gráf több összefüggő komponensből áll, a Laplace-mátrix felírható blokkdiagonális alakban, ahol minden blokk az illető komponens Laplace-mátrixa lesz.

A Laplace-mátrixnak léteznek más változatai is, nevezetesen a véletlen bolyongás típusú (*random walk*) és a szimmetrikus normalizált (*symmetric*) Laplace-mátrix:

$$\begin{aligned} \mathbf{L}_{\text{rw}} &= \mathbf{D}^{-1}\mathbf{L} = \mathbf{I} - \mathbf{D}^{-1}\mathbf{W}, \\ \mathbf{L}_{\text{sym}} &= \mathbf{D}^{-1/2}\mathbf{L}\mathbf{D}^{-1/2} = \mathbf{I} - \mathbf{D}^{-1/2}\mathbf{W}\mathbf{D}^{-1/2}. \end{aligned}$$

³A tulajdonságok bizonyításához lásd a [9] munkát.

E változatok is a Laplace-mátrixéhoz hasonló tulajdonságokkal rendelkeznek, éspedig:

1. Minden $\mathbf{f} \in \mathbb{R}^N$ vektorra:

$$\mathbf{f}' \mathbf{L}_{\text{sym}} \mathbf{f} = \frac{1}{2} \sum_{i,j=1}^N w_{ij} \left(\frac{f_i}{\sqrt{d_i}} - \frac{f_j}{\sqrt{d_j}} \right)^2.$$

2. $\lambda \mathbf{L}_{\text{rw}}$ sajátértéke $\mathbf{u}$ sajátvektorral akkor és csak akkor, ha $\lambda \mathbf{L}_{\text{sym}}$ sajátértéke $\mathbf{w} = \mathbf{D}^{1/2} \mathbf{u}$ sajátvektorral.
3. $\lambda \mathbf{L}_{\text{rw}}$ sajátértéke $\mathbf{u}$ sajátvektorral akkor és csak akkor, ha λ és $\mathbf{u}$ az $\mathbf{L}\mathbf{u} = \lambda \mathbf{D}\mathbf{u}$ általánosított sajátfeladat megoldása.
4. 0 az $\mathbf{L}_{\text{rw}}$ sajátértéke az $\mathbf{1}$ sajátvektorral, és 0 az $\mathbf{L}_{\text{sym}}$ sajátértéke $\mathbf{D}^{1/2} \mathbf{1}$ sajátvektorral.
5. $\mathbf{L}_{\text{rw}}$ és $\mathbf{L}_{\text{sym}}$ pozitív szemidefinit mátrixok, melyek d nemnegatív valós sajátértékkel rendelkeznek, $0 = \lambda_1 \leq \dots \leq \lambda_N$.

Megjegyezzük, hogy $\mathbf{L}_{\text{rw}}$ és $\mathbf{L}_{\text{sym}}$ ugyanannyi zérus sajátértékkel rendelkezik, mint $\mathbf{L}$.

5. Klaszterezés

5.1. Minimális vágatok és spektrális klaszterezés

A klaszterezés felügyelet nélküli tanulást jelent, azaz nincsenek tanulási példáink, melyek megmondják, adott bemenetre mi legyen a kimenet. A klaszterezés adott pontok egymástól minél jobban elkülöníthető, *homogén* csoportokba való szervezését jelenti, melyen belül az adatok *jobban hasonlítanak* egymáshoz – ezeket a csoportokat nevezzük klasztereknek. Ha adott egy összefüggő irányítatlan súlyozott gráf, akkor a legkézenfekvőbb ötlet megkeresni azokat a részeket, melyek a *leglazább* kapcsolatban állnak a gráf más részeivel. Bináris esetben ez a gráf két részre való bontását/vágását jelenti: ha A és $\bar{A}$ jelöli a V halmaz egy ilyen diszjunkt felbontását, akkor ez megfogalmazható az A és $\bar{A}$ halmazok közötti hasonlóságok összegének minimalizálásával:

$$\operatorname{argmin}_A \sum_{\mathbf{x} \in A, \mathbf{z} \in \bar{A}} W(\mathbf{x}, \mathbf{z}). \quad (1)$$

Ha az A és $\bar{A}$ halmazokat, vagyis a V felbontását az $\mathbf{y} \in \{-1, +1\}^N$ vektorral jelöljük, ahol a -1 címke az egyik, a $+1$ pedig a másik klaszterbe való tartozást jelenti, akkor ezt felhasználva (1) minimalizálandó kifejezése felírható a következőképpen:⁴

$$\begin{aligned} \frac{(\mathbf{y} + 1)'}{2} \mathbf{W} \frac{(-\mathbf{y} + 1)}{2} &= -\frac{1}{4} \mathbf{y}' \mathbf{W} \mathbf{y} + \frac{1}{4} \mathbf{1}' \mathbf{W} \mathbf{1} \\ &= \frac{1}{4} (\mathbf{y}' \mathbf{D} \mathbf{y} - \mathbf{y}' \mathbf{W} \mathbf{y}) = \frac{1}{4} \mathbf{y}' (\mathbf{D} - \mathbf{W}) \mathbf{y} \\ &= \frac{1}{4} \mathbf{y}' \mathbf{L} \mathbf{y}. \end{aligned}$$

Látható, hogy a feladat felírható diszkrét optimalizálási feladatként, amely polinomiális időben megoldható az Edmonds–Karp algoritmussal [4], viszont ez a megoldás az esetek többségében az egyik halmazban nagyon kevés pontot fog tartalmazni – akár egyetlen pontot, hogyha például létezik olyan csúcs a gráfban, mely csak egyetlen csúccsal van összekötve egy kisebb súlyú éllel –, azaz a halmazok mérete nem lesz kiegyenlített. Emiatt behozzuk azt a megkötést, hogy a halmazok mérete legyen egyenlő, azaz teljesüljön az $\mathbf{y}' \mathbf{1} = 1$ feltétel. Ezáltal viszont a feladat NP-nehéz feladattá válik [5], amit relaxációval oldunk meg: nem követeljük meg a diszkrét, $\{-1, 1\}$ feletti megoldást, hanem áthelyezzük a feladatot a valós térbe, majd a végén zérusnál *küszöböljük* a kapott értékeket, így alakítva vissza az eredményt diszkrét megoldássá. A feladat így a következő folytonos optimalizálási feladattá válik:

$$\underset{\mathbf{y} \in \mathbb{R}^N}{\operatorname{argmin}} \quad \mathbf{y}' \mathbf{L} \mathbf{y} \tag{2}$$

$$\text{ú.h. } \mathbf{y}' \mathbf{1} = 0 \text{ és } \mathbf{y}' \mathbf{y} = 1,$$

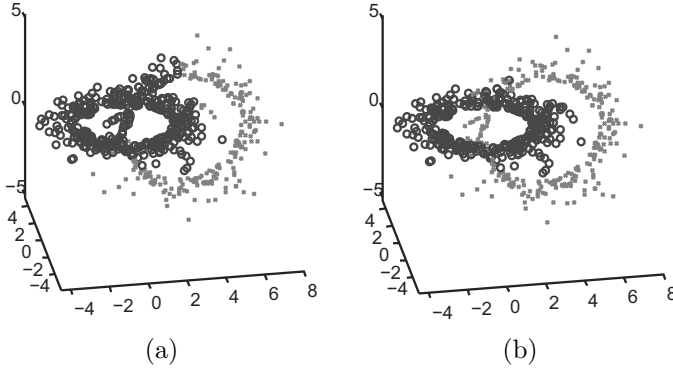
ahol a második megkötés a $\mathbf{0}$ megoldás elkerülése miatt jelent meg. Ez átírható az

$$\underset{\mathbf{y} \in \mathbb{R}^N}{\operatorname{argmin}} \quad \frac{\mathbf{y}' \mathbf{L} \mathbf{y}}{\mathbf{y}' \mathbf{y}} \tag{3}$$

$$\text{ú.h. } \mathbf{y}' \mathbf{1} = 0$$

alakra, melynek megoldása az $\mathbf{L}$ második legkisebb sajátértékéhez

⁴A levezetésben felhasználtuk az alábbi azonosságokat: $\mathbf{1}' \mathbf{W} \mathbf{1} = \mathbf{1}' \mathbf{D} \mathbf{1} = \mathbf{y}' \mathbf{D} \mathbf{y}$.



1. ábra. Spektrális klaszterezés szemléltetése egy kis adathalmazon. Teljes gráfot használtunk Gauss-féle hasonlósági mértékkel és (a) $1/(2\sigma^2) = 0,5$ (88, 17%-os helyes hozzárendelés), illetve (b) $1/(2\sigma^2) = 2$ paraméterrel (100%-os helyes hozzárendelés).

tartozó sajátvektor lesz⁵ [9]. A kapott valós vektort 0-nál küszöböljük: zérusnál kisebb érték az egyik, nálánál nagyobb pedig a másik klaszterbe való tartozást fogja jelenteni.

A minimális vágat helyett sokszor *normalizált* minimális vágatot [7] használunk, amely előnyösebb tulajdonságokkal rendelkezik:

$$\operatorname{argmin}_A \frac{\sum_{\mathbf{x} \in A, \mathbf{z} \in \bar{A}} W(\mathbf{x}, \mathbf{z})}{\sum_{\mathbf{x} \in A, \mathbf{v} \in V} W(\mathbf{x}, \mathbf{v})} + \frac{\sum_{\mathbf{x} \in A, \mathbf{z} \in \bar{A}} W(\mathbf{x}, \mathbf{z})}{\sum_{\mathbf{z} \in \bar{A}, \mathbf{v} \in V} W(\mathbf{z}, \mathbf{v})}.$$

Az optimalizálási feladatot ebben az esetben is – az előbbihez hasonló módon – folytonos térbe helyezzük és ott oldjuk meg, mivel a diszkrét optimalizálási feladat ez esetben NP-teljes [7]. Ebben az esetben a megoldás az $\mathbf{L}_{\text{sym}}$ második legkisebb sajátértékéhez tartozó sajátvektor lesz, pontosabban $\mathbf{D}^{-1/2} \mathbf{v}_2$, ahol $\mathbf{v}_2$ az illető sajátvektort jelöli – ezt

⁵Ha az $\frac{\mathbf{y}' \mathbf{L} \mathbf{y}}{\mathbf{y}' \mathbf{y}}$ kifejezést (Rayleigh-együttható) akarjuk minimalizálni, akkor ennek optimumpontja az $\mathbf{L}$ legkisebb sajátértékéhez tartozó sajátvektorban lesz [6]. Az $\mathbf{y}' \mathbf{1} = 0$ megkötés viszont ekkor nem teljesül, mivel láttuk, hogy $\mathbf{L}$ legkisebb sajátvektora az $\mathbf{1}$ vektor 0 sajátértékkel. Tekintsük a Rayleigh-együttható következő tulajdonságát [6, 7]: ha $\mathbf{M}$ valós szimmetrikus mátrix és ha megköveteljük, hogy $\mathbf{y}$ ortogonális legyen a $j - 1$ legkisebb $\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_{j-1}$ sajátvektorra, akkor $\frac{\mathbf{y}' \mathbf{M} \mathbf{y}}{\mathbf{y}' \mathbf{y}}$ a következő legkisebb $\mathbf{u}_j$ sajátvektorban veszi fel minimumát.

fogjuk majd zérusnál küszöbölni [9].

Mivel a megoldást a Laplace-mátrix egyik sajátvektora szolgáltatja, ezért ezt a típusú klaszterezést *spektrális klaszterezésnek* nevezzük. A spektrális klaszterezésnek természetesen létezik több klaszterre kiterjesztett változata is – ezen változatokról például a [9] munkában olvashatunk. Itt nem térünk ki részletekbe menően ezek tárgyalására, mindössze annyit jegyzünk meg, hogy a többklaszteres esetben szintén a Laplace-mátrix sajátvektorai szolgáltatják a megoldást, ekkor viszont több sajátvektor is a megoldás része lesz. A sajátvektorokat a pontok egy kisebb dimenziós térbe való leképezésére használjuk, majd ebben a térben a k -közép klaszterező algoritmust használjuk a *célklaszterek* meghatározásához.

Az 1. ábrán a spektrális klaszterezés kimenetét láthatjuk egy kis adathalmazon. Az adathalmaz összesen 600 pontot tartalmaz, melyből 322 pont az egyik, 278 pedig a másik klaszterben található. A két klaszter a két láncszerűen összekapcsolt pontfelhőt jelenti; a pontok hovatartozását (azaz az algoritmus kimenetét) kék körökkel és piros x-ekkel jelöltük. Az adatgráf mindkét esetben teljes, a súlyokat a Gauss-féle hasonlósági függvénnyel adtuk meg különböző paraméterrel, ezek eredményei láthatók az (a) és (b) rajzokon.

6. Osztályozás

6.1. Laplace típusú regularizált legkisebb négyzetek módszere

Az osztályozás felügyelt tanulást jelent, vagyis a rendszer (adat, címke) tanulási példákon keresztül tanulja meg, hogy adott bemenetre (adat) mi legyen a kimenet (címke). A klaszterezéssel ellentétben – ahol sokszor a klaszterek számát sem ismerjük, ennek meghatározása is a feladat része – a csoportok száma véges. Ezeket a csoportokat osztályoknak nevezzük.

Egy elterjedt osztályozási, illetve *regressziós* módszer⁶ a legkisebb négyzetek módszere [2]. A legkisebb négyzetek módszere – bináris osztályozási esetben – a pontokat úgy próbálja meg szétválasztani

⁶Felügyelt tanuláskor lehetnek valós címkéink is, ekkor regresszióról beszélünk. A legkisebb négyzetek módszere valójában egy regressziós metódus, viszont osztályozásra is könnyedén alkalmazható.

egy hipersíkkal⁷, hogy az a legkisebb négyzetes hibát eredményezze az ismert címkékre:

$$\arg\min_{\mathbf{w}} \frac{1}{\ell} \sum_{i=1}^{\ell} (\mathbf{w}'\mathbf{x}_i - y_i)^2 = \frac{1}{\ell} \|\mathbf{X}'\mathbf{w} - \mathbf{y}\|^2. \quad (4)$$

Ehhez az objektív függvényhez általában hozzátoldunk egy *regularizációs* tagot⁸ is, mert az $\mathbf{X}\mathbf{X}'$ mátrixtól nem tudjuk megkövetelni, hogy mindig invertálható legyen. Így a (4) függvényéhez hozzáadva a $\lambda\|\mathbf{w}\|^2$ tagot, majd ezt $\mathbf{w}$ szerint deriválva és egyenlővé téve zéróval kapjuk, hogy

$$\mathbf{w} = (\mathbf{X}\mathbf{X}' + \lambda\ell\mathbf{I})^{-1} \mathbf{X}\mathbf{y}.$$

A döntési függvényünk, vagyis a pontokhoz címkét rendelő függvényünk ez esetben

$$f(\mathbf{x}) = \text{sgn}(\mathbf{w}'\mathbf{x})$$

lesz. Ez egy *induktív* tanuló rendszer, azaz a döntési függvény *általánosan* alkalmazható bármilyen pontra. Ezzel szemben az ez után bemutatásra kerülő, címkepropagálás nevet viselő algoritmus egy másfajta, ún. *transzduktív* tanulási módszert ír le.

A gráf alapú vagy Laplace típusú legkisebb négyzetek módszerében egy *jobban* szétválasztó hipersík elérése érdekében címkézetlen pontokat is bevonunk az optimalizálási feladatba. Az így kapott módszert félig-felügyelt tanuló módszernek nevezzük, mert címkézett és címkézetlen pontokat egyaránt felhasznál. A félig-felügyelt tanuló rendszerek egyik alapfeltevése az úgynevezett *simasági* feltevés (*smoothness assumption*): ha két pont közel áll egymáshoz, azaz hasonlóságuk nagy, az osztályozó kimenete nagy valószínűséggel ugyanaz lesz a két pontra [3]. Ezt a következőképpen vihetjük be a feladatba. Tekintsünk először egy hasonlósági mértéket. Az ismert címkék függvényében felírt négyzetes hibához hasonlóan most az osztályozó kimenetei közötti négyzetes hibát vesszük minden pontpárra, majd ezt a hasonlósággal skálázzuk – ebből adódik hibafüggvényünk második része:

$$\arg\min_{\mathbf{w}} \frac{1}{\ell} \|\mathbf{X}'_{\ell}\mathbf{w} - \mathbf{y}\|^2 + \lambda\mathbf{w}'\mathbf{w} + \frac{\mu}{2N^2} \sum_{i,j=1}^N a_{ij} (\mathbf{w}'\mathbf{x}_i - \mathbf{w}'\mathbf{x}_j)^2,$$

⁷ A hipersík egy n -dimenziós tér $(n-1)$ -dimenziós altere, két dimenzióban például egy egyenes, három dimenzióban egy sík.

⁸ A regularizáció valamilyen többletinformáció, követelmény bevezetését jelenti egy adott problémába, a feladat megoldhatóvá tételének érdekében.

ahol N a címkézett és a címkézetlen pontok együttes számát jelöli, $N = \ell + u$. Ezt a Laplace típusú regularizált legkisebb négyzetek módszerének nevezzük [1]. Az egyszerűbb és kompakt jelölés érdekében osszuk fel a teljes adatmátrixot két részre, a címkézett és címkézetlen pontok vektoraira, melyeket jelöljünk rendre $\mathbf{X}_\ell$, illetve $\mathbf{X}_u$ -val. A teljes adatmátrix tehát ezek konkatenációjából áll elő, $\mathbf{X} = [\mathbf{X}_\ell \ \mathbf{X}_u]$. Ha az új, utolsó tagban – az egyszerűbb jelölés érdekében – elvégezzük az $\mathbf{f}_i := \mathbf{w}'\mathbf{x}_i$ és $\mathbf{f} := \mathbf{X}'\mathbf{w}$ helyettesítéseket, akkor a következőket vehetjük észre:

$$\begin{aligned}
 \sum_{i,j=1}^N a_{ij} (\mathbf{f}_i - \mathbf{f}_j)^2 &= \sum_{i,j=1}^N a_{ij} (\mathbf{f}_i^2 - 2\mathbf{f}_i\mathbf{f}_j + \mathbf{f}_j^2) \\
 &= 2 \sum_{i,j=1}^N a_{ij}\mathbf{f}_i^2 - 2 \sum_{i,j=1}^N a_{i,j}\mathbf{f}_i\mathbf{f}_j \\
 &= 2 \sum_{i,j=1}^N \mathbf{f}_i^2 d_i - 2\mathbf{f}'\mathbf{A}\mathbf{f} \\
 &= 2\mathbf{f}'\mathbf{D}\mathbf{f} - 2\mathbf{f}'\mathbf{A}\mathbf{f} = 2\mathbf{f}'\mathbf{L}\mathbf{f},
 \end{aligned} \tag{5}$$

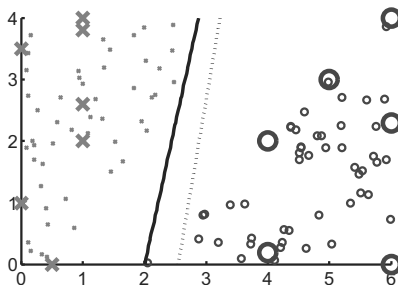
ahol újra megjelent a hasonlósági gráf Laplace-mátrixa. Visszahelyettesítve $\mathbf{f}$ -et, minimalizálandó függvényünk a következőképpen alakul:

$$\operatorname{argmin}_{\mathbf{w}} \frac{1}{\ell} \|\mathbf{X}'_\ell \mathbf{w} - \mathbf{y}\|^2 + \lambda \mathbf{w}'\mathbf{w} + \frac{\mu}{N^2} \mathbf{w}'\mathbf{X}\mathbf{L}\mathbf{X}'\mathbf{w}.$$

Innen – az előbbi összefüggést $\mathbf{w}$ szerint deriválva majd egyenlő téve zérussal – kapjuk, hogy

$$\mathbf{w} = \left(\mathbf{X}_\ell \mathbf{X}'_\ell + \lambda \ell \mathbf{I} + \frac{\mu \ell}{N^2} \mathbf{X}\mathbf{L}\mathbf{X}' \right)^{-1} \mathbf{X}_\ell \mathbf{y}.$$

A legkisebb négyzetek módszere, amint az a fentiekben látható volt, egy szétválasztó hipersíkot keres az adatokhoz úgy, hogy a négyzetes hiba minimális legyen. A Laplace típusú regularizált legkisebb négyzetek módszere pedig ezt az alapötletet terjeszti ki úgy, hogy a szétválasztó hipersíkot a pontok közötti hasonlóságok is befolyásolják. Ha a hipersíkot csak a normálvektorral definiáljuk, akkor mindig egy az origón átmenő hipersíkot kapunk. Viszont jelen esetben nem csak ilyen hipersíkok



2. ábra. A regularizált legkisebb négyzetek módszerének szemléltetése egy kis adathalmazon. A szaggatott, illetve a folytonos vonal a kapott szétválasztó hipersíkot jelöli a regularizált legkisebb négyzetek módszerével, illetve annak Laplace típusú kiterjesztésével. A megcímkézés szempontjából a rajz a félig-felügyelt eset kimenetét mutatja. Ebben az esetben hasonlósággént skalárszorzatot használtunk szimmetrikus normalizált Laplace-mátrixszal és $\mu = 200$ paraméterrel. A λ együttható értékét mindkét esetben 0,001-re állítottuk.

jöhetnek számításba, ezért az általános egyenlet minden paraméterét meg kell határoznunk, vagyis döntési függvényünk $\mathbf{w}'\mathbf{x} + b$ alakú. Hogy ne bonyolítsuk el az optimalizálási feladatot egy új b paraméter bevezetésével, az adatainkat terjesszük ki egy új *konstans* dimenzióval:

$\begin{bmatrix} \mathbf{X} \\ \mathbf{1}' \end{bmatrix}$, így az objektív függvényen nem kell változtatnunk.

A 2. ábrán a regularizált legkisebb négyzetek módszerének és annak Laplace-típusú kiterjesztésének kimenetét láthatjuk egy kis adathalmazon. A tanuló halmaz összesen 100 pontot tartalmaz, melyből 13-at (7 pozitív, 6 negatív példa) tartalmaz a címkézett és 97-et (49 pozitív, 48 negatív példa) a címkézetlen halmaz. Habár mindkét hipersíkot jelöltük az ábrán, a rajz a Laplace típusú regularizált legkisebb négyzetek módszerének (100%-osan pontos) kimenetét mutatja: a piros x-ek a pozitív, a kék körök a negatív pontokat jelölik, ahol a nagyobb méretű jelek a címkézett pontokat jelentik.

6.2. Címkepropagálás

A félig-felügyelt tanulás egy tipikus példája a címkepropagálás [12]. Az adatokon a már látott módon egy gráfot építünk, majd a címkeket a

tanulási adatoktól a címkézetlen adatok felé *propagáljuk* a kapcsolatokat erősségétől függően.

A címkék propagálásának megvalósítása érdekében egy átmenet-valószínűség mátrixot építünk a hasonlóságok segítségével. Ha a hasonlósági mátrixot a $\mathbf{W}$ szimbólummal jelöljük, az átmenet-valószínűség mátrixot pedig $\mathbf{P} = (p_{ij})_{i,j=1,\dots,N}$ -vel, akkor a valószínűségeket a következő módon számítjuk ki:

$$p_{ij} = \frac{w_{ij}}{\sum_{k=1}^N w_{ik}}.$$

Ez röviden a $\mathbf{P} = \mathbf{D}^{-1}\mathbf{W}$ összefüggéssel is felírható, ahol $\mathbf{D}$ a már ismert fokszámmátrix.

Az algoritmust most is csak bináris osztályozásra adjuk meg, viszont a feladat nagyon egyszerűen átírható többosztályos esetre [11, 12]. Jelölje a címkék vektorát $\mathbf{y} \in \{-1, 1\}^N$, és bontsuk ezt fel két részre: jelölje a felső ℓ elem az ismert címkéket, az alsó rész pedig a címkézetlen adatokét:

$$\mathbf{y} =: \begin{bmatrix} \mathbf{y}_\ell \\ \mathbf{y}_u \end{bmatrix}.$$

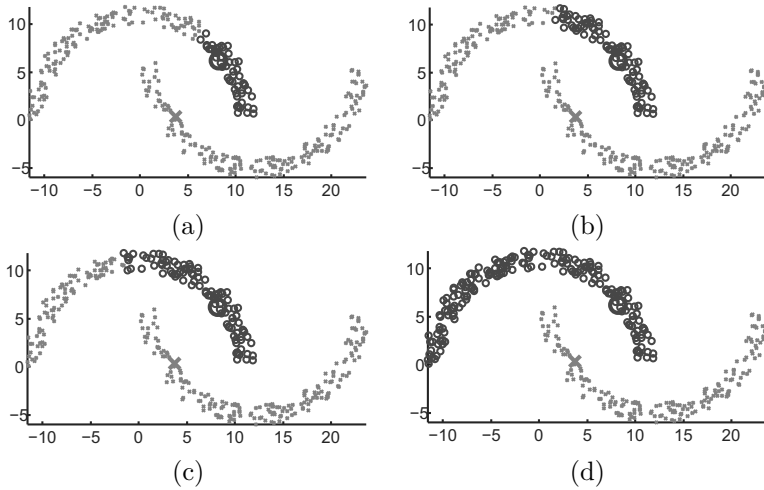
Célunk a címkézetlen adatok $\mathbf{y}_u$ címkéinek meghatározása. A módszer alapötlete: az i -edik pont címkéje legyen egyenlő az illető pont *bemenő* szomszédainak az átmenet-valószínűségek szerint súlyozott címkéjével. Azaz, minden bemenő szomszédja propagálja a címkéjét az i -edik pontnak az átmenet-valószínűség szerint. Természetesen, kezdetben a címkézetlen pontoknak nincs címkéjük, ellenben ezek is lehetnek szomszédai az i -edik címkézetlen pontnak. A címkézetlen pontoknak választhatunk tetszőleges címkét – akár mindegyiknek 1-et vagy -1 -et –, a későbbiekben látni fogjuk, hogy ez nem befolyásolja a végső eredményt – az iterációk során az eredményvektor egy stabil konfigurációhoz konvergál. Tehát legyen

$$y_i = p_{1i}y_1 + p_{2i}y_2 + \dots + p_{Ni}y_N, \quad i = 1, \dots, N.$$

Ezt a címkepropagálást mátrix alakban a következőképpen írhatjuk fel az összes pontra:

$$\mathbf{y} = \mathbf{P}'\mathbf{y}. \quad (6)$$

Az algoritmus a következő lépésekből áll:



3. ábra. A címkepropagálás iteratív változatának szemléltetése egy kis adathalmazon. Az adatgráf ebben az esetben is teljes, a hasonlóságokat a Gauss-féle hasonlósági függvénnyel adtuk meg, $1/(2\sigma^2) = 0,2$ paraméterrel. A négy rajz a címkepropagálás kimenetét mutatja az (a) 50-edik, (b) 100-adik, (c) 200-adik és (d) 300-adik iterációban.

1. $\mathbf{y} = \mathbf{P}'\mathbf{y}$
2. Helyettesítsük vissza az *eredeti*, ismert címkéket $\mathbf{y}_\ell$ -be.
3. Vissza az 1. lépésre.

A fenti lépéseket addig kell ismételnünk, amíg az $\mathbf{y}_u$ vektor konvergálni fog egy stabil megoldáshoz. A konvergencia ellenőrzését például úgy végezhetjük el, hogy megnézzük, mennyit változott az $\mathbf{y}_u$ vektor az előző lépésben kapott vektorhoz képest⁹, és amint ez egy előre meghatározott kis érték alá esik, megállunk.

Könnyen megmutatható, hogy az algoritmus kimenete nem függ a kezdeti $\mathbf{y}_u$ címkek megválasztásától. Ha a címkepropagálást megvalósító (6) rekurzív kifejezést a következőképpen írjuk fel,

$$\begin{bmatrix} \mathbf{y}_\ell \\ \mathbf{y}_u \end{bmatrix} = \begin{bmatrix} \mathbf{T}_{\ell\ell} & \mathbf{T}_{\ell u} \\ \mathbf{T}_{u\ell} & \mathbf{T}_{uu} \end{bmatrix} \begin{bmatrix} \mathbf{y}_\ell \\ \mathbf{y}_u \end{bmatrix},$$

⁹A változást mérhetjük a vektorok közötti euklideszi távolsággal.

ahol $\mathbf{T}$ a $\mathbf{P}$ mátrix transzponáltját jelöli, akkor innen kifejezhető az $\mathbf{y}_u$,

$$\begin{aligned}\mathbf{y}_u &= (\mathbf{I} - \mathbf{T}_{uu})^{-1} \mathbf{T}_{u\ell} \mathbf{y}_\ell \\ &= (\mathbf{I} - \mathbf{W}_{uu} \mathbf{D}_u^{-1})^{-1} \mathbf{W}_{u\ell} \mathbf{D}_\ell^{-1} \mathbf{y}_\ell,\end{aligned}\quad (7)$$

ahol a $\mathbf{W}$ mátrixot a fenti $\mathbf{T}$ -hez hasonlóan bontottuk fel, a $\mathbf{D}$ diagonálmátrixot pedig a $\begin{bmatrix} \mathbf{D}_\ell & \mathbf{0} \\ \mathbf{0} & \mathbf{D}_u \end{bmatrix}$ módon. Ha a Laplace-mátrixokat is felbontjuk hasonlóképpen, akkor az előbbi kifejezés felírható ezek függvényében is:

$$\begin{aligned}\mathbf{y}_u &= -\mathbf{D}_u \mathbf{L}_{uu}^{-1} (\mathbf{L}_{rw})'_{\ell u} \mathbf{y}_\ell \\ &= -\mathbf{D}_u (\mathbf{L}_{rw})_{uu}^{-1} \mathbf{D}_u^{-1} (\mathbf{L}_{rw})'_{\ell u} \mathbf{y}_\ell.\end{aligned}$$

Ez tulajdonképpen azt jelenti, hogy a címkepropagálás megvalósítható iteratíván a bemutatott háromlépéses algoritmussal, de kiszámíthatjuk a címkéket a (7) összefüggés segítségével is. Mivel (7) mátrixinverziót is tartalmaz, amely köbös bonyolultságú, nagy adathalmazok esetén hatékonyabb lehet az iteratív változat használata.¹⁰

A 3. ábrán a címkepropagálás iteratív változatának működését szemléltettük egy kis adathalmazon. Az adathalmaz összesen 385 pontot tartalmaz, melyből mindössze kettő címkézett, a maradék 383 pont címkéje ismeretlen. A címkézetlen pontok két különálló felhője 191, illetve 192 pontot tartalmaz. A négy rajzon az algoritmus kimenete látható az iterációs szám függvényében. A piros x-ek a pozitív, a kék körök a negatív pontokat jelölik, ahol a nagyobb méretű jelek a címkézett pontok.

A címkepropagálás – mint azt már korábban említettük – egy transzduktív tanuló algoritmus. Az ilyen típusú algoritmusok, ellentétben az induktív módszerekkel, nem határoznak meg egy tetszőleges pontra alkalmazható általános függvényt, hanem csak a függvény értékeit adják meg a kérdéses pontokban [3, 8]. A címkepropagálásban tehát egy pont címkéje csak akkor határozható meg, hogyha azt hozzáadjuk a címkézetlen pontok halmazához, és újra kiszámítjuk az összes címkét. A következőkben röviden bemutatjuk a címkepropagálás egy másik változatát, amely jobb tulajdonságokkal rendelkezik. A különbség a már

¹⁰ A címkék csak akkor lesznek meghatározhatók, illetve az algoritmus csak akkor fog konvergálni, hogyha az $\mathbf{I} - \mathbf{T}_{uu}$ mátrix invertálható. Megjegyezzük, hogy a Gauss-féle hasonlóság használata esetén ez mindig teljesül.

bemutatott módszer és e között mindössze az, hogy a propagálást most az $\mathbf{y} = \mathbf{P}\mathbf{y}$ egyenlettel írjuk le. Ezt azt jelenti, hogy egy pont címkéjét a pont *kimenő* szomszédai határozzák meg,

$$y_i = p_{i1}y_1 + p_{i2}y_2 + \dots + p_{iN}y_N, \quad i = 1, \dots, N.$$

Ezzel az egyszerű változtatással azt érjük el, hogy a keresett címkéket megadó explicit kifejezésünk a következőképpen módosul:

$$\mathbf{y}_u = (\mathbf{I} - \mathbf{P}_{uu})^{-1} \mathbf{P}_{ul} \mathbf{y}_\ell = -\mathbf{L}_{uu}^{-1} \mathbf{L}_{ul} \mathbf{y}_\ell. \quad (8)$$

Ebben az esetben megfigyelhetjük, hogy az optimalizálási problémát felírhatjuk a következő alakban:

$$\underset{y_i, i=\ell+1, \dots, N}{\operatorname{argmin}} \quad \frac{1}{2} \sum_{i,j=1}^N a_{ij} (y_i - y_j)^2, \quad (9)$$

ahol a_{ij} újfent az i és j -edik pont hasonlóságát jelöli. Az (5) alapján az objektív függvényt felírhatjuk az $\mathbf{y}' \mathbf{L} \mathbf{y}$ alakban, ahonnan a Laplace-mátrix felbontásával az

$$\mathbf{y}'_u \mathbf{L}_{uu} \mathbf{y}_u + 2\mathbf{y}'_u \mathbf{L}_{ul} \mathbf{y}_\ell + \mathbf{y}'_\ell \mathbf{L}_{\ell\ell} \mathbf{y}_\ell$$

kifejezéshez jutunk. Ha ennek a deriváltját egyenlővé tesszük zérussal és kifejezzük belőle az $\mathbf{y}_u$ -t, a következőt kapjuk:

$$\mathbf{y}_u = -\mathbf{L}_{uu}^{-1} \mathbf{L}_{ul} \mathbf{y}_\ell,$$

amely megegyezik a (8) egyenlettel. A címkepropagálás ezen új változatával fel tudunk írni egy egyszerű induktív függvényt egy új pont címkéjének meghatározására. Tételizzük fel, hogy bizonyos címkézetlen pontokra már kiszámítottuk a címkéket. Ekkor egy új $\mathbf{x}$ pont a (9) objektív függvényt a következőképpen módosítja:

$$C + \sum_{i=1}^N W(\mathbf{x}, \mathbf{x}_i) (y - y_i)^2,$$

ahol C a (9) objektív függvény értékét jelöli, y pedig az új pont címkéje. Ennek deriváltját egyenlővé téve zérussal y -ra az

$$y = \frac{\sum_{i=1}^N W(\mathbf{x}, \mathbf{x}_i) y_i}{\sum_{i=1}^N W(\mathbf{x}, \mathbf{x}_i)}$$

egyenletet kapjuk, amely alkalmazható tetszőleges $\mathbf{x}$ pont címkéjének kiszámítására.

7. Összefoglalás

A tanulmányban bemutattuk a gráf alapú tanulás néhány módszerét, és láthattuk, hogy habár ezek egymástól eltérő, illetve különböző feladatokat megoldó algoritmusok, mindegyikben megjelenik a Laplace-mátrix. Ezért ezt a speciális mátrixot sokszor a gráf alapú tanuló módszerek egyik központi fogalmaként definiálják. Bemutatásra került egy klaszterező algoritmus, egy regressziós módszer, illetve egy transzduktív tanuló algoritmus. Mindhárom módszernél csak a bináris esetet tárgyaltuk, de az algoritmusok viszonylag egyszerűen kiterjeszthetők több klaszterre, illetve osztályra. A cél nem a módszerek részletekbe menő elemzése és vizsgálata volt, hanem inkább egy bevezető nyújtása a gráf alapú gépi tanulási módszerekhez. Ezen módszerek további tanulmányozásához a [9], [11] és [3] munkákat ajánljuk.

Hivatkozások

- [1] M. Belkin, P. Niyogi, V. Sindhwani, Manifold regularization: A geometric framework for learning from labeled and unlabeled examples, *Journal of Machine Learning Research*, 7 (2006) pp. 2399–2434.
- [2] C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006.
- [3] O. Chapelle, B. Schölkopf, A. Zien, *Semi-Supervised Learning*, MIT Press, 2006.
- [4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, The MIT Press, 3rd edition, 2009.
- [5] M. R. Garey, D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Co., 1979.
- [6] G. H. Golub, C. F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, 3rd edition, 1996.
- [7] J. Shi, J. Malik, Normalized cuts and image segmentation, *IEEE Conf. Computer Vision and Pattern Recognition*, June 1997.

- [8] V. N. Vapnik, *Statistical Learning Theory*, Wiley, 1998.
- [9] U. von Luxburg, A tutorial on spectral clustering, *Statistics and Computing*, 17(4) (2007) pp. 395–416.
- [10] D. Zhou, B. Schölkopf, T. Hofmann, Semi-supervised learning on directed graphs, *NIPS*, MIT Press, 2005, pp. 1633–1640.
- [11] X. Zhu, *Semi-supervised learning with graphs*, PhD thesis, 2005.
- [12] X. Zhu, Z. Ghahramani, Learning from labeled and unlabeled data with label propagation, Technical Report CMU-CALD-02-107, Carnegie Mellon University, 2002.

Erlang folyamatok és a köztük lévő kapcsolatok felderítése

Bozó István, Tóth Melinda

Eötvös Loránd Tudományegyetem, Informatikai Kar

{bozo_i, toth_m}@inf.elte.hu

1. Bevezetés

Az Erlang [3, 4] egy dinamikusan típusos funkcionális programozási nyelv, amely masszívan párhuzamos, illetve amelyet elosztott alkalmazások fejlesztésére terveztek.

A RefactorErl [1] nevű eszköz egy statikus elemző és transzformáló keretrendszer, melynek egyik célja Erlang programok megértésének támogatása. Ehhez hozzátartozik az elosztott és párhuzamos programok elemzése is. Azonban ezek elemzése jóval nehezebb, mint a szekvenciális programok elemzése, főként egy olyan dinamikus nyelv esetén, mint az Erlang. Sok esetben statikus módszertannal a dinamikus információ töredéke számítható ki (becsülhető meg) fordítási időben; ám az is igaz, hogy ez még mindig több lehet és gyorsabban számítható, mint egy programozó által kézzel összeszedhető információ.

A célunk az volt, hogy a statikus elemzés lehetőségeinek megfelelően adjunk egy olyan folyamat elemzést, amely segítségével az Erlang programozási nyelven írt párhuzamos és elosztott programokból kinyerhető azok kommunikációs modellje. Ez a modell nagyban elősegíti a programozók számára a kód megértést, hiszen ennek segítségével képet alkothatnak arról, milyen folyamatok vannak egy rendszerben, és ezek hogyan kommunikálnak.

A RefactorErl keretrendszert bővítettük egy újabb elemzéssel, amelyvel előállítható egy könnyen átlátható statikus kommunikációs gráf.

2. RefactorErl

A RefactorErl [2] egy statikus elemző és transzformáló keretrendszer. Ezen keretrendszer Erlang programok elemzését és transzformálását (refaktorálását) teszi lehetővé.

A keretrendszer egy szemantikus programgráfban (SPG [5]) reprezentálja a programokat, amely három rétegből épül fel:

- Lexikális
- Szintaktikus
- Szemantikus

A gráf egy $SPG = (N, A_N, A_V, A, T, E)$ hatossal adható meg, ahol

- N – a gráf csúcsainak a halmaza
- A_N – attribútum nevek halmaza
- A_V – lehetséges attribútum értékek halmaza
- $A : N \times A_N \rightarrow A_V$ – csúcs címkéző parciális függvény
- T – él címkék halmaza
- $E : N \times T \times N_0 \rightarrow N$ – élcímkéző parciális függvény rendezéssel

Az elemzések során az ebben a gráfban tárolt információt használjuk majd fel, illetve a gráfhoz új éleket és csúcsokat is adunk meg majd.

A lexikális elemző a forráskódból előállítja a tokeneket, amelyekből a szintaktikai elemző segítségével egy szintaxisfa épül.

A szemantikus elemzők különféle információkkal bővítik ezt a szintaxisfát:

- változók kötési és hivatkozási helye
- függvények definíciója és ezek alkalmazásai
- függvényhívási információ
- rekord és rekordmező információk
- stb.

A szemantikus elemzők nagy része inkrementális, azaz ha változás történik a gráfban, akkor ezeket lokálisan kezeli és csak a szükséges gráfrészletekben állítja helyre az információkat. A `RefactorErl` további, nem inkrementális elemzéseket is tartalmaz, mint például a dinamikus függvényhívás elemző. Ezek olyan információkat használnak és állítanak elő, amelyeket minden módosításnál újra elő kell állítani a teljes gráfra.

Az általunk megvalósított elemzés segítségével a keretrendszerbe betöltött programok statikus kommunikációs modelljét lehet előállítani, melyhez a szintaktikus és szemantikus elemzők által előállított információkat használjuk fel [6, 7]. Az elemzés nem inkrementális, azaz ha változás történik a gráfban, az elemzést újra végre kell hajtani.

3. Használt fogalmak

Ebben a fejezetben az elemzés leírásánál használt fogalmakat tárgyaljuk részletesen, hogy az Erlangban kevésbé jártas olvasó is könnyedén átláthassa az algoritmus részleteit.

3.1. Folyamatok indítása

Erlang nyelven írt programok esetén az `erlang` modul `spawn/1,2,3,4`, `spawn_link/1,2,3,4` és `spawn_monitor/1,3` függvényeinek segítségével indíthatunk újabb folyamatokat. A különböző változatok segítségével megadható, melyik Erlang node-on induljon a folyamat, illetve melyik függvényt szeretnénk a folyamatban végrehajtani. Ezeket a paraméterek akár futási időben is megadhatóak, így kellően nagy szabadságot ad, hogy dinamikusan kezelhetőek legyenek.

Ez a szabadság megnehezíti a statikus elemzést, hiszen csak a forráskódban fellelhető információkkal tudunk dolgozni.

3.2. Folyamatok azonosítója

A folyamatok indításának visszatérési eredménye egy egyedi folyamat azonosító (`pid`). A `pid` segítségével a processzek globálisan címezhetőek, azaz ha több Erlang node van összekötve, akkor is garantált ezek egyedisége.

Egy folyamat a saját azonosítóját a `self()` függvény segítségével határozhatja meg.

3.3. Folyamatok regisztrálása

A folyamatok nevesítése a `register/2` függvény segítségével lehetséges. A regisztrált név csak egy Erlang node-on belül érvényes. A folyamat a nevének felhasználásával címezhető, anélkül hogy a folyamat azonosítója ismert lenne.

A folyamatok nevének globális regisztrálásához a `global:register_name/2,3` függvények használhatók. Ebben az esetben a folyamat tetszőleges Erlang node-ról címezhető ezzel a névvel.

3.4. Kommunikációs primitívek

Két folyamat kommunikációjára az alábbi primitívek használhatóak:

- `Pid ! Msg` – A `!` operátor segítségével üzenetet (`Msg`) küldhetünk a `Pid` azonosítóval rendelkező folyamatnak. A `Pid` helyett használható a folyamat neve is, amennyiben az regisztrálva lett.
- `erlang:send*/2,3` – Hasonlóan a `!` operátorhoz, ezekkel a függvényekkel üzenetet küldhetünk egy másik folyamatnak.
- `receive` – A `receive` konstrukcióval egy üzenetet fogadhatunk az üzenetsorról. A konstrukció lehetőséget ad szelektív üzenet fogadásra (megfelelő minta segítségével) és mindaddig blokkolódik, amíg nem sikerül a mintáknak megfelelő üzenetet fogadnia.

3.5. *ets* táblák

Az *ets* az Erlang beépített adattárolója, amely nagy mennyiségű adat tárolására ad lehetőséget és az adatok elérése konstans idejű. Az *ets* táblák létrehozásával egy újabb folyamat indul, amely az adat tárolásáért felel. A tábla mindaddig elérhető, amíg azt közvetlenül nem töröljük, vagy a szülő folyamat be nem fejeződik. A táblák indításakor különböző opciók segítségével befolyásolható annak típusa, láthatósági köre, neve, stb.

4. Az elemzés algoritmus

Az elemző algoritmus a RefactorErl keretrendszer részeként lett megvalósítva. Az elemzés a szemantikus programgráf bejárásával kiterjeszti a gráfot, majd az így létrejött gráfból előállítja a program statikus kommunikációs modelljét.

Az elemző folyamat négy főbb lépésre osztható fel, melyet az alábbiakban részletesen ismertetünk.

4.1. Folyamatok felderítése

Az első lépésben az algoritmus lokalizálja azon pontokat a gráfban, ahol új folyamatok kerülnek elindításra. Ezután a következő lépésekben meghatározza a lehető legtöbb információt a folyamatról, amely statikus elemzések segítségével elérhető. Ez a lépés a szemantikus programgráfot bővíti újabb szemantikus `pid` típusú csúcsokkal és gráf élekkel.

4.1.1. Folyamatok meghatározása

Az elemzés kezdeti lépéseként meg kell határozni, hogy mely függvények indulhatnak különálló folyamatként. Az elemzés során a legpontosabb eredményre törekszünk, amely statikus elemzéssel előállítható. Ezért, ha szükséges, akkor adatfolyam elemzési eredményeket is felhasználunk. A folyamatokhoz rendelt csúcspontok és a folyamatokat indító kifejezések közötti kapcsolatot a `spawn_def` címkéjű él adja.

4.1.2. Regisztrált folyamatok

A nyelv, illetve a virtuális gép lehetőséget biztosít, hogy a folyamatokat globális névvel lássuk el. A folyamat ezek után elérhető mind az azonosítója, mind pedig a regisztrált neve segítségével is. A regisztrálással lehetőség nyílik, hogy a folyamat az azonosítójának hiányában is elérhető csupán a nevének ismeretével.

Ahhoz, hogy minél pontosabb legyen az elemzés, szükséges meghatározni, hogy mely folyamatok kerültek regisztrálásra és milyen névvel regisztrálták őket. Az elemzéshez itt is felhasználjuk az adatfolyam elemzést, mert a regisztrált név érkezhetsz paraméterként, így az explicit módon nem feltétlen jelenik meg a regisztrálás végző kifejezésben. Ha a

név csak futási időben kerül megadásra, azaz ha statikusan ez nem határozható meg a forráskódból, akkor ezen információ hiányában csökken az elemzés pontossága.

A beazonosított kifejezések és a folyamatok a `reg_def` címkéjű éllel kerülnek összekötésre. A regisztráláshoz használt nevek pedig a folyamatot reprezentáló csúcs egyik argumentuma lesz.

4.1.3. Kommunikáció felderítése

Következő lépésben a felderített folyamatok közötti lehetséges kommunikációt határozzuk meg. Első lépésként beazonosítjuk a küldést és fogadást végző kifejezéseket a folyamatok által végrehajtott függvényekben, majd adatfolyam elemzéssel meghatározzuk, hogy mely folyamatok között történt az üzenetváltás. A küldő, illetve a fogadó kifejezések közötti kommunikációt a `flow` címkéjű él jelöli.

4.2. Újabb folyamatok felderítése

A második lépésben további folyamatokkal bővítjük a meglévő folyamatokat. Ez a lépés a szemantikus gráfot bővíti újabb csúcsokkal és élekkel.

4.2.1. Kifejezések felderítése

A szemantikus gráfban megkeressük azon üzenetet küldő, üzenetet fogadó kifejezéseket, amelyek egyetlen folyamat végrehajtási útjában sem szerepelnek. Ezek azon függvényekben találhatóak, amelyek nem külön folyamatként indulnak, vagy statikus elemzéssel nem határozható meg hogy folyamatként indulhatnak. Minden egyes függvényhez, amely ilyen kifejezést tartalmaz, egy újabb csúcsot adunk meg.

4.2.2. Kifejezések folyamatkörnyezete

Miután a fennmaradó kifejezésekhez is hozzárendeltük a megfelelő folyamatokat, újabb élekkel bővítjük a gráfot. Az `eval_in` címkéjű élék azt határozzák meg, hogy az adott kifejezés melyik folyamatban kerül kiértékelésre. Természetesen csak az elemzés szempontjából releváns kifejezések (folyamat indítás, üzenet küldés, regisztrálás, stb.) és a folyamat azonosítók közé kerülnek behúzásra ezek az élék.

4.3. Gráf előállítása

Az elemzés harmadik lépése egy különálló gráfot állít elő, amely magába foglalja a folyamatokat és a köztük lévő kapcsolatokat.

A gráf csúcsai:

- A szemantikus gráfban is megjelenő folyamatokat reprezentáló csúcsok.
- A superprocessz (SP) csúcs, amely a környezet/virtuális gép szerepét tölti be.

A superprocessz csúcsnak kitüntetett szerepe van, ez alá kerül bekötésre minden olyan folyamat, amelynek nincs szülő folyamata.

A gráf élei:

- `spawn_link` – Egy folyamatot elindító és az ez általa elindított folyamat között jelenik meg, amely a processzek közötti szülő-gyermek kapcsolatot írja le.
- `register` – A regisztrálást végrehajtó folyamat és a regisztrált folyamat között jelenik meg.
- `spawn_sp` – A szülő nélküli folyamatok és a virtuális gépet reprezentáló superprocessz között kerül behúzásra.
- `{send, Message}` – Az üzenetet küldő és az üzenetet fogadó folyamat között jelenik meg. A `Message` a küldött üzenetet jelenti, amennyiben ez statikusan kinyerhető a forráskódból.

Ez így előállított gráf képezi a folyamatok kommunikációs modell-jének alapját, amely még további információkkal bővíthet az elemzés következő fázisában.

4.4. Rejtett függőségek/kommunikáció

Az elemzés negyedik lépése újabb információkkal bővíti a szemantikus gráfot, valamint a különálló gráfot, amelyet az előző (4.3) fejezetben mutattunk be.

Az elemzés ezen fázisa elemzi az `ets` modul által definiált táblák létrehozását, írását és olvasását. Ahogyan azt a fogalmak áttekintésében leírtuk, az `ets` egy tábla szerkezet, amely e különálló folyamatként

létezik. A táblát a futó folyamatok írni és olvasni is tudják, ha az megfelelő opciókkal lett létrehozva. Ezáltal egy új kommunikációs csatorna nyílik a folyamatok számára.

4.4.1. Táblák létrehozása

ETS táblákat az `ets:new/2` függvény alkalmazásával lehet létrehozni. Elemezve ezen alkalmazásokat, meg tudjuk határozni, hogy mely pontokon keletkeznek ilyen táblák. Minden létrehozott táblának egy új `ets_tab` típusú szemantikus csúcsot hozunk létre mindkét gráfban.

A szemantikus gráfban a következő új élek jelennek meg:

- `ets_tab` – a gyökércsúcsból az `ets_tab` típusú csúcsba vezető élék.
- `ets_def` – az `ets_tab` típusú szemantikus csúcsból kiinduló él, amely a definiálási helyét határozza meg.
- `ets_ref` – az `ets_tab` típusú szemantikus csúcsból kiinduló él, amely a hivatkozási helyeit határozza meg.

A kommunikációs gráf egy új éllel bővül. A táblát létrehozó folyamat és a táblát reprezentáló csúcsok között a `create` címkéjű él kerül behúzásra.

4.5. Tábla olvasások

Az ETS táblákat több különböző függvény segítségével lehet olvasni. A szemantikus gráfot bejárva meghatározhatjuk, hogy ezek a függvények hol vannak alkalmazva. Ha egy folyamatban végrehajtódik valamelyik olvasást szolgáló függvény, az azt jelenti, hogy kommunikáció történik a táblán keresztül. Adatfolyam elemzéssel megpróbáljuk kideríteni, hogy melyik táblából történik az olvasás. Ha ezt sikerül meghatározni, akkor ezt a szemantikus gráfban az utasítást végrehajtó folyamatot és a táblát reprezentáló csúcsok között `read` címkéjű csúcs jelöli. A kommunikációs gráfot szintén bővítjük egy `{read, Data}` címkéjű éllel, ahol a `Data` az olvasott elem kulcsát, vagy a kereséshez használt mintát tartalmazza.

4.6. Tábla írások

Az ETS táblák írásához több függvény is adott. A szemantikus gráfból lekérdezhetőek a függvények alkalmazásainak helye, illetve hogy melyik folyamatban kerül végrehajtásra a függvény. Ezen információk ismeretében az adatfolyam elemzés segítségével meghatározzuk, hogy melyik táblában történt az írás. Amennyiben ez meghatározható statikusan, akkor a szemantikus gráfot egy új `write` címkéjű éllel bővítjük, amely a folyamat és az tábla között kerül behúzásra.

5. Kommunikációs gráf előállítása és megjelenítése

5.1. Elemzés végrehajtása

A `RefactorErl` shelljéből a `refanal_proc:anal_proc()` függvény kiértékelésével állíthatjuk elő a kommunikációs gráfot. Az elemzés lefuttatása által kibővül a szemantikus programgráf az új élekkel és szemantikus csúcsokkal, illetve létrejön a kommunikációs gráf.

A kommunikációs gráf csúcsait és a köztük futó éleket egy `processes` nevű `ets` táblában tároljuk. A gráf ebben a formájában is megtekinthető a virtuális gép tábla megjelenítőjével (`tv:start()` parancs), de ez nagy gráfok esetén nehezen áttekinthető.

5.2. Gráf megjelenítése

A `dot` gráf leíró nyelv segítségével közvetlenül emberek számára is könnyen olvashatóvá tehető a kommunikációs gráf. Ezért lehetővé tettük a gráf `dot` formátumba való exportálását. Ez a formátum átalakítható grafikus formátumra is, erre több program is lehetőséget biztosít.

Az elemzés végrehajtása után a `refanal_proc:create_dot()` függvény segítségével előállíthatjuk a `dot` formátumú fájlt. A függvény kiértékelése létrehoz az eszköz `data` könyvtárában egy `processes.dot` fájlt, amely tartalmazza a teljes kommunikációs gráfot.

Unix alapú rendszereken a

```
dot -Tpdf processes.dot -o processes.pdf
```

utasítással egy `pdf` formátumú dokumentummá alakítható a `dot` fájl. Ter-

mészetesen nem csak *pdf* állítható elő, hanem különböző formátumokba (*svg*, *ps*, *gif*, *png*, stb) konvertálható.

A forráskód a https://plc.inf.elte.hu/erlang/repos/branches/process_com helyről érhető el.

5.3. Példa

Az alábbiakban tekintsük az alábbi két kliens-szerver Erlang modult (1. és 2. ábrák) és a belőlük készített kommunikációs gráfot (3. ábra). A szerver modul elindítja a `job_server`-t, mely kliensek csatlakozására vár, majd a tőlük érkező kéréseket fogadja és dolgozza fel. Érdekesség, hogy a kliensekkel való kommunikáció, az eredmények visszaadása, egy `ets` táblán keresztül történik. A kliens folyamatok csatlakoznak a szerverhez és a beolvasott értékek alapján a szerver felé kéréseket intéznek.

A 3. ábra mutatja, melyek azok a kapcsolatok, amelyeket statikusan fel tudott ismerni a bemutatott algoritmus.

6. Összefoglalás

Programok megértésének támogatása a programozók mindennapjait nagyban megkönnyíti. Különösen igaz ez olyan szoftverekre, ahol az egyszerűbb nyelvi elemek mellett párhuzamosságot, konkurenciát, elosztottságot támogató elemek is jelen vannak.

A RefactorErl elemző eszköz egyik célja, hogy a kód megértést támogassa. Ebben a cikkben egy olyan kiterjesztését mutattuk be az eszköznek, mely segítségével statikusan elemzhető az Erlang folyamatok kommunikációja.

```

-module(server).

-export([start/0, stop/0]). %% Server interface
-export([init/0, loop/1]). %% Server callbacks

-define(Name, job_server).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
start() -> register(?Name, spawn_link(?MODULE, init, [])).

stop() -> ?Name ! stop.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
init()->
    process_flag(trap_exit, true),
    ?MODULE:loop([]).

loop(State)->
    receive
        stop -> ok;
        {connect, Cli} -> ?MODULE:loop([Cli|State]);
        {disconnect, Cli} ->
            ?MODULE:loop(lists:filter(fun(A) -> A /= Cli
                                     end, State));
        {do, Mod, Fun, Tab} ->
            handle_job(Mod, Fun, Tab),
            ?MODULE:loop(State)
    end.

handle_job(Mod, Fun, Tab) ->
    Data = ets:select(Tab, [{{'$1', '$2'}, [{'/=', '$1', result}],
                           ['$$', '']}]),
    Result = Mod:Fun(Data),
    ets:insert(Tab, {result, Result}).

```

1. ábra. Szerver modul

```
-module(client).
-export([start/1, input/1]).

-define(Name, job_server).

start(Client)->
    spawn_link(?MODULE, start_cl, [Client]).

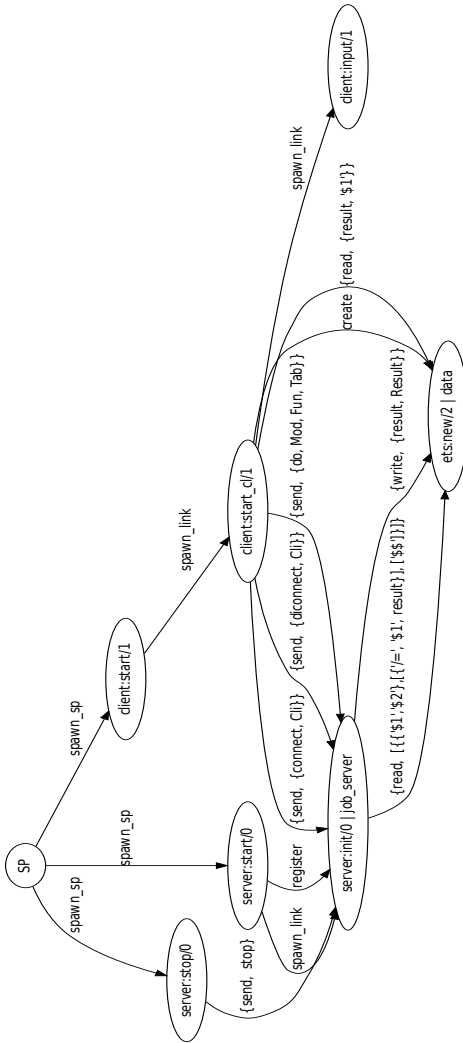
start(Client) ->
    ?Name ! {connect, Client},
    ets:new(data, [named_table, public]),
    spawn(?MODULE, input, [self()]),
    loop(data, Client).

loop(Tab, Name) ->
    receive
        quit ->
            ?Name ! {disconnect, Name},
            io:format("~p~n", [ets:match(Tab, {result, '$1'})]);
        {job, {Mod, Fun}} ->
            ?Name ! {do, Mod, Fun, Tab},
            loop(Tab, Name)
    end.

input(Loop) ->
    case read_input() of
        quit ->
            Loop ! quit,
            ok;
        Job ->
            Loop ! {job, Job},
            input(Loop)
    end.

read_input() ->
    [ets:insert(data, Data) || Data <- init_data()],
    returns_the_job_to_be_executed().
```

2. ábra. Kliens modul



3. ábra. Folyamat kommunikációs gráf

Hivatkozások

- [1] RefactorErl Home Page, 2011.,
<http://plc.inf.elte.hu/erlang/>
- [2] I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Kőszegi, M. Tejfel, M. Tóth, Refactorerl – source code analysis and refactoring in RefactorErl – Source Code Analysis and Refactoring in Erlang, *Proceeding of the 12th Symposium on Programming Languages and Software Tools, Tallin, Estonia* 2011.
- [3] F. Cesarini, S. Thompson, *Erlang Programming*, O'Reilly Media, 2009.
- [4] Ericsson AB, *Erlang Reference Manual*,
http://www.erlang.org/doc/reference_manual/part_frame.html
- [5] Z. Horváth, L. Lövei, T. Kozsik, R. Kitlei, M. Tóth, I. Bozó, R. Király, Modeling semantic knowledge in Erlang for refactoring, *International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009, Selected papers, Presa Universitara Clujeana*, pp. 38–53.
- [6] M. Tóth, I. Bozó, Static analysis of complex software systems implemented in Erlang, *Central European Functional Programming School, LNCS 7241* (2012), Springer, pp. 440–498.
- [7] M. Tóth, I. Bozó, Z. Horváth, M. Tejfel, First order flow analysis for Erlang, *Proceedings of the 8th Joint Conference on Mathematics and Computer Science, MaCS 2010*.

Egy startup anatómiája

Czigola Gábor*

Eötvös József Collegium**

`czigola.gabor@vanmit.hu`

Vannak ötleteink, és fel kell ismernünk, hogy megvalósításuk akadályai csak saját magunk vagyunk. Egy ötlet megvalósulása gazdasági tevékenység, így közgazdasági szempontból, mind pénzügyi, mind üzleti folyamatként elemezhető. Vizsgálódásunkban egy startup közösség munkáját kísérhetjük nyomon.

Vanmit.hu néven céget majd weboldalt hoztunk létre. Célunk egy online közösségi piactér létrehozása volt. Hasonló oldalak már léteznek, folyamatosan fejlődnek, jelentős forgalommal bírnak, bizonyítva az ötlet működőképességét (vatera.hu, aprod.hu, ebay.com, ...). További motivációt jelentett, hogy az online kereskedelem világszinten, de különösképpen Magyarországon is fejletlennek számít, folyamatos az innovációs tevékenység, a parétóhatékonyság messze nem teljesen kiaknázott. Évi 10-20%-os növekedés sem ritka, sőt, mivel az online kereskedelemnek komparatív előnye van a hagyományos boltokhoz képest, recessziós időszakokban nemhogy nem visszaesés, sokkal inkább növekedés tapasztalható.

A kereskedelem alapja, egy tranzakció előfeltétele a bizalom. Kereskedelmi oldalak attól közösségiek, hogy a tagok nyilvános profillal rendelkeznek, korábbi tranzakcióik megtekinthetőek, a felhasználók bizalmi hálót alkotnak. Az oldalnak lényegében nincs szerepe a tranzakcióban (*disintermediate*), az a felek között közvetlenül zajlik (*peer-to-peer*).

A **vanmit.hu** azzal próbálta tovább növelni a kereslet és kínálat egymásra találásának hatékonyságát, hogy egy térképen, térben,

*EPAM Systems, Budapest

**2004–2010

közigazgatási határoktól függetlenül jelenítette meg és tette kereshetővé az ajánlatokat, megmutatva hol kapható meg azonnal a keresett termék. Hasonló oldalak a kezdéskor még nem léteztek, de közben megjelentek ([airbnb.com](https://www.airbnb.com), [shpock.com](https://www.shpock.com)).

1. Egy jó ötlet

Gyakran vannak ötleteink. Társasági beszélgetéseket hallgatva megfigyelhető, hogy milyen gyakran áll elő valaki egy ötlettel, hogy bezzeg így-vagy-úgy kellene ezt-vagy-azt csinálni. Ami érdekes ugyanakkor, hogy milyen kevesen vállalják fel saját ötletüket az okoskodáson túl, próbálják legalább alapjaiban megvalósítani, kitéve ezzel a valóság próbájának.

Megint másoknál azt láttam, hogy valóban a megvalósítás szándékával hoznak fel ötleteket, de tucatszámra. Folyamatosan újabb és újabb lehetőségekről, irányokról beszélnek, melyek között alig van kapcsolat vagy akár ellentétesek is. Például láttam olyan jegyzeteket, melynek címlapjára „Ötleteim” volt írva, valójában „Ötleteim, melyeket sose fogok megvalósítani” kellett volna hogy legyen.

Tudnunk kell, hogy egy ötlet megvalósítása elköteleződést jelent. Ha nem is egy életre, de tipikusan legalább fél, de inkább két-három év aktív, rész vagy fő elfoglaltságot jelent, még sikertelen projekt esetében is. Amit szintén vegyünk még számításba, hogy amit birtoklunk, az birtokba is vesz minket. Tényleg erre a termékre van szüksége a világnak? Tényleg nem másra tennénk fel elkövetkező éveinket? Tényleg ki fogunk állni érte? Tényleg szeretnénk, hogy sikeres legyen?

Dolgozzunk ki egy üzleti tervet: Foglaljuk össze egy mondatban az ötletet. Mi az a probléma amit megold? Hogyan nyújt az ötlet megvalósítva megoldást? Pénzügyileg megalapozott az ötlet? Milyen alternatív megoldások léteznek, használnak jelenleg? Ki a célközönségünk?¹ Hogyan fognak rólunk tudomást szerezni? Kik a versenytársaink? Miért használnának inkább minket? Kik vennének részt a megvalósításban? Mi van eddig az asztalon?

¹A „mindenki” nem egy célközönség. Pontosan határozzuk meg az elsődleges piaci szegmensünket olyan jellemzők alapján, mint életkor, lakhely, közösség, hobbi, munka. Ha a termékünk mindenkinek szól, ez egy intő jel, hogy alapvető hiányosság van a tervezésben, vagy nem értjük igazán saját ötletünket.

2. Egy megvalósítható ötlet

Egy valóban jó ötlet esetén a kifogások keresése nem megengedhető. Ha problémát, akadályt keresünk, biztosan meg is fogjuk találni (még talán ott is, ahol valójában nincs is.) Felnőtt emberként saját magunk vagyunk lehetőségein korlátja, nem mutogathatunk másra, se a társadalomra, se a rendszerre, se a jogi környezetre. Azt is érdemes figyelembe venni, hogy az életben semmi sem jár alanyi jogon, még akkor sem, ha az élettől kapott lapjaink nem azonosak másokéval.

Egy ötlet megvalósíthatósága úgy vizsgálható, hogy pénzügyi, üzleti tervet készítünk (*business plan*). Egyik oldalon soroljuk fel kiadásainkat, másikon a bevételeket.² Először a fejlesztési időszakra:

Kiadások (SUM 1.880.000 Ft)	Bevételek (SUM: 1.680.000 Ft)
Szerverbérlet: 12 x 10.000 Ft = 120.000 Ft Ügyvéd és cégalapítás: 200.000 Ft Könyvelő: 12 x 5.000 Ft = 60.000 Ft Tervezés és fejlesztés: 12 x 100.000 Ft = 1.200.000 Ft Marketing: 300.000 Ft	Rendelkezésre álló alaptőke: 200.000 Ft Szerver szolgáltatás megosztása: 12 x 5.000 Ft = 60.000 Ft Tagok havi hozzájárulása: 12 x 3 x 150 € = 5.400 € = 1.620.000 Ft

Ugyanerre a táblázatra szükség van az ezt követő éles időszakra is, egy olyan távlatban, ami a megtérülést megalapozza. A kiadási oldal viszonylag jól tervezhető, skálázható, nem úgy a bevételek. Spekulációra nem lehet alapozni. A megtérülés (*ROI – return of investment*) mégis approximálható megfelelő metrikákkal³:

²A vanmit.hu legnagyobb tévedése volt, hogy a kiadások nagy része fejlesztésre, cégalapításra ment el. Alig maradt marketingre. Későbbi kutatásokból azt következtetem, hogy legalább kétszer, de inkább öt-tízszer annyit ajánlott költeni reklámra, marketingre, pr-ra, mint minden másra összesen. Nálunk ez az arány pont fordított volt. Magától jövő viralításra nem lehet alapozni, ugyanakkor virális elemeket (*customer acquisition vehicle*) beépíteni erősen ajánlott.

³A vanmit.hu esetében arra alapoztunk, hogy a magyar e-kereskedelem mintegy 200 milliárd forint, évi 10-20%-os növekedéssel, és ha ennek az 1%-t sikerül megszerezni, és a konkurenciánál kisebb, 5%-os részt kérünk az eladoktól (azoktól tehát, akik pénzt csinálnak rajtunk keresztül), az évi 50 millió forintos bevételt eredményez, ami bőven fedezte volna a működési költségeket. Ehhez százszáz nagyságrendben lett volna szükség profi, és pár ezer alkalmi eladóra. Marketing hiányában már látjuk, hogy ezek elérése nem igazán lehetséges.

- Hogyan szerezzük meg az első tíz ügyfelet?
- Milyen folyamat fog további ügyfeleket hozni?
- Mekkora a teljes potenciális ügyfélkör?
- Milyen bevételi formáink vannak?
- Mi a tervünk a bevételek beszedésére?
- Mekkora az egy ügyfélre eső bevétel összesen? (*TLV – total lifetime value*)
- Mennyibe kerül nekünk egy-egy ügyfél megszerzése?

Ezután helyezzük el ezeket egy időskálán, megvizsgálva, hogy nemcsak éves szinten, de negyedéves, havi lebontásban is rendelkezésre fog-e állni a mindenkor szükséges forgótőke. Ezután válasszuk szét a tételeket két csoportba, fix és változó költségekre. Például a könyvelő fix költség, viszont a marketing változó, hisz szabadon skálázható. Ez a felosztás a fenntarthatóság, skálázhatóság vizsgálatánál lesz fontos. (Mik garantált/garantálandó tételek, mik azok, amelyek forgalom/igény függvényében változtathatóak?)

Mit tehetünk akkor ha nem fedezik a kiadásokat a bevételek? Azt kijelenteném, hogy egy olyan ötlet, ami gazdaságilag nem fenntartható, nem egy jó ötlet. Nem kell profitot termelni, nem kell profitra optimalizálni, elég a fenntarthatóság, tehát a bevételek hosszú távon fedezzék a kiadásokat. Nem várhatjuk el másoktól, hogy a mi ötletünket ok nélkül finanszírozzák.

Alapesetben a csapat maga bocsátja rendelkezésre a szükséges erőforrásokat, és a tulajdon(jogok)ban azonos arányban osztoznak. Ha az ötlet megálmodóinak nem éri meg a szükséges rizikót, akkor kétséges az ötlet jogossága. Különösen informatikai projekteken igaz ez, ahol a belépési költség igen alacsony. Például egy étterem jól bejáratott üzleti modell, mégis viszonylag nagy kezdeti befektetést igényel, ezzel szemben egy sikeres weboldal valóban elindítható egy vidéki garázból és a felhőből.

Banki kölcsönt vagy hitelt akkor érdemes felvenni, ha teljesen biztos a megtérülés. Főleg akkor érdemes ezt az opciót választani, ha csak forgóhitelre van szükségünk, például hónap elején jelentkezik a kiadások zöme, míg a bevétel hónap végén érkezik. Bankok a saját hitelüket

kockáztatják kihelyezéskor, így általában elvárják az önrészt. A pénzügyi alapismeretek megléte feltétlenül szükséges.

Startup közösségekben, konferenciákon is találhatunk további csapattagokat, akikkel együtt már rendelkezésre állna minden szükséges erőforrás. Arra vigyázzunk, hogy ezek rendkívül innovatív, kompetitív környezetek, sokszor üres ígéretekre szervezve, ne csodálkozzunk, a mások méritenek „saját” ötletünkből, és mi se féljünk másoktól tanulni. Ha kritikát kapunk, például nem értik a termékünket, akkor feltétlen vegyük ezt figyelembe. Ha lehetőséget kapunk előadni, a csapat és az ötlet „zszenialitása” helyett inkább arra fókuszáljunk, hogy a) mit értünk el eddig, b) mik a céljaink, c) miért reálisak a céljaink, d) mit akarunk elérni, kapni.

Angyali befektetőknek (angel investors) hívjuk az égből kapott pénzt (*seed money*) adókat. Ez lehet akár egy gazdag nagybácsi, akár egy alacsony kamatkörnyezethez hozzáférő kalandor befektető (*venture capitalist*). Kulturális jellemző, hogy Amerikában gyakran nem igénylik a megtérülést, inkább keresik a legeket (legnagyobb, legtöbb, leggyorsabb stb.) Ingyen ebédre mégse alapozzunk és azt se felejtjük, hogy aki pénzt ad, az egyszer el is fog várni valamit.

Lehetőség még pályázati úton, állami, pontosabban fogalmazva újra-elosztott pénzt szerezni. Ez a legkiszámíthatóbb tőkeforrás ami gyakran vissza nem térítendő támogatás, kamattámogatás vagy hitelgarancia formájában érkezik. Előfordulhat, hogy a kiadásokat először saját zsebből kell fedezni, és csak később kaphatjuk vissza valamilyen formában. Az állami pályázatok, amelyekkel lényegében a saját adóforintjainkra pályázunk, szándékosan piactorzító hatásúak. Előfordulhat, hogy egy megkérdőjelezhetően életképes ötlettel, termék és előkészítés nélküli, de formailag helyes pályázó, aki egy zsákfaluban jegyzi be a céget, a nem kevés dokumentációt elkészítve milliókat kap egy regionális alaptól munkahely- és esélyegyenlőség teremtésére. Közben egy gazdagabb nagyvárosban reális és valós megtérüléssel sem találunk pályázati alapot, vagy ha igen, a papírmunka nagyobb költséggel bírhat, mint az elnyerhető összeg. Jelentősebb pályázati pénzknél sajnos a korrupció is számottevő lehet.

3. Egy fenntartható ötlet

Mielőtt belevágunk az ötlet megvalósításába, vegyünk még pár tényezőt számításba.

Rendelkezésre áll-e egy hatékony és sikeres csapat? Az erőforrásokon túl (beleértve pénzt és időt), a csapat rendelkezik-e a szükséges készségekkel (*skills*), azok aránya megfelelő-e? Van-e eltávolítandó ballaszt a csapatban? Nem ritka, hogy szerencsehuszárok a karnak csatlakozni, őket onnan ismerhetjük fel, hogy véleményük, beleszólásuk általában van, de hozzáadott érték, munka nem látszik, tipikus menedzser alkat. Vagy éppen szállítana valamit, de olyasmit amire igazából nincs is szükség. Nem kell ellenségesnek lenni, hiszen érthető, hogy szívesen csatlakozna egy potenciálisan sikeres projekthez, de aki nem része a csapatmunkának, az a csapatban se legyen benne, mert vissza fogja tartani a munkát és részesedést követelhet később. Ne féljünk nemet mondani, akire később mégis szükség van, az később is szívesen fog csatlakozni.

Először tervezőasztalon készítsük el a terméket, készítsünk megvalósítási tervet (ld. később). Mik a főbb funkciók? Hogyan érhetőek el? Milyen elemekből áll a megvalósítás? Milyen vizuális, formatervek használhatóak? Milyen tervminták? Nem szükséges vízesés-specifikáció, követelmény dokumentum, hanem egy agilis termékleírásra van szükség, ami már meghatározza a fejlesztés irányát.

Megvalósítás előtt is merjünk, sőt, ajánlott előtesztelni. Ez azt jelenti, hogy a megvalósítási tervet, véletlenszerű ismerősöknek (kollégáknak, barátoknak stb.) bemutatjuk, kikérjük szabad véleményüket. Nem ritka, hogy ekkor derül ki, mégiscsak van már ilyen termék, hogy nem értik vagy nem tudnák használni, hogy más megközelítést igényelne, vagy hogy akár teljesen más megoldásra lenne szükségük. Rendkívül sokat nyerhetünk az ilyen korai visszajelzésekből, ne ignoráljuk azokat.

Készítsünk skálázási tervet. Mi történik ha a termékünk sikeres lesz? Milyen hatással lenne a változó költségekre? Például hogyan alakulnának marketing és infrastrukturális kiadások egy, tíz, száz és ezer százalékos éves növekedés esetén? Bírni fogjuk nemcsak az átlagos de a maximális terhelést is? Hasonlóan elemezzük a visszaskálázást is. Ha senkit se érdekel a termék, mit lehet felmondani és lekapcsolni? Hogyan állítanánk le a termék elérhetőségét? Nem kerülünk adósságszpiálba?

Ezen a ponton érdemes meghatározni a megállási kritériumot is.

Befektetést lehet a pókerhez hasonlítani: ameddig nyerő lapjaink vannak, emeljük a tétet, de ha nem, akkor ki kell szállni, azonnal. Határozzunk meg olyan idő és pénzügyi korlátokat, amelyeket ha elérünk, és nem látunk konkrét, kézzelfogható megtérülést, meg kell állni, ki kell szállni. Veszélyes kísértés, hogy folyton még csak egy kicsit hosszabbítunk, még egy kis pénzt beleteszünk. Egy lyukas lábosba hiába öntünk több vizet, sose fog megtelni. Egyértelműsítsük magunkban, hogy a siker nem garantált, mik alapján tudjuk belátni, hogy nem szabad tovább folytatni?

Analóg módon egy kitörési terv sem árt. Mi van siker esetén? Mi a célunk? Eladjuk a jogokat? Tovább növekedünk? Sokan álmodnak gazdagságról egy vállalkozás kezdetekor, de vajon tudnánk-e kezelni azt a helyzetet, ha hirtelen milliárdokat ér? Meg tudnánk őrizni önmagunkat, józan eszünket?

4. A legszűkebb értelemben vett termék (LÉT / MVP)

Amikor ötletünk megvalósításába kezdünk, ill. az ahhoz kapcsolódó tervezési fázis során, törekedjünk a legszűkebb értelemben vett termék (*MVP - minimum viable product*) előállítására. Ez azt jelenti, hogy azt, és csak azt fejlesszük le, csak arra fordítsunk erőforrást, ami az ötletünk apavető magját képezi (*core principles*).

Például a **vanmit.hu** esetében ezek a kereskedelem, a lokalitás és a bizalom volt. Ez azt implikálta, hogy szükségünk van 1) új eladás indítására 2) vásárlás lehetőségére 3) térképen listázni, szűrni az eladásokat 4) felhasználói profilra (ez utóbbi már vitatható, mert az oldal a közvetlen, nem postai értékesítésre szolgál, a bizalom az üzenetküldés és személyes találkozás során is megteremthető) .

Minden egyes funkciót és fejlesztést meg kell kérdőjelezni ebben a fázisban, hogy tényleg szükség van-e rá? Nem az a kérdés, hogy jó ötlet-e, hogy hasznos lenne-e, hogy szeretnék-e, hogy van-e létjogosultsága, hogy zseniális-e, hanem hogy tényleg használhatatlan lenne-e a termék ha kihagyjuk, tényleg helyettesíthetetlen-e, tényleg nélkülözhetetlen-e?

Minden egyes másodperc, amit egy kihagyható, helyettesíthető, nélkülözhető képesség fejlesztésére telik elvesztegetett erőforrás. Hiába hasznos, hiába jó ötlet, hiába szeretnénk. Más, valóban feltétlenül szük-

séges képesség fejlesztésétől veszi el az erőforrást. Ha működni tud a termék nélküle, nincs rá szükség a termék életképességének bizonyításához. Ha már egyszer a termék beindult, úgyis lesz kapacitásunk új képességeket fejleszteni, és akkor sokkal több információnk lesz, hogy a prioritásokat jól határozzuk meg.

5. Megvalósítás

Az eddigiek során meghatároztunk egy megvalósítandó ötletet, ami egy meghatározott célközönség meghatározott problémájára fog megoldást adni. Először üzleti terv készült (szembeállítva a kiadásokat és bevételeket, szükséges befektetést, várható nyereséget), majd ha a megvalósítás mellett döntünk, megvalósítási tervet (*product plan*) készítünk. Ez már technikai jellegű, tartalmazza:

- domén modell (*domain design*)
Definiálja az entitásokat, adattípusokat, a szolgáltatások interfészét valamint az elemi műveleteket. Miközben ezek meghatározásra kerülnek, a fejlesztők és üzleti résztvevők között sok kommunikáció szükséges, ennek pozitív hozadéka egy egységes nyelvezet kialakulása.
- megvalósítási architektúrát (*implementation architecture*)
A domén modellre épülve meghatározandó, hogy milyen fejlesztői eszközöket, platformot, keretrendszereket, programkönyvtárakat és tervezési mintákat használjunk. Mik az integrációs pontok és hogyan garantáljuk a nem-funkcionális követelményeket.
- grafikai terv
Hogyan néz ki a termék? Milyenek az átmenetek? Hogyan mutatja a hibákat? Milyen a szín és formavilága? Milyen ikonokat és képi elemeket használ?⁴
- fejlesztési és szállítási terv és infrastruktúra (*development and release plan and infrastructure*)

⁴Személyes tapasztalatom az, hogy kerülendő a .psd/.pdf formátumban szállítani ezt, mert helyet ad az interpretációnak és a részletek leplezésének. Ajánlott a termék végső formátumában kérni ezeket, weboldal esetén például egy html/css template-ben.

Megválaszoljuk, hogyan fog(nak) a fejlesztői csapat(ok) dolgozni, hogyan lesznek képesek folyamatosan szállítani (*continuous delivery*), hogyan fogunk tesztelni, hogyan fogunk élesbe menni. Egyrészt folyamatokat és infrastruktúrális követelményeket kell megadni, másrészt az integrációs pontok fejlesztői és tesztelői változatát.

- minimum képességek (*minimum features*)
A legszűkebb értelemben vett termék a legszűkebb értelemben vett képességekkel, és csak azokkal kell rendelkezzen. Ezek lényegében funkcionális követelmények csoportosítása. Amennyiben agilis módszerrel fejlesztünk (pl. scrum, kanban) akkor ezek lényegében sztorik vagy taszkok, amivel a fejlesztést majd elindítjuk.

Ezen a ponton fontos fejben tartani, hogy a legtöbb startup csak nagyon kis mértékben innovál, valójában már meglévő lehetőségeket csomagol be, tesz egyszerűbbé, könnyen elérhetővé. Ezt tükrözzé a megvalósítási tervünk is, tehát használjunk nyílt forráskódú és szabad szoftvereket, nyílt, felhő alapú szolgáltatásokat. Licenzekre és infrastruktúrára költeni ebben a fázisban jelzésértékű, rossz értelemben.

Hasonlóan fontos szem előtt tartani, hogy a valódi termék nem a terv, nem a dokumentáció, nem az infrastruktúra és nem a szoftver. *A valódi termék a felhasználói élmény!* Számtalan projekt hasalt el a döntésképtelenség és a végtelen refaktorálások feneketlen vermében, nevük is van e jelenségeknek, íme néhány:

- bearanyozás (*gold plating*)
A fejlesztés során újabb és újabb javaslatokkal állnak elő, hogyan lehetne még jobban csinálni, alapvető tervezési döntéseket felülírva, és egyébként üzleti szempontól jelentéktelen vagy amúgy is jól működő részek refaktorálását megkövetelve. Ha van kézenfekvő megoldás, ne keressünk tovább még tökéletesebbet. Ha van egy működő megoldás, ne akarjuk a „újabbra” és „modernebbre” cserélni. Ha üzleti szempontból nem bír jelentőséggel, nem szabad időt pazarolni rá. Természetesen törekedni kell a legjobb döntésekre, de egy döntés megváltoztatása mindig költséges, megalapozott nyereség nélkül ne vágjunk bele. (*cost/benefit analysis*)

- analízis-paralízis (*analysis paralysis*)

Gyakori, hogy a döntéshozók nem akarnak, nem tudnak vagy nem képesek döntéseket hozni. Ki kell tudni kényszeríteni ezeket. Ha van több megoldás, mindig éljünk a legkézenfekvőbb választásával. Nem ördögtől való akár egy dobókockát segítségül hívni. Egy termék nem attól lesz sikeres mert ilyen vagy olyan adatbázis rendszerrel, ilyen vagy olyan perzisztencia réteggel, ilyen vagy olyan MVC keretrendszerrel működik. Az sem ritka, hogy valaki pusztán fontoskodásból akadékoskodik, szervez véget nem érő megbeszéléseket, követel senkinek sem kellő dokumentációkat, miközben a döntés valójában üzleti szemponttól nem releváns. Ha mi se tudjuk jobban, bízzuk inkább a fejlesztőkre, mintsem hogy akadályozzuk a haladást.

- túltervezés (*overengineering*)

Egyrészt akkor jelentkezik ha valaki mindent a kezében szeretne tartani, mindenben kritikus problémákat vél felfedezni, másrészt ha mindenre saját megoldást, saját tervet követel. Ezeket mégse képes időben leszállítani, vagy ha le is szállítja mégse kapunk teljes megoldást, vagy igazából nem is látjuk mi a valójában a probléma. Ez gyakran néhány személy hozzá nem értésének az eredménye akitől meg kell vonni a döntési jogkört.

- túlabsztrahálás (*overabstraction*)

Jellemzően folyamatos spekuláció („mi lesz ha”) és megalapozatlan nagyvállalati tervminták hozadéka. Ha egy interfésznek csak egy implementációja van tervben, ne készítsünk interfészt, csak implementációt. Ha egy alkalmazásrétegnek nincs üzleti célja, ne hozzuk létre. Ha egy funkció nem megalapozott tervbe vett felhasználói esettel, ne implementáljuk. Legyünk rendkívül kritikusak és törekedjünk a szigorú minimumra, a spekuláció csak kifogáskeresés.

Ezeket követve sikeresen szállítjuk le termékünket a nagyközönség elé. Fontos pillanat ez, hisz minden a puding próbája az evés (*dogs gonna eat dog food*). Ezek során a fejlesztésről átkerül a hangsúly a marketingre, ami nem tárgya e írásnak. Annyit megemlítenék, hogy ez nem triviális és nem technikai terep, szakértelmet és odafigyelést igényel. Végtelen pénzt lehet marketingre elkölteni, eredmények nélkül, és kicsi pénzből

is lehet jó marketinget csinálni. Ajánlásom, hogy lényegesen több pénzt költsünk marketingre mint fejlesztésre, ne kövessük el azt a hibát, hogy a sivatag közepére építünk áruházat⁵. Ha nem marad pénzünk és nincs valódi (nem spekuláción, közhelyeken alapuló) marketing tervünk a termékünk felhasználókhoz való eljuttatására, nem sok esély van a sikerre⁶. Tapasztalatom szerint a „jó bor eladja magát” inkább városi legenda mintsem marketing valóság. (Persze szűk, alaposan ismert célközönség esetén igaznak tűnhet, hisz pl. egy kis faluban mindenki mindenről tud, a termék szájról-szájra terjed.)

6. Konklúzió

Sikeres szállítás után a terméket birtokba veszik a felhasználóink. Ismét idézzük fel, a valódi termék maga a felhasználói élmény. Mérések és metrikák segítenek ebben a fázisban, hogy megértsük valójában milyen élményt is nyújt a termék. Ezek segítségével elkerülhetjük, hogy saját prekoncepcióink áldozatává váljunk. Gyakori, hogy a felhasználók egy része (vagy egésze) máshogy fogja fel, máshogy értelmezi a funkciókat, mint ahogy mi hisszük, elképzeltük, megterveztük, megvalósítottuk. Nehezíti a felismerést, hogy mi hónapokon keresztül dolgoztunk és mélyen belénk ivódott egy konkrét nézet, ismeret és felhasználás, míg a valódi felhasználók tabula rasa, először találkozva a termékkel, teljesen eltérő képet kaphatnak. Léteznek eszközök, hogy időben felismerjük és reagáljunk az ilyen helyzetekre:

- vakteszt (*hallway usability testing*)

Mutassuk meg a kész vagy félkész terméket véletlen kollégáknak, ismerősöknek teljesen váratlanul és minden magyarázat nélkül. Értik? Felismerik? Tudják használni? Ezt megtehetjük már a tervezési folyamat során egy prototípuson vagy félkész fejlesztői változaton is, így időben értesülünk problémákról, félreértésekről; szükséges módosításokat időben eszközölhetünk. (Általános igazság szoftverfejlesztés során, hogy egy hibás döntés és a hiba

⁵A vanmit.hu esetében ez történt, nagyon kevesen tudnak a létezéséről, nincs meg a működéshez szükséges kritikus tömeg.

⁶A vanmit.hu esetén elmondható, kis túlzással, hogy nem volt marketing tervünk. A közösségi oldalakon bohóckodás, és néhány fizetett hirdetés messze nem váltotta be a reményeket. Próbálkoztunk fizetett marketingesekkel is, akik maguk sem rendelkeztek jobb tervvel.

korrekciója között lévő időbeli távolsággal exponenciálisan nő a javítás költsége.)

- látogatottság (*visitor analytics*) Hány oldalletöltésünk van naponta? Hány egyedi látogatónk van? Mennyi az átlagos új egyedi látogató? Melyek a leglátogatottabb oldalak? Milyen régiókból jönnek? Milyen eszközöket (*user agent*) használnak? Milyen csatornákból érkeznek (hirdetések, közösségi oldal, blogok, keresők, közvetlen látogatók)? Meddig maradnak az oldalon? Milyen arányban hagyják el az oldalt (*bounce rate*)? Milyen utat járnak be tipikusan az oldalon?
- kulcsszó analitika (*keyword analytics*)
Milyen kulcsszavakkal lehet rátalálni a termékünkre? Milyen oldalak linkelnek ránk? Keresőkben az oldalunk megfelelően és releváns információkkal jelenik-e meg? Kihasználunk-e minden SEO technikát nemcsak a főoldalunkon de az aloldalakon is?
- konverzió (*conversion*)
Ez a metrika különösen értékes mert a bevételek becslését is lehetővé teszi. Ezzel azt mérjük, hogy egy adott üzletileg (és potenciálisan bevételileg) releváns lépéssorozatot hányan követnek végig ill. hányan hagyják el. Például az oldalra látogatók közül hányan regisztrálnak? A regisztrálók közül hányan *vásárolnak* (ide helyettesítsünk be bármilyen a termék által nyújtott üzletileg releváns funkciót, pl. üzenetküldés, keresés)? Milyen korreláció van csatornák és funkciók között? (Lehetséges pl. hogy egy reklámkampányból sok látogató érkezik, de alig regisztrálnak!)
- A/B teszt (*A/B testing*) Ez a metrika a saját prekonceptióinkat validálja. Úgy működik, hogy a felhasználóknak véletlenszerűen máshogy teszi elérhetővé ugyanazt a funkciót. Például más bejelentkezési, regisztrációs felületet ad. Lehet ez apró különbség (szín, megjelenés) de jelentősebb is (egy vagy több lépéses, egyszerűsített formok, extra validációk, mint captcha kihagyása, más aggregált nézetek mutatása). Ezek az ún. A/B variánsok, melyeknek a konverzióit külön-külön mérve megismerhetjük felhasználóink értelmezését, szokásait, preferenciáit. Vegyünk itt figyelembe korrelációt más dimenziókkal. Például ugyanarra az

A/B tesztre más régió (US/EU), más eszközök (mobile/desktop) felhasználói más eredményt produkálhatnak.

- Kapcsolat, PR Hasznos visszajelzéseket kaphatunk közvetlenül felhasználóinktól. Érdekes egyszerűen, regisztráció és e-mail cím megadása nélkül elérhetővé tenni kapcsolat és hibajelentés funkciókat. A termék által küldött e-maileket ne egy *noreply*, hanem egy kapcsolat e-mail címről küldjük.

Az Internet egy gyorsan mozgó célpont, így gyorsan kiderül, hogy megfelelő célközönséggel sikeresek lehetünk-e vagy sem.⁷

7. Utószó

Siker esetén nincs más dolgunk mint hátradőlni. Ez se tarthat sokáig, fontos kérdések várnak minket a jövőben. Hogyan skálázzuk az termékünket, kiadásainkat, bevételeket? Hogyan bővítsük piacunkat? Milyen befektetések, fejlesztések hoznák a legnagyobb növekedést? Különösen nagy siker esetén fenyeget a veszély, hogy áldozatává válunk ennek. Ekkor fontos, hogy megbízható és kompetens emberekkel vegyük magunkat körül, ne pedig keselyűkkel, és magunk se üljünk elefántcsonttoronyba. Tapasztalt technikus a sikertől nem válik jó üzletemberré.

Sikertelenség esetén se csüggedjünk. Egyrészt ne adjuk fel mielőtt nem veszítenénk (*don't give up before you fail*), másrészt ne is erőltessük azt, ami nem működik (Einstein szerint a hülyeség definíciója az, amikor ugyanazt cselekedjük újra-meg-újra mégis más eredményt remélve.) Ne felejtsük el levonni a következtetéseket sem, egy jó tudós mintájára, akinek a negatív eredmény is eredmény. Ha veszítünk, sose veszítsük el a tanulságot. (*If you lose, don't loose the lesson.*) (Többet tanulhatunk

⁷A vanmit.hu esetében a siker egyelőre elmaradt. Nem tartottuk be az MVP szabályait, jelentős erőforrásokat pazaroltunk lényegtelen dolgokra (pl. cégalapítás, képszerkesztés funkció, aukció és extra opciók). Túl sokáig fejlesztettünk, túl későn kaptunk visszajelzést. A fejlesztés túl sok iteráción ment át, rögtön reszponzív, mobil kompatibilis designnal kellett volna kezdenünk. Az erőforrások jelentős részét a fejlesztés felemésztette, nem maradt lényegében marketingre, az oldal ismeretlen. Nem határoztunk és céloztunk meg egy kellően szűk célközönséget. Jelenlegi üresjáratot arra használjuk, hogy egy új, leegyszerűsített MVP koncepcióval álljunk elő az év végére.

egy sikertelen vállalkozásból, mint egy drága de tisztán elméleti MBA képzésből.) Bátorítson minket az a tény, hogy tízből kilenc startup kudarcba fullad. Szilícium-völgyi mondás: ha kétszer nem buktál eddig legalább, akkor nem is lehetsz sikeres. (Bukott startupokat ne féljünk feltüntetni CV-n se.) A bukás lehetőséget tartsuk szem előtt kezdetektől: ha nem tartjuk lehetségesnek a bukást, ha nem buknánk büszkén és szívesen a csapatunkkal, ha nem számolunk eleve a teljes tőke potenciális elvesztésével, azt ajánlom, inkább bele se vágjunk.

Mindezek fényében szeretnék mindenkit arra bátorítani, hogy merje megvalósítani ötleteit, álmait. Mire való az élet, ha nem erre? Több beszélgetésben is volt részem, ahol egy tervet bemutatva valaki azt mondta, igen, erre én is gondoltam évekkel ezelőtt, de semmit sem tettem érte. Olyan időket élünk ahol startupok milliárdokat érnek, például az Instagram egy milliárd dolláros felvásárlásakor összesen 13 munkatárs és 9 befektető osztozott a pénzen. Nagy cégek jellemzően a meglévő pozícióikból élnek (oligopólium), nem jellemző, hogy képesek innovációra. Az Interneten piacra lépni nagyon alacsony költséggel, befektetéssel jár, csak saját magunk vagyunk a lehetőségeink korlátja, ugyanakkor a startupok szabályai a hagyományos vállalkozásokra, befektetésekre is igazak. Ne hagyjuk, hogy életünk meg nem valósított álmaink könyvévé váljon.

Hivatkozások

- [1] Széchenyi I., *Hitel*, <http://mek.oszk.hu/06100/06132/>
- [2] B. Aulet, *Disciplined Entrepreneurship: 24 Steps to a Successful Startup*, 2013., <http://www.amazon.com/gp/product/1118692284>
- [3] S. Sinek, *Start With Why*,
<https://www.youtube.com/watch?v=sioZd3AxmnE&sns=em>
- [4] Amir, *The Stanford Startup and the MIT startup*, Reconfigurable Computing blog, November 5, 2013.
<http://fpgacomputing.blogspot.sg/2013/11/the-stanford-startup-and-mit-startup.html>
- [5] B. Aulet, Our Dangerous Obsession with the MVP, *Techcrunch*, March 1, 2014.

<http://techcrunch.com/2014/03/01/our-dangerous-obsession-with-the-mvp/>

- [6] Wikibooks, *Introduction to Software Engineering*.
http://en.wikibooks.org/wiki/Introduction_to_Software_Engineering

Képregisztráció a frekvenciatartományban

Gilián Zoltán

Eötvös József Collegium*

`zoltan.gilian@gmail.com`

1. Bevezetés

A képregisztráció feladata egy objektumról vagy hasonló objektumokról különböző módon alkotott képek egymásra illesztése, azok közti megfeleltetés keresése. A képregisztráció lényeges lépés minden olyan probléma esetén, amely különböző képek információtartalmának egyesítését, feldolgozását tűzi ki célul. Az alkalmazási területek közt említhető a számítógépes látás, távérzékelés, és orvosi diagnosztika.

A probléma képkalkotási mód szerint négy kategóriába sorolható. A képek készülhettek különböző nézőpontokból, különböző időpontokban és különböző szenzorok segítségével, valamint lehet szó színtér-modell regisztrációról is. Ez utóbbi esetben a célobjektumról valamely előzetes elképzelés alapján felállított modell és az objektumról alkotott kép között keresünk megfeleltetést. A fenti kategóriákra példaként említhető rendre adott területről készült légifelvételek összefésülése, kameramozgás nyomon követése, egy betegről készült CT és MR képek regisztrációja, valamint automatikus célpontfelismerés.

A regisztrációs feladat megoldásakor több kérdés is felmerülhet. Rög-zítendő transzformációk egy halmaza, melyen a valamilyen szempontból legjobb transzformáció keresése zajlik. Ez a halmaz általában a hasonlósági, affin, projektív vagy deformálható transzformációk tere. További kérdés lehet, hogy mit jelent a legjobb transzformáció, hogyan mérjük egy transzformáció minőségét. A regisztráció a képek különféle jellemzői

*2010–

alapján történhet, ezek lehetnek például pont, él vagy területi jellemzők, de gyakran maguk az intenzitásértékek.

A továbbiakban a kétdimenziós diszkrét Fourier-transzformáción alapuló *fázis korrelációs* képregisztrációs eljárás [3] kerül bemutatásra. A szükséges elméleti háttér rövid áttekintését követően rátérünk a módszer tárgyalására, itt ismertetjük az eltolás detektálására képes alapváltozatot, majd pedig a hasonlósági transzformációkra kiterjesztett algoritmust.

2. Elméleti háttér

Az alábbiakban formalizáljuk a képregisztrációs feladatot, majd áttekintjük a diszkrét Fourier-transzformált módszer megértéséhez szükséges tulajdonságait.

A feldolgozandó képeket itt kétváltozós valós értékű függvényekkel modellezzük, amelyek adott térbeli ponthoz egy intenzitásértéket rendelnek hozzá. A trigonometrikus Fourier-transzformált értelmezéséhez a függvényekről megköveteljük, hogy Lebesgue-integrálhatóak legyenek, azaz $f, g \in L^1(\mathbb{R}^2)$. Képek, vagy általánosabban jelek ezen reprezentációját folytonos aperiodikus modellnek nevezzük.

Jelölje $\mathcal{T} \subset \{T : \mathbb{R}^2 \rightarrow \mathbb{R}^2\}$ a megengedett transzformációk valamely terét. A képregisztrációs feladat egy olyan $T \in \mathcal{T}$ transzformáció keresése, amelyre $g(T(x))$ és $f(x)$ ($x \in \mathbb{R}^2$) képek hasonlóak. Itt szándékosan nem vezetünk be a hasonlóság mérésére alkalmas függvényt, mert az itt tárgyalt algoritmus nem építkezik képek hasonlóságának a fogalmára. A fentiek mellett az $f(x, y)$, illetve a $g(T(x, y))$ ($x, y \in \mathbb{R}$) egyszerűsített jelöléseket is használjuk.

Képek számítógépes feldolgozásakor intenzitásértékeknek csupán véges sok pontban tekintett tömbje áll rendelkezésünkre, a térváltozók egy véges rács pontjain futnak végig. Ezen követelményekhez illeszkedő alapstruktúra definiálásához tekintsük a Fourier-transzformált értelmezéséhez szükséges $\mathbb{Z}_m$ modulo m maradékosztályok összeadással tekintett csoportját. Ennek segítségével kétdimenziós, $M \times N$ ($M, N \in \mathbb{N}^+$) méretű képek $\mathbb{Z}_M \times \mathbb{Z}_N \rightarrow \mathbb{R}$ típusú függvényekkel reprezentálhatóak. Az egyszerűség kedvéért legyenek a regisztrálandó képek azonos méretűek, így a $\mathbb{Z}_{MN} := \mathbb{Z}_M \times \mathbb{Z}_N$ jelöléssel $f, g : \mathbb{Z}_{MN} \rightarrow \mathbb{R}$. A térváltozók ily módon bevezetett periodicitása miatt a reprezentációt diszkrét periodikus modellnek nevezzük.

A fent bevezetett diszkrét változós függvények $\mathbb{R}^2 \rightarrow \mathbb{R}^2$ típusú leképezésekkel közvetlenül nem transzformálhatóak. A gyakorlatban az ilyen transzformációt interpolációs eljárás segítségével alkalmazzák a képekre. Az intenzitásértékek kvantálásától itt eltekintünk.

A diszkrét Fourier-transzformált központi jelentőségű a képfeldolgozás területén. Ismerete elengedhetetlen az itt tárgyalt algoritmus megértéséhez, ezért röviden kitérünk a definícióra és egy számunkra fontos tulajdonságra. Vezessük be az

$$\epsilon_\lambda(x) = e^{i2\pi\lambda^\top x} \quad (x, \lambda \in \mathbb{R}^2)$$

λ “frekvenciájú” alapfüggvényeket. Ezek olyan periodikus függvények, melyek valamely változójának egységnyi változása alatt λ megfelelő komponense által megadott periódust tesznek meg. A képlet vektorai oszlopvektorok, így $\lambda^\top x$ a λ és az x vektorok skaláris szorzatát jelenti.

A képeket szeretnénk különböző frekvenciájú alapfüggvények lineáris kombinációjaként felírni. Az m -szerint periodikus térváltozó csak m -szerint periodikus alapfüggvényt tartalmazhat, így a frekvencia csak $\frac{k}{m}$ alakú lehet. A $\frac{k}{m}$ és a $\frac{k+m}{m}$ frekvenciájú alapfüggvények $\mathbb{Z}$ pontjain nem különböztethetők meg, így elegendő például a $k = 0, 1, \dots, m-1$ esetekre szorítkozni. Valóban,

$$e^{i2\pi\frac{k+M}{M}j} = e^{i2\pi\frac{k}{M}j+i2\pi j} = e^{i2\pi\frac{k}{M}j} \quad (k, j \in \mathbb{Z}).$$

Válasszuk tehát az $\mathbb{M}_m = \frac{1}{m}\mathbb{Z}/\mathbb{Z}$ ($m \in \mathbb{N}^+$) faktorcsoportokat a megfelelő összeadás művelettel, vagy másként az

$$\mathbb{M}_m = \left\{ 0, \frac{1}{m}, \dots, \frac{m-1}{m} \right\}$$

halmazokat a modulo 1 összeadással a frekvenciaváltozók alapstruktúrája gyanánt. A többváltozós esetben a korábbiakhoz hasonlóan vezessük be az $\mathbb{M}_{MN} := \mathbb{M}_M \times \mathbb{M}_N$ jelölést.

Megmutatható, hogy az alapfüggvényekkel történő reprezentáció létezik és egyértelmű, a reprezentáció együtthatóit *diszkrét Fourier-transzformáltak* vagy röviden DFT-nek nevezik. A bevezetett jelölések mellett az $f: \mathbb{Z}_{MN} \rightarrow \mathbb{R}$ függvény diszkrét Fourier-transzformáltja a

$$\mathcal{F}f(\lambda) = \sum_{x \in \mathbb{Z}_{MN}} f(x) \overline{\epsilon_\lambda(x)} \quad (\lambda \in \mathbb{M}_{MN})$$

képlettel állítható elő. A diszkrét Fourier-transzformált ismeretében az eredeti függvény visszaállítható az *inverz diszkrét Fourier-transzformált* segítségével:

$$\mathcal{F}^{-1}(\mathcal{F}f)(x) = \frac{1}{MN} \sum_{\lambda \in \mathbb{M}_{MN}} \mathcal{F}f(\lambda) \epsilon_{\lambda}(x) = f(x) \quad (x \in \mathbb{Z}_{MN}).$$

Az itt tárgyalt algoritmus alapja a diszkrét Fourier-transzformált eltolási tulajdonsága. Jelölje rögzített $t \in \mathbb{Z}_{MN}$ mellett τ_t a t -vel történő *eltolási operátort*, amely tetszőleges f függvényhez hozzárendeli a

$$\tau_t f(x) = f(x + t) \quad (x \in \mathbb{Z}_{MN})$$

szabállyal definiált függvényt. Szükségünk lesz továbbá a ν_a ($a \in \mathbb{R}$) ún. *modulációs operátorra*, amelyet a

$$\nu_a f(x) = \epsilon_a(x) f(x) \quad (x \in \mathbb{Z}_{MN})$$

képlettel definiálunk. Megmutatható, hogy a DFT az eltolást modulációba viszi át:

$$\mathcal{F}\{\tau_a f\} = \nu_a \mathcal{F}f \quad (a \in \mathbb{Z}_{NM}). \quad (1)$$

A diszkrét Fourier-transzformált előállítására naiv algoritmussal $\mathcal{O}(N^2)$ műveletet vesz igénybe, ahol N a képpontok száma. Ez jelentős gátat emelne a gyakorlati alkalmazhatóság elé. A DFT kiszámítására azonban létezik egy $\mathcal{O}(N \log N)$ műveletigényű algoritmus, a *gyors Fourier-transzformáció* (Fast Fourier Transform, FFT), lásd pl. [1]. DFT megvalósításakor célszerű FFT algoritmust alkalmazni.

A DFT folytonos, aperiodikus modellbeli megfelelője a trigonometrikus Fourier-transzformált, melynek definícióját a teljesség kedvéért említjük.

$$\mathcal{F}f(\lambda) = \int_{\mathbb{R}^2} f(x) \overline{\epsilon_{\lambda}(x)} dx \quad (\lambda \in \mathbb{R}^2),$$

ahol $f \in L^1(\mathbb{R}^2)$. A folytonos aperiodikus és a diszkrét periodikus modell közti átjárást a mintavételezés és a periodizáció adja. A modellek pontos viszonyára itt nem térünk ki, az érdeklődő olvasó például a [4] jegyzetben mélyebb betekintést nyerhet a témakör rejtelmeibe.

3. A módszer

A következőkben egy hasonlósági transzformációmodellt használó képregisztrációs algoritmust mutatunk be, amely a transzformáció paramétereit – eltolás, forgatás és nagyítást illetve kicsinyítést – a diszkrét Fourier-transzformáció segítségével becsli. Az eljárás alapötlete két kép közötti eltolás megtalálására alkalmas, míg egy továbbfejlesztett változata a forgatást és a nagyítást illetve kicsinyítést vagy más néven skálázást is képes kezelni.

3.1. Eltolás detektálása

Tekintsük az $f, g : \mathbb{Z}_{MN} \rightarrow \mathbb{R}$ képeket, melyek egymás periodikus eltoltjai, azaz fennáll valamely $[t_x \ t_y]^\top = t \in \mathbb{Z}_{MN}$ eltolásvektorral az

$$f = \tau_{-t}g$$

összefüggés. Ekkor (1) alapján

$$\mathcal{F}f(\lambda) = \epsilon_{-t}(\lambda)\mathcal{F}g(\lambda). \quad (2)$$

Tekintsük az

$$R(\lambda) = \frac{\mathcal{F}f(\lambda)\overline{\mathcal{F}g(\lambda)}}{|\mathcal{F}f(\lambda)\mathcal{F}g(\lambda)|} \quad (3)$$

függvényt. Ha (2) fennáll, akkor

$$R(\lambda) = \epsilon_{-t}(\lambda) = e^{-i2\pi t^\top \lambda}. \quad (4)$$

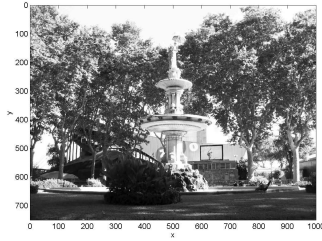
Könnyen ellenőrizhető, hogy (4) inverz diszkrét Fourier-transzformáltjára

$$\mathcal{F}^{-1}R(x) = \mathcal{F}^{-1}\epsilon_{-t}(x) = \begin{cases} 1 & \text{ha } x = t \\ 0 & \text{egyébként} \end{cases}$$

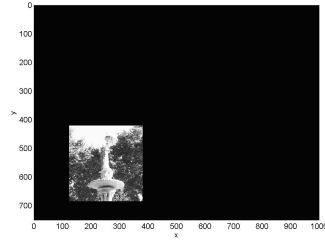
teljesül. Az eltolásvektor előállítható az $\mathcal{F}^{-1}R$ maximumhelyének megkeresésével.

A módszer tehát a következő:

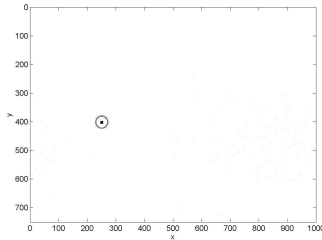
- Előállítjuk a képek $\mathcal{F}f$ és $\mathcal{F}g$ diszkrét Fourier-transzformáltjait. Ehhez célszerű FFT algoritmust alkalmazni.



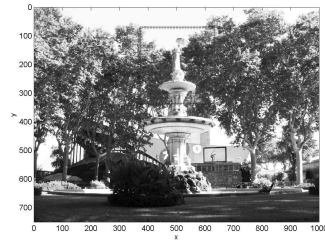
(a) Első kép



(b) Második kép



(c) Keresztkorreláció



(d) A regisztráció eredménye

1. ábra. A fázis korrelációs módszer működése. A keresztkorrelációs függvény maximumhelye adja az optimális eltolást. A modell periodicitása miatt az eltolásértékek csak a megfelelő maradékosztályok erejéig meghatározottak, itt például az y irányú eltolásból ki kell vonni a kép magasságát, hogy megkapjuk a tényleges eltolást.

- (3) alapján kiszámítjuk az R függvény $\lambda \in \mathbb{M}_{MN}$ pontokban felvett értékeit.
- Előállítjuk az $\mathcal{F}^{-1}R$ inverz diszkrét Fourier-transzformáltat. Ezt szintén FFT algoritmussal érdemes megvalósítani.
- Megkeressük az $\mathcal{F}^{-1}R$ függvény $[t_x t_y]^\top$ maximumhelyét, amely megadja a keresett eltolást.

Az algoritmus működését az 1. ábra illusztrálja.

A fent ismertetett módszert *fázis korrelációs* eljárásnak (phase correlation) nevezik. Az algoritmus tulajdonképpen a két kép keresztkor-

relációját állítja elő, ami a frekvenciatartományra történő áttéréssel és az FFT algoritmus bevetésével a naiv implementációhoz viszonyítva jelentős gyorsulást eredményez.

3.2. Forgatás és skálázás detektálása

Az előző szakaszban eltolás által egymásba vihető képek regisztrációjára adtunk algoritmust. Az ötletet az alábbiakban kiterjesztjük forgatást és skálázást is kezelni képes eljárásra.

A kiterjesztés alapja a képek logpolár-transzformációja. Az $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ kép logpolár-transzformáltján a

$$\mathcal{P}f(\log r, \theta) = f(r \cos \theta, r \sin \theta) \quad (r \in \mathbb{R}^+, \theta \in [0, 2\pi))$$

képlettel definiált függvényt értjük. Diszkrét képek logpolár-transzformáltja a 2. szakaszban tett megjegyzéseknek megfelelően diszkrét pontokban kiértékelve, interpolációval adódik.

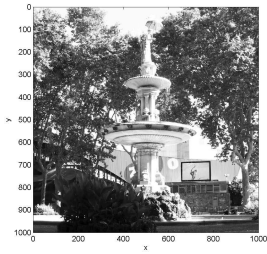
A logpolár-transzformált előnye, hogy f forgatása és skálázása $\mathcal{P}f$ eltolását eredményezi:

$$\tau_{\log s, \varphi} \mathcal{P}f(\log r, \theta) = \mathcal{P}f(\log rs, \theta + \varphi) = f(rs \cos(\theta + \varphi), rs \sin(\theta + \varphi)).$$

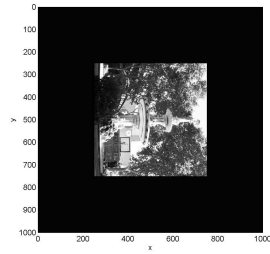
A fázis korrelációs eljárással a forgatási szög és a skálázási arány így meghatározható. Az eljárást a 2. ábrán szemléltetjük.

Problémák merülhetnek fel, ha forgatás és skálázás mellett eltolást is figyelembe kell venni, ugyanis az eredeti kép eltolása a logpolár-transzformált egy nemlineáris transzformációját jelenti. Ennek megoldására tekintsük az eredeti kép helyett a DFT abszolútértékének logpolár-transzformáltját. Ennek közvetlen előnye, hogy az (1) eltolási összefüggés alapján $|\mathcal{F}f|$ érzéketlen az eltolásokra. Fennáll továbbá az invertálható mátrixú lineáris változótranszformáció és a trigonometrikus Fourier-transzformáció közti alábbi összefüggés:

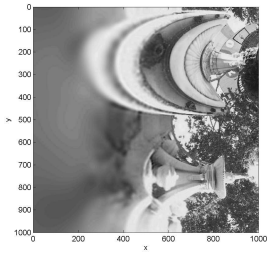
$$\begin{aligned} \mathcal{F}\{f(Ax)\}(\lambda) &= \int_{\mathbb{R}^2} f(A\lambda) \overline{\epsilon_\lambda(x)} d\lambda \\ &= \frac{1}{|A|} \int_{\mathbb{R}^2} f(\lambda) \overline{\epsilon_{A^{-1}\lambda}(x)} d\lambda \\ &= \frac{1}{|A|} \mathcal{F}f(A^{-1}\lambda). \end{aligned}$$



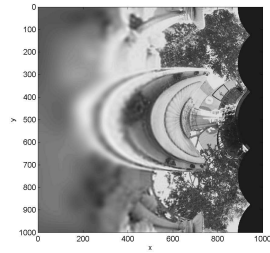
(a) Első kép



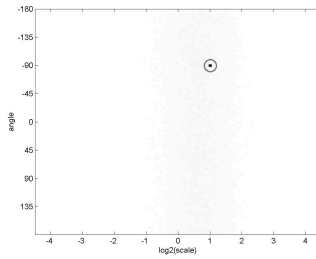
(b) Második kép



(c) (2a) logpolár-transzformáltja



(d) (2b) logpolár-transzformáltja



(e) Keresztkorreláció

2. ábra. Fázis korrelációs módszer logpolár-transzformációval. Az első kép a második -90° szögű elforgatása ill. 2 arányú nagyítása. Az algoritmus által detektált forgatási szög -89.64° és átméretezési arány 2.0197.

ahol $A \in \mathbb{R}^{2 \times 2}$ invertálható. A második egyenlőségnél változóhelyettesítést alkalmaztunk. Speciálisan ha

$$R = \begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix}$$

θ szöggel való forgatás mátrixa, akkor a forgatást és skálázást leíró $A = sR$ ($s \in \mathbb{R}^+$) mátrixra

$$\mathcal{F}\{f(sRx)\}(\lambda) = \frac{1}{s^2} \mathcal{F}f\left(\frac{1}{s} R^\top x\right).$$

Az elforgatott és skálázáson átesett kép trigonometrikus Fourier-transzformáltja tehát az $\frac{1}{s^2}$ szorzótól eltekintve a forgatási szög -1 -szeresével fordul el, illetve a skálázási szorzó reciprokéval skálázódik. Ez az észrevétel indokolja, hogy a forgatást és skálázást a DFT abszolút értékének logpolár-transzformáltja segítségével keressük az eredeti kép logpolár-transzformáltja helyett. Mivel a DFT abszolútértéke 180° -os forgatásra érzéketlen, ezért a fázis korrelációval kapott θ szög mellett a $\theta + \pi$ szöget is meg kell vizsgálni.

Az eltolást, forgatást és skálázást is figyelembe vevő regisztrációs algoritmus tehát a következő lépésekből áll (most ismét $f, g : \mathbb{Z}_{MN} \rightarrow \mathbb{R}$):

- Előállítjuk az $F := |\mathcal{F}f|$ és a $G := |\mathcal{F}g|$ mátrixokat, célszerűen FFT algoritmus segítségével.
- Kiszámoljuk az F és a G logpolár-transzformáltját.
- Fázis korrelációs algoritmussal meghatározzuk a képek közti θ forgatási szöget és s skálázási szorzót.
- A g képet skálázzuk az s szorzóval, majd elforgatjuk a θ valamint $\theta + \pi$ szögekkel, így nyerjük a g_1 illetve a g_2 képeket.
- Fázis korrelációs eljárással meghatározzuk az eltolási viszonyt f és g_1 valamint f és g_2 képek között. Amely esetben a fázis korrelációs csúcs magasabb, az ahhoz tartozó transzformációs paramétereket fogadjuk el mérvadónak.

A fenti algoritmus sajnos nagyon érzékeny a zajokra, közvetlen alkalmazása nem javasolt. Ennek ellenére a logpolár-transzformáció ötlete igen fontos eredmény, ennek felhasználásával robusztus algoritmusok fejleszthetőek [3, 5].

4. Összefoglalás

Az előző néhány oldalban ismertettük a képregisztráció problémakörét, valamint röviden ismertettünk egy regisztrációs algoritmust, amely eltolások illetve forgatás és skálázás által egymásba vihető képek közt keres ilyen transzformációt. Itt csupán az alapötlet felvetését tűztük ki célul, mind az elmélet mélyebb áttekintése, mind az algoritmus javításának, robusztusabbá tételének tárgyalása túlmutat a jelenlegi kereteken. A témakörrel az érdeklődő olvasó a [2, 3, 5] munkákban tájékozódhat.

Hivatkozások

- [1] Járai A., *Bevezetés a matematikába*, 3. kiad. (2009), ELTE Eötvös Kiadó, pp. 332–334.
- [2] H. S. Stone, M. T. Orchard, E-C. Chang, S. A. Martucci, A fast direct fourier-based algorithm for subpixel registration of images, *IEEE Transactions on Geoscience and Remote Sensing*, 39(10) (2001), pp. 2235–2243.
- [3] B. S. Reddy, B. N. Chatterji, An fft-based technique for translation, rotation, and scale-invariant image registration, *IEEE Transactions on Image Processing*, 5(8) (1996), pp. 1266–1271.
- [4] Schipp F., Fourier-analízis, 2006., ELTE jegyzet.
- [5] M. McGuire, An image registration technique for recovering rotation, scale and translation parameters, *NEC Res. Inst. Tech. Rep., TR*, 1998., pp. 98–018.

Aspektusok Scalához*

Góbi Attila, Kozsik Tamás, Vezér Boglárka

Eötvös Loránd Tudományegyetem, Informatikai Kar

`gobi@pnyf.inf.elte.hu`; `kto@inf.elte.hu`; `vezerb@gmail.com`

A programozók körében egyre nagyobb népszerűségnek örvendnek a multiparadigmás nyelvek. Az objektumelvű és a funkcionális programozási paradigmák integrációja a szoftveriparban alkalmazott nyelvek esetében is megjelenik. Erre egy jó példa a *Java Virtuális Gépre* (JVM-re) forduló Scala nyelv. A Java virtuális gép használatával a Scala programokba közvetlenül beépíthetünk Java osztályokat is, valamint a korábban létrehozott Java programok akár Scala kóddal is bővíthetők.

Ennek a megközelítésnek a fent említett előnyén kívül hátrányai is vannak: a Scala nyelven írt osztályok nem mindig a legkézenfekvőbb módszerrel fordulnak bájtkódra, így a Java osztályok a Scala nyelven írt definíciókat esetenként elég körülményesen tudják csak meghivatkozni. Különösképpen nagy problémákba ütközünk, ha egy olyan szoftverbe szeretnénk Scala komponenseket integrálni, amely a Java mellett annak népszerű aspektusorientált kiterjesztését, az AspectJ-t [3] is használja.

Az aspektusorientált programozás egy napjainkban szintén elterjedt paradigma, amelynek egyik legismertebb implementációja az AspectJ nyelv és szövőprogram. Az AspectJ a Java nyelvet egészíti ki aspektusorientált eszközökkel. Elterjedtségét jól mutatja, hogy a népszerű Spring keretrendszer közvetlenül támogatja az AspectJ nyelvet.

A Java virtuális gép, mint közös platform, azonnal felkeltheti az érdeklődésünket: mi történik, ha a Javát, a Scalát és az AspectJ-t együttműködésre szeretnénk bírni? Volna értelme Scala programokat aspektusokkal bővíteni? Gazdagítaná ez a programozói eszközkészletünket? Megvalósítható technikailag az együttműködés? Ebben a cikkben arra keressük a választ, hogy az említett nyelvek és fordítóprogramok

*Támogatta a 18370-9/2013/TUDPOL számú pályázat.

mennyire kombinálhatóak, és milyen várt vagy nem várt működést produkálnak: a Scalából készített osztályokra képes-e szőni az AspectJ, és az így szőtt kód azt csinálja-e, amit elképzeltünk. Megvizsgáljuk, hogy melyek azok a Scala konstrukciók, amelyre a szövéss az elvárt viselkedést adja.

A következő szakasz a Java virtuális gép működését mutatja be nagy vonalakban. A második szakaszban ismertetjük az aspektusorientált programozást [4], és annak használatát AspectJ-ben. A harmadik szakasz azt elemzi, hogy a Scala fordító a különböző konstrukciókból milyen bájtkódot generál. Itt megvizsgáljuk, hogy a generált bájtkód szövéssével milyen eredményeket lehet elérni. Az utolsó szakasz levonja a tanulságokat, miközben bemutatja mások ezen a téren eddig elért eredményeit.

1. A Java bájtkód

A Java Virtuális Gép (JVM, Java Virtual Machine) az az elképzelt gép, amelyre a Java nyelvet a Java fordító lefordítja. Ennek a gépnek a bináris kódját Java bájtkódnak nevezik. Egy konkrét platform felett megvalósítva a virtuális gépet (például a bájtkód interpretálásával) a platform Java programok futtatására alkalmassá válik. Ez az, ami a Java nyelv platformfüggetlenségét biztosítja. Azáltal, hogy a bájtkód és a hozzá tartozó fájlformátum (*class* fájl) szabványosított szerkezetű, egy *class* fájl ugyanazt jelenti minden platformon.

A Java virtuális gép kódja jól illeszkedik a Java nyelvhez, de nem korlátozódik pusztán a Java nyelv kiszolgálására. Más nyelvekből, például Scalából is fordíthatunk Java bájtkódra, és ezzel kiaknázhatjuk a hordozhatósággal, elterjedtséggel, támogatottsággal járó előnyöket.

Cikkünkben a Java bájtkód kimerítő bemutatására nem vállalkozunk. Az alább szereplő példák a bájtkódok mély ismerete nélkül is megérthetők. A fogalomrendszer gyors áttekintéséhez vizsgáljunk meg egy egyszerű Java osztályt, illetve a belőle készülő bájtkód fontosabb részleteit. Képzeljük el, hogy van egy *Time* osztályunk, mellyel órát és percet reprezentálhatunk. Ennek egy példányát hozzuk létre a *main* metódusban, értékét megnöveljük 137 perccel, majd az eredményt kiírjuk a szabványos kimenetre.

```
public class Main {
```

```

    public static void main( String[] args ){
        Time t = new Time();    // 00:00
        t.advance(137);         // add 137 minutes
        System.out.println(t);  // 02:17
    }
}

```

1. kódlista. Java osztálydefiníció

A fenti `Main.java` forrást lefordítva megkapjuk a `Main` osztály class fájlját, amelyet ezután a Java SE Development Kit részét alkotó `javap` programmal (Java Class File Disassembler) tudunk megvizsgálni. A `Main.main` metódus bájtkódja a következőképpen néz ki.

```

public static void main(java.lang.String[]);
Code:
  0: new #2 // class Time
  3: dup
  4: invokespecial #3 // Method Time."<init>":()V
  7: astore_1
  8: aload_1
  9: sipush 137
 12: invokevirtual #4 // Method Time.advance:(I)V
 15: getstatic #5 // Field java/lang/System.out:Ljava/io/PrintStream;
 18: aload_1
 19: invokevirtual #6 // Method
    java/io/PrintStream.println:(Ljava/lang/Object;)V
 22: return

```

2. kódlista. Egy metódus bájtkódja

A példányosítás memóriaallokációval kezdődik (`new`), majd a konstruktor lefuttatásával (`invokespecial`) folytatódik. A paraméterek (`t` és `137`) előkészítése után az `advance` példánymetódus kerül végrehajtásra (dinamikus kötés: `invokevirtual`). Ezután meghívódik a `java.lang.Object` paraméterű `java.io.PrintStream.println` példánymetódus a `System.out` és a `t` paraméterekkel: az előbbit a `getstatic`, az utóbbit az `aload_1` készíti elő. A kettőskereszt utáni számokkal kapcsolatban az úgynevezett „Constant pool” segítségével tájékozódhatunk.

```

Constant pool:
  #1 = Methodref #8.#17 // java/lang/Object."<init>":()V
  #2 = Class #18 // Time
  #3 = Methodref #2.#17 // Time."<init>":()V

```

```
#4 = Methodref #2.#19 // Time.advance:(I)V
#5 = Fieldref #20.#21 // java/lang/System.out:Ljava/io/PrintStream;
...
#18 = Utf8 Time
#19 = NameAndType #26:#27 // advance:(I)V
#20 = Class #28 // java/lang/System
#21 = NameAndType #29:#30 // out:Ljava/io/PrintStream;
...
#26 = Utf8 advance
#27 = Utf8 (I)V
#28 = Utf8 java/lang/System
#29 = Utf8 out
#30 = Utf8 Ljava/io/PrintStream;
```

3. kódlista. Részletek a Main osztály bájtkódjából

Végigkövethetjük például a `Time.advance` metódus meghívását, mely a #4 hivatkozás segítségével történik. A #4 egy `Methodref`, mely a #2 osztályra és #19 szignatúrára hivatkozik. A #2 szimbólumhoz tartozó osztálynév a #18, amihez a `Time` érték tartozik. A #19 szignatúra elemeit a #26, valamint a #27 határozza meg. Az előbbi az `advance` nevet, az utóbbi az `(I)V` paraméterlista-specifikációt (int paraméterű, void visszatérési értékű) tárolja.

2. Az AspectJ nyelv

Az aspektus-orientált programozás a programkód modularizálásában kíván segíteni. A paradigma alapvetése, hogy egy hagyományos módszerrel modulokra bontott szoftverben lesznek olyan funkcionalitások (más szóval vonatkozások, concerns), melyek implementációja szétszóródik több modulra, és összekeveredik más vonatkozások kódjával. Ezeket az úgynevezett keresztbe vágó vonatkozásokat (cross-cutting concerns) tudjuk külön modulokba, aspektusokba kiemelni, a fő modulfelbontásról leválasztani az AOP segítségével. Közismert példák aspektusok használatára a naplózás (logging) és a jogosultság-ellenőrzés (authorization). Ezek olyan programfunkciók, amelyek a teljes programon végighúzódnak, és hagyományos, például objektum-orientált megközelítéssel nem igazán jól modularizálhatók. A keresztbe vágó vonatkozások az egész programban szétszóródó megvalósítása jelentősen hozzájárul a program komplexitásához, nehezíti a kód megértését és karbantartását.

Az aspektusok segítségével egy modulba gyűjthetjük egy keresztbe

vágó vonatkozás megvalósításához tartozó kódunkat. A szoftvernek továbbra is lesz egy elsődleges dekompozíciója, egy alapvető modulokra bontása, amely például Java esetén lehet egy objektumelvű megközelítésnek megfelelő, típusdefiníciókból álló struktúra, de ezt most kiegészíthetjük másfajta modulokkal, aspektusokkal is. Az aspektusok és az elsődleges modulfelbontásból származó definíciók összefüggő programmá való alakítását aspektusszövésnek (*aspect weaving*) nevezzük. Ennek során az aspektusokban leírt viselkedést beillesztjük az eredeti modulok minden olyan pontjára (*join point*), ahol ez szükséges. Ehhez persze az kell, hogy az aspektusokból meg tudjuk nevezni a *join point*okat, azaz hivatkozni tudjunk az eredeti modulfelbontással előállt struktúra egyes részeire. A módszernek akkor van leginkább haszna, ha az aspektusokban olyan kódrészleteket tudunk összegyűjteni, amelyeket több helyre is be kell szőni, azaz amikor a szövést több *join point*on rendeljük el (kvantált hivatkozás *join point*okra, *quantification*).

A Java nyelv egy aspektusorientált kiterjesztése az AspectJ. Az AspectJ csupán pár nyelvi elemmel bővíti ki a Javát, hogy a fenti célokat elérje.

- A *pointcut*-kifejezések *join point*ok halmazát nevezik meg, valamint hivatkozhatunk segítségükkel a *join point*okon elérhető programértékekre. *Pointcut*oknak (*pointcut*-deklarációkkal) nevet is adhatunk.
- Az *advice* egy alprogramszerű programegység, amelyet egy *pointcut*tal megnevezett *join point*ok elérésekor hajthatunk végre. Az AspectJ *before*, *after* és *around* típusú *advice*-t támogat, melyek rendre a hivatkozott *join point* előtt, után, illetve helyett/körül hívódnak meg.
- Az *inter-type declaration* segítségével típusok mezőkkel és metódusokkal való kiterjesztését, öröklődési kapcsolatok finomítását, domain-specifikus szemantikus összefüggések fordítási idejű ellenőrzését írhatjuk elő.
- Az *aspect* az a modul, amelyben egy keresztbe vágó vonatkozás megvalósításához szükséges *pointcut*ok, *advice*-ok, *inter-type* deklarációk és Java elemek bevezetésre kerülnek.

Az aspektusszövés folyamatát egy példán keresztül mutatjuk be. Tekintsük a már emlegetett *Time* osztályt egy részletét.

```
public class Time {
    private int hour, min;
    public int getHour(){ return hour; }
    public int getMin(){ return min; }
    public void advance(int minutes){
        min += minutes;
        hour += min/60;
        min %= 60;
    }
    @Override public String toString(){
        return String.format("%1$02d:%2$02d",hour,min);
    }
}
```

4. kódlista. A Time osztály

A `Time.advance` metódus egy végrehajtása kapcsán kétféle join pointot is azonosíthatunk: a metódus meghívását, illetve a metódustörzs lefutását. Az előbbit *call*, az utóbbit *execution* join pointnak nevezzük. Ha rászövünk ezekre a join pointokra, akkor az aspektusszöveő az első esetben a hívást tartalmazó kódokat változtatja meg, a második esetben pedig a metódus kódját transzformálja.

Mind az *execution*, mind a *call* joint pointra van lehetőség advice-okat szőni, melyek futhatnak a metódus *előtt* (before), *után* (after), illetve a metódus vagy a metódushívás *körül*. Ez utóbbi tulajdonképpen azt jelenti, hogy az advice fut le a metódus helyett, és az advice-ban lehetőségünk van úgy dönteni, hogy a metódushívást ténylegesen végrehajtjuk (a *proceed* utasítással kezdeményezhetjük a join point végrehajtását a köré szőtt around advice-ban). A következő aspektus, mellyel (a design-by-contract szemléletet követve) szerződéseket fogalmazunk meg a `Time` osztályhoz, ezeket demonstrálja.

```
public aspect Contracts {

    void around(Time time, int mins):
    execution(public void Time.advance(int)) && args(mins) && this(time)
    {
        int oldMinutes = time.getHour() * 60 + time.getMin();
        proceed(time,mins);
        int newMinutes = time.getHour() * 60 + time.getMin();
        if( oldMinutes + mins != newMinutes )
            throw new AssertionError("Time.add: postcondition violated!");
    }
}
```

```

    after(Time time) returning:
    call(public * *.*(..)) && target(time) && !within(Contracts) {
        int min = time.getMin();
        if( min < 0 || min >= 60 )
            throw new AssertionError("Time: class invariant violated!");
    }
}

```

5. kódlista. AspectJ példa

Az első advice (**around execution**) metódustörzs végrehajtása köré szövődik, a második (**after call**) pedig metódus hívása utánra. Az első advice a **Time.advance** metódus törzsét módosítja. A megadott pointcut kifejezés nem csak a szöveg helyét adja meg: felfogja a metódusvégrehajtás paramétereit is. A **this** pointcut segítségével köthetünk nevet az objektumhoz, amire a metódust meghívták, az **args** segítségével pedig a paraméterlistában szereplő paraméterhez rendelhetünk nevet. A két paramétert az advice paraméterlistájában deklarálva a statikus típushelyességről is gondoskodik az aspektusszövő. Az eredeti metódustörzs (**proceed(time,mins)**) végrehajtása előtti és utáni állapotot összehasonlítva az advice eldönti, hogy a metódus megfelel-e az elvárt viselkedésnek, és (nem ellenőrzött) kivételt vált ki, ha a **Time.advance** nem felel meg a szerződésének.

A második advice nem csak egy metódus meghívására hivatkozik, hanem a **Time** osztály minden publikus metódusáéra, a visszatérési érték típusától, valamint formális paraméterek számától és típusától függetlenül. A join pointok kvantálását a pointcutban a csillag szimbólum és a két pont értelemszerű használatával érhetjük el. A célunk ezzel az advice-szal, hogy minden metódushívás után ellenőrizzük a **Time** osztály invariánsát, és kivételt váltunk ki, ha az osztályinvariáns sérül. A **returning** záradék az **after** advice-ban azt fejezi ki, hogy csak a rendben véget ért hívások után szeretnénk az advice végrehajtását, ha a join point kivételt váltott ki, akkor nem.

A metódushívás join pointok esetében a **target** pointcut kötheti névhez azt az objektumot, amire a metódust meghívták; vegyük észre a hasonlóságot az **execution** join point esetén használt **this** pointcuttal. A **call** esetében is használható a **this** pointcut, de az **call** esetben a metódushívást kezdeményező objektumot azonosítja, nem azt, amire a metódust meghívták.

Az advice paraméterlistájában a híváshoz használt objektumot `Time` típusuként deklaráltuk, ez szűkíti le a vizsgált metódusokat a `Time` osztály metódusaira. A `within` pointcuttal a vizsgált join pointokat szűrhetjük. Az itt használt tagadó alakú `!within` segítségével kizárjuk a megnevezett join pointok közül azokat, amelyek a `Contracts` aspektusban vannak. Ez azért fontos, mert ezzel az aspektussal nem szeretnénk szőni az ugyanebben az aspektusban bekövetkező eseményekre (nehogy a szövés végtelen rekurziót eredményezzen).

Nézzük most meg, hogyan módosul a `Main` osztály `main` metódusa (1. kódlista) a szövés hatására. Hasonlítsuk össze a 2. és a 6. kódlistát!

```
public static void main(java.lang.String[]);
Code:
  0: new #17 // class Time
  3: dup
  4: invokespecial #19 // Method Time.<init>:()V
  7: astore_1
  8: aload_1
  9: sipush 137
 12: istore_2
 13: astore_3
 14: aload_3
 15: iload_2
 16: invokevirtual #20 // Method Time.advance:(I)V
 19: invokestatic #47 // Method Contracts.aspectOf:()LContracts;
 22: aload_3
 23: invokevirtual #51 // Method
    Contracts.ajc$afterReturning$Contracts$2$aa874fa4:(LTime;)V
 26: nop
 27: getstatic #24 // Field java/lang/System.out:Ljava/io/PrintStream;
 30: aload_1
 31: invokevirtual #30 // Method
    java/io/PrintStream.println:(Ljava/lang/Object;)V
 34: return
```

6. kódlista. Megszótt `Main.main` metódus

A `Time.advance` metódus hívása után a szótt változatban egy osztályszintű és egy példánymetódus is meghívódik, mindkettő az aspektus kódját tartalmazó `Contracts` típusdefinícióból. Vessünk egy pillantást az utóbbira, a 7. kódlistán. Világosan látható benne az osztályinvariáns ellenőrzésének és az `AssertionError` kivétel kiváltásának logikája.

```
public void ajc$afterReturning$Contracts$2$aa874fa4(Time);
```

Code:

```
0: aload_1
1: invokevirtual #44 // Method Time.getMin:()I
4: istore_2
5: iload_2
6: iflt 15
9: iload_2
10: bipush 60
12: if_icmplt 25
15: new #50 // class java/lang/AssertionError
18: dup
19: ldc #88 // String Time: class invariant violated!
21: invokespecial #54 // Method
    java/lang/AssertionError."<init>":(Ljava/lang/Object;)V
24: athrow
25: return
```

7. kódlista. Az **after** advice bájtkódja

Hasonlóan érdekes az is, hogy a szövés hatására hogyan változik meg a `Time` osztály bájtkódja, hogyan kerül be az **advance** metódusba az **around** advice által definiált programlogika. Ennek a felderítését azonban most az Olvasóra hagyjuk.

Végezetül megemlíjtjük, hogy az AspectJ többféle módon is képes szőni. A szövés valójában nem a forráskódok szintjén történik, hanem a forráskódból előállított bájtkód transzformálásával. Az **ajc** aspektusszövő Java és AspectJ forráskódok lefordítására és összeszövésére is használható, de arra is, hogy (Javából, Scalából vagy más nyelvből) előállított bájtkódot módosítson az aspektusok kódja alapján. Egy másik lehetőség az, hogy a bájtkódokat ne fordítási időben módosítsuk, hanem akkor, amikor futáskor betöltődnek a Java virtuális gépbe. A betöltési idejű szövés az AspectJ **aj** parancsával történhet.

3. A Scala nyelv szövése

Az előző szakaszokban példát mutattunk arra, hogy a Java hogyan képződik le bájtkódra, és az AspectJ ezen a bájtkódon milyen módosításokat hajt végre. Ebben a szakaszban azt vizsgáljuk, hogy a Scala forrásból milyen bájtkód keletkezik. Ezzel párhuzamosan azt is megvizsgáljuk, hogy ennek milyen következményei vannak a szövés szempontjából.

A Scala nyelv egy multiparadigmás programozási nyelv, az objektum-orientált nyelvi elemek mellett elsősorban a funkcionális nyelvek vonásai találhatók meg benne. A következőekben megvizsgáljuk a nyelvi elemek egy részét – a teljes nyelv vizsgálata túlmutatna a cikk keretein.

Először is nézzünk meg egy egyszerű példát. Objektum-orientált értelemben vett osztályokat a Scala nyelvben is definiálhatunk. Készítsük el a `Time` osztályt Scala nyelven (8. kódlista), és hasonlítsuk össze a bájtkódját a Java nyelvű definíció bájtkódjával.

```
class Time {  
  private var hour: Int = 0  
  private var min: Int = 0  
  def getHour = hour  
  def getMin = min  
  def advance( minutes: Int ){  
    min += minutes  
    hour += min/60  
    min %= 60  
  }  
  override def toString: String = f"$hour%02d:$min%02d"  
}
```

8. kódlista. A Time osztály Scalában

Számos apró különbség található a Java definíció bájtkódjához képest (lásd az 1. ábrát), de a Java nyelven írt `Main` osztály a Scala definícióval is elboldogul. Nem meglepő, hogy a korábban már látott `Contracts` aspektus gond nélkül rászótható erre a Scala típusra is, és ugyanúgy működik, mint a Java verzió esetében.

Azokban az esetekben, amikor egy Scala definíció hasonlít a megfelelő Java definícióra, az AspectJ szövével történő transzformálás működik, és a kívánt eredményt adja. Emiatt számos esetben használhatunk AspectJ aspektusokat Scala kód szövéseire. Sok példát, amelyeket jellemzően jól lehet aspektusokkal kezelni, Scala esetében is megírhatunk. Fejlesztési idejű aspektusok nyomonkövetésre és profilozásra, esetleg szerződések ellenőrzésére – bár ez utóbbi már nem csak fejlesztési időben, hanem éles kódban is hasznos lehet – gyakran jól rászótható Scala kódjainkra (például Jan Machecek [5] egy példát mutat arra, hogy hogyan lehet a Scala nyelv Akka keretrendszerében megírt program üzenetküldéseit AspectJ segítségével megfigyelni). Azokban az esetekben, amikor a Scala olyan nyelvi elemeit használjuk, amelyek nem feleltethetők meg egyértelműen Java kódnak, már sokkal nehezebb helyzetben vagyunk. Éles kódokban

<code>public void advance(int);</code>	<code>public void advance(int);</code>
Code:	Code:
0: aload_0	0: aload_0
1: dup	1: aload_0
2: getfield #3	2: invokespecial #28
5: iload_1	5: iload_1
6: iadd	6: iadd
7: putfield #3	7: invokespecial #31
10: aload_0	10: aload_0
11: dup	11: aload_0
12: getfield #2	12: invokespecial #25
15: aload_0	15: aload_0
16: getfield #3	16: invokespecial #28
19: bipush 60	19: bipush 60
21: idiv	21: idiv
22: iadd	22: iadd
23: putfield #2	23: invokespecial #33
26: aload_0	26: aload_0
27: dup	27: aload_0
28: getfield #3	28: invokespecial #28
31: bipush 60	31: bipush 60
33: irem	33: irem
34: putfield #3	34: invokespecial #31
37: return	37: return

9. kódlista. Javából

10. kódlista. Scalából

1. ábra. A `Time.advance` metódus Java és Scala implementációjának bájtkódja

futó aspektusokat (production aspects) mindezért igen körültekintően kell használni. Naplózásra, tranzakciókezelésre, jogosultságkezelésre előszeretettel használunk aspektusokat, és sok eddig megírt Java keretrendszer használ aspektus-orientált eszközöket (például a JavaEE platform és a Spring is). Amennyiben ezeket a keretrendszereket szeretnénk Scala nyelvből használni, meg kell vizsgálnunk, hogy ezekkel az eszközökkel hogyan működik együtt a Scalából generált bájtkód. A következőkben felvillantunk néhány olyan példát, amelyben az AspectJ használata odafigyelést igényel.

3.1. Osztály, egyke- és társobjektum

A Scala nyelv egyik érdekessége, hogy elkerüli a statikus metódus fogalmát. Statikus metódusok helyett a Scalában egyke objektumokat használunk. Ezeket az egyke objektumokat a `class` helyett az `object` kulcsszó vezeti be, és mindenben hasonlítanak az osztályra, kivéve, hogy mindössze egy példány lehet belőlük. Egyke objektumra példát a 11. kódlista szolgáltat.

```
object Main {  
  def main(args: Array[String]) {  
    val t = new Time  
    t.advance(137)  
    println(t)  
  }  
}
```

11. kódlista. A `Main` objektum

A fordítással kapcsolatos érdekességek azonnal szembeötlenek: az `object` definíció nem egy, hanem két osztályra fordult le, abban az értelemben, hogy a `Main` mellett egy `Main$` nevű `class` fájl is elkészült. Vizsgáljuk meg a generált bájtódban a `Main` osztály `main` nevű statikus metódusát a 12. kódlistában. A metódus úgy működik, hogy először elkéri a `Main$` osztály `MODULE$` mezőjét, majd ezen a `Main$` típusú objektumon meghívja a példányszintű `main` metódust.

```
public static void main(java.lang.String[]);  
Code:  
  stack=2, locals=1, args_size=1  
    0: getstatic #16 // Field Main$.MODULE$:LMain$;  
    3: aload_0  
    4: invokevirtual #18 // Method Main$.main:([Ljava/lang/String;)V  
    7: return
```

12. kódlista. A `Main.main` metódus bájtódbja

A `Main$` osztály bájtódbjából néhány érdekes gondolatot emel ki a 13. kódlista. A privát konstruktort a statikus inicializátor blokk hívja meg, hogy létrehozza az osztály egyetlen példányát, melyet a `MODULE$` mezőben tárol el. (Védelmet biztosít az is, hogy ez a mező véglegesített

```
public final class Main$  
  public static final Main$ MODULE$;  
  public void main(java.lang.String[]);  
  private Main$();  
  public static {};  
  Code:  
    0: new #2 // class Main$  
    3: invokespecial #12 // Method "<init>":()V  
    6: return
```

13. kódlista. Részletek a `Main$` osztály bájtkódjából

értékű, azaz `final`, és a `Main$` osztályból nem lehet származtatni, azaz `final`.) A `Main$.main` metódusban van a bájtkódja a 11. kódlistánban látott `main` logikának.

A Scala egyke objektum speciális esete a társobjektum (companion object), amikor egy osztály és egy objektum neve megegyezik. Erre mutat példát a 14. kódlista, melyben a `Time` osztály társobjektumát mutatjuk be. Egy osztály és a társobjektuma látják egymás privát mezőit, de az osztály és az objektum ezen kívül teljesen független, a típusuk is eltér. Az osztály bájtkódja az osztály metódusai mellett a társobjektum metódusaival megegyező nevű statikus metódusokat is tartalmazza, amelyek az előbb látott módon delegálják a hívásokat a társobjektum megfelelő mezőinek.

```
object Time {  
  def add(t1 : Time, t2: Time): Time = {  
    val t = new Time  
    t.advance(t1.hour*60+t1.min)  
    t.advance(t2.hour*60+t2.min)  
    t  
  }  
}
```

14. kódlista. A `Time` objektum

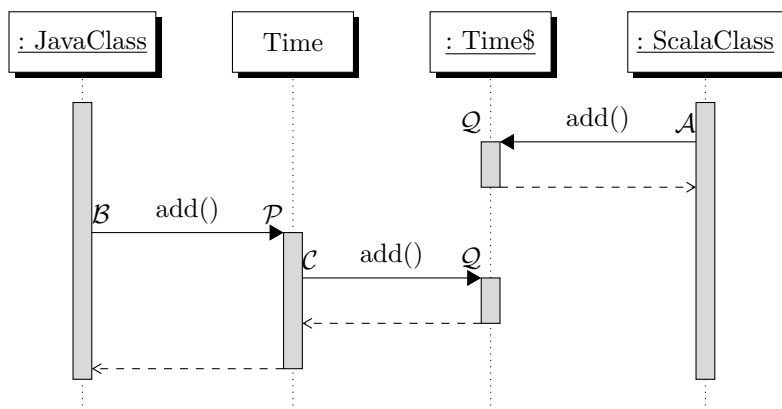
Az érdekesség ebben az, hogy a statikus metódusok feladata lényegében csak a Javával való kompatibilitás fenntartása. Ezt könnyen ellenőrizhetjük: ha a társobjektum metódusát meghívjuk Scala kódból,

```

public static final Time add(Time, Time);
  flags: ACC_PUBLIC, ACC_STATIC, ACC_FINAL
  Code:
    stack=3, locals=2, args_size=2
      0: getstatic #11 // Field Time$.MODULE$:LTime$;
      3: aload_0
      4: aload_1
      5: invokevirtual #13 // Method Time$.add:(LTime;LTime;)LTime;
      8: areturn

```

15. kódlista. Statikus metódus a `Time.class`-ban



2. ábra. Egy object metódusának meghívása

a hívás egyből a `Time$` példánymetódusát hívja meg, a statikus metódus megkerülésével.

Ezeket a lehetséges hívásokat a 2. ábra részletezi. Tegyük fel, hogy egy `before` advice-t szeretnénk az `execution add` pontcutra szőni. Az lenne jó, ha a szövése a Javából és Scalából vegyesen álló kódokra is jól működne. Az egyke objektum metódusa Javában a statikus metódusra képződik le, tehát egy ennek megfelelően megírt AspectJ kód is erre szőződik ($\mathcal{P}$ pont). A helyes megoldás ebben az esetben a statikus metódus helyett a két ($\mathcal{Q}$) pontra történő szövése: vessük össze az alábbi két pointcut definíciót:

```

P  execution(static Time Time.add(..))
Q  execution(Time Time$.add(..))

```

Hasonló esetnek tűnik a `call add` pointcut is, azonban itt sokkal súlyosabb problémákba ütközünk. Felületesen azt gondolhatjuk, hogy az osztály nevéhez egy dollár jelet adva az $\mathcal{A}$ és $\mathcal{C}$ joinpointokat lefedtük, azaz tulajdonképpen készen is vagyunk. A megoldás azonban nem ilyen egyszerű.

```

A,C  call(* Time$.add(..))
B    call(* Time.add(..))

```

Tegyük fel, hogy a programunk több jar fájlból tevődik össze, az egyik – nevezzük `time.jar` fájlnak – tartalmazza a `Time` osztály és társobjektuma kódjait, a másikat pedig ennek felhasználásával fordítjuk, azaz az említett jar fájl a classpathban van lefordítva. Ekkor a $\mathcal{C}$ joinpontot a `time.jar` fájl tartalmazza, tehát egy új, szőtt jar fájlt kell készíteni.

Még nagyobb probléma, ha az aspektus megpróbálná a szövés helyét felhasználni, erre a legegyszerűbb eset egy `this(o)` használata. Ahhoz, hogy az `o` változó helyes értéket vegyen fel, a kódot az $\mathcal{A}$ és a $\mathcal{B}$ pontokra kell szűnünk. Ekkor viszont ki kell zárnunk a $\mathcal{C}$ pontot, például:

```

before(): call(Time Time.add(..))
        || (call(Time Time$.add(..)) && !withincode(Time
            Time.add(..)))

```

Ez viszont a pointcut kódját teszi bonyolulttá, tehát érdemes lehet egy futási idejű megoldást is adni, hogy kizárjuk azt, hogy egy hívás alatt az advice kétszer fusson le.

```

String ec = thisEnclosingJoinPointStaticPart
            .getSignature().getDeclaringTypeName();
String th = thisJoinPointStaticPart
            .getSignature().getDeclaringTypeName();
if (th.equals(ec + "$"))
    // do nothing

```

Vegyük észre, hogy ebben az esetben úgy tűnik, hogy nincs tökéletes megoldás, ha figyelembe vesszük, hogy a közbeékelődött osztály miatt vagy a hívó, vagy a hívott eltér a Java és a Scala esetében. Ez akkor

lehet gond, hogyha az advice törzsében a `target` és a `thisJoinPoint` is felhasználásra kerül. Ennek azonban szerencsére egyke objektum esetében nincs értelme, hiszen egy statikus metódust hívunk, tehát a `target` nem létezik.

3.2. Függvények

A Scala nyelv egyik erénye a funkcionális paradigma támogatása. Tisztességtelen lenne, ha pont a funkcionális eszközöket nem vizsgálnánk.

```
object Fun extends App {  
  def app2(f : Int => Int, x : Int) : Int = f(f(x))  
  val v = app2(_ : Int=>Int, 2)  
  val w = v((x:Int) => x + 20)  
  
  println(w)  
}
```

A fenti kódban az `app2` egy magasabb rendű függvény, amelynek az első argumentuma egy függvény, és ezt kétszer alkalmazza a másodikra. A `v` ennek egy részleges alkalmazása, egy olyan függvény, amely a paraméterként kapott függvényt kétszer alkalmazza a 2 konstansra. A `w` függvény elkészítéséhez egy lambda függvényt használtunk, a `v` függvénynek átadtunk egy olyan névtelen függvényt, amely 20-at ad a paraméteréhez.

Mivel az így készült `Fun` egy egyke objektum, tanulságos megnézni, hogy a függvények milyen Java konstrukcióra fordulnak le (16. kódlista). Nem meglepő, hogy minden függvényt egy osztály ábrázol, például a `Function1` nevű trait az, amely példányai az egy paraméteres függvényeket reprezentálják. Egész pontosan N paraméter esetén az objektum típusa `FunctionN` lesz, és az osztály $N + 1$ generikus paramétere megadja a paraméterek, valamint a visszatérési érték típusát. Látható, hogy ezzel a magasabb rendű függvények is könnyedén leírhatóak.

A Scala a függvényhívást magát úgy kivitelezi, hogy minden a függvényhívás valójában az `apply` nevű metódus meghívására fordul le. Ennél bonyolultabb eset a `v()` függvény definíciójában az `app()` függvény részleges alkalmazása. Valójában a részleges alkalmazás csak egy szintaktikus cukormáz, és ugyanezt egy névtelen függvénnyel is leírhatjuk, a két megoldás pontosan ugyanazt csinálná. A névtelen

```
public static final int w();
public static final scala.Function1<scala.Function1<java.lang.Object,
    java.lang.Object>, java.lang.Object> v();
public static final int app2(scala.Function1<java.lang.Object,
    java.lang.Object>, int);
```

16. kódlista. A `Fun` osztály függvényei

függvények pedig – nem meglepő módon – a `Function` osztályokból származó névtelen osztályok:

```
public final Function1<Function1<Integer,Integer>, Integer> v =
    new Function1<Function1<Integer,Integer>, Integer> {
        int apply(Function1<Integer, Integer> f) {
            return app2(f, 2);
        }
    }
}
```

Az eddigi eredmények ugyan biztatóak, sajnos azonban a helyzet bonyolultabb. Ebben a példában is látható ugyanis, hogy mivel a Java nyelvben primitív típus nem szerepelhet típusparaméterként, az `int` típus is `Integer` típusként szerepel, a kettő közötti konverziót pedig a nyelv autoboxing mechanizmusa végzi. Egy bonyolult függvényláncnál a ki- és becsomagolás már olyan teljesítménycsökkenést okoznak, amely a Scalát, mint funkcionális nyelvet használhatatlanná tenné. Ennek elkerülésére a Scala a nulla, egy és kétparaméteres függvényekhez definiálja az `apply` metódusokat minden primitív típusra is.

A fenti példában a `w` függvény definíciójában a `v` függvénynek átadunk egy olyan névtelen függvényt, amely a paraméteréhez 20-at ad. A `Function1` absztrakt metódusát implementálja az `apply` nevű metódus, amely objektumokat vár, azzal tér vissza, és elvégzi a primitív típusok ki- és becsomagolását, valamint a típuskonverziókat, és meghívja a másik `apply` nevű metódust, amely viszont már `int` típust vár, és azzal is tér vissza. Ez a metódus pedig meghívja a `apply$mcII$sp` nevűt, amely a tényleges függvénytörzset tartalmazza.

Ennek oka egyrészt a Javával való kompatibilitás biztosítása, másrészt az, hogy a Scala kódon belül a felesleges konverziók elkerülhetőek legyenek, tehát Scala kódon belül használva a fenti függvényt, közvetlenül az `apply$mcII$sp` nevű metódus hívódik meg. A függvény

névében a két nagy I betű a paraméter és visszatérési érték típusát adja meg, tehát minden két primitív típusra létezik egy külön metódus (pontosabban a `void` és a `boolean` csak visszatérési érték lehet), így a `Function0`-ban 6, a `Function1`-ben 24, míg a `Function2`-ben 96 ilyen metódus van. Amennyiben azonban a függvénynek van nem primitív típusú paramétere, ezek a speciális metódusok nem fognak részt venni a függvény kiértékelésében.

Ezek szövése hasonló problémákat vet fel, mint az objektumok esete. Az `execution` esete általában egyszerű, elég a megfelelő `apply` metódusra szöni, a típusok függvényében. A `call` típusú pointcut szövésénél ügyelni kell arra, hogy minden `apply` kezdetű függvényre szőjünk, miközben az osztályon belüli hívásokat érintetlenül hagyjuk.

3.3. Tail-call

Vegyük a következő rendkívül ártatlannak tűnő Scala osztályt, és vizsgáljuk meg a metódusból generált bájtkódot.

```
class Recursion {  
  final def factorial(n:Int, acc:Int) : Int = {  
    if (n==0) acc  
    else factorial(n-1, acc*n)  
  }  
}
```

17. kódlista. Faktoriális függvény

Meglepődve vesszük észre, hogy a bájtkódból (18. kódlista) hiányzik a rekurzív hívás! Valóban, a Scala nyelv képes a funkcionális nyelvekre jellemző ún. tail-call optimalizációra, amely a végrekurziókat képes ciklussá átalakítani. Szerencsére ez az optimalizáció viszonylag ritkán lehetséges, hiszen nem elég, hogy a függvénynek végrekurzívnak kell lennie, de vagy `private`-nak, vagy `final`nak is lennie kell: ellenkező esetben ugyanis a függvény felüldefiniálható, így a dinamikus kötés miatt nem biztos, hogy az eredeti kódban szereplő függvényhívás önmagát hivatkozza, így a hívás nem is optimalizálható ki.

Mindenesetre ez a példa mutatja, hogy teljes nyelvi támogatás nélkül a helyes szöves problémája csak részlegesen oldható meg. Elképzelhető az AspectJ egy olyan bővítése, amely észreveszi, hogy a bájtkód utolsó eleme egy `goto 0`, és ez alapján szó a tail-callra is.

```
0: iload_1
1: iconst_0
2: if_icmpne 7
5: iload_2
6: ireturn
7: iload_1
8: iconst_1
9: isub
10: iload_2
11: iload_1
12: imul
13: istore_2
14: istore_1
15: goto 0
```

18. kódlista. A faktoriális bájtkódja

4. Összefoglalás

Az objektumorientált nyelvek aspektusorientált kiterjesztései után nem sokkal megjelentek az első ötletek a funkcionális nyelvek kiterjesztésére. Ezek közül is elsőként az ML nyelvcsalád (MinAML [9], PolyAML [1] és Aspectual Caml [6]) kiterjesztései jöttek létre. Tisztán funkcionális nyelvek közül a Haskell esetében vizsgálták az aspektusorientáltságot, ennek keretében sikerült kimutatni összefüggéseket az aspektusorientáltság és a típusosztályok között [8].

Multiparadigmás nyelvekben is próbálkoztak már aspektusorientált kiterjesztések készítésével. Egy blogban megjelent ötlet [2] úgy készíti a Scala nyelvben aspektusorientáltságot, hogy a szövendő objektumokat egy gyárral állítják elő, amely a Java nyelv dinamikus proxyjával tér vissza, erre a dinamikus proxyra pedig a Scala nyelv mixinjeit szövik. A pointcutokat az AspectJ library segítségével készítik el, és futási időben ellenőrzik.

A [7]-ban a szerzők egy DSLt definiálnak, és metódusszintű proxyk segítségével érnek el aspektus-orientáltságot. Azaz, míg az előző cikkben az objektumokat készítik elő szövéssre, és dinamikus proxyval hozzák őket létre, ebben az esetben a metódusokat kell úgy megírni, hogy az aspektus-orientált mixint minden metódusnál meghívják.

Ez a két utóbbi cikk a Scala nyelv szövéssével foglalkozik, és

különböző technikákat adnak a szövekre, amelyek nem igényelnek külső fordítót. Mi a cikkünkben egy meglévő külső fordító felhasználhatóságát vizsgáltuk, az említett cikkekkel szemben elsősorban statikus idejű szövéssre. A második cikk a Scala nyelvet használja fel a szövéssre, így az eredmény nem okoz meglepetéseket, viszont az osztályokat fel kell készíteni a szövéssre, ami a forráskód módosítását jelenti. Az első cikk Java konstrukciót használ fel, és nem tér ki az emiatt jelentkező problémákra, így – többek között – a végrekurziót sem kezeli.

Az eddigiek során láthattuk, hogy a Scala nyelv aspektus-orientált használata nem lehetetlen, sőt a vizsgált AspectJ nyelv erre alkalmasnak is bizonyult sok esetben. Mindazonáltal meg kell ismételtnünk, hogy az AspectJ a Java nyelvhez készült, és nem minden esetben képes a Scala kód megfelelő szövéssre. Az egyik példánkban a végrekurzív függvények optimalizációja okoz gondot, mivel a függvényhívás helyett ilyenkor a kódba egy ugrás kerül. Az AspectJ ugyan nem képes erre szőni, de az optimalizáció csak elég speciális esetben történhet meg, ahol a metódus nem felüldefiniálható.

Emellett megvizsgáltuk az egyke objektumokból fordított Java bájtkódok szövéssét is; ebben az esetben azt figyeltük meg, hogy az így készült bájtkód osztályokat Scalából és Javából különbözőképpen kell elérnünk. Figyelembe kell továbbá vennünk, hogy az elkészült wrapper osztályra nem szerencsés szőnünk, mert az nem a kívánt eredményre vezet. Függvényeknél hasonló jelenséggel szembesültünk; itt a problémát az okozza, hogy az `apply` függvény specializálva van a primitív típusok összes lehetséges kombinációjára, és az általánosabb metódusok egyre specializáltabbakat hívnak. Itt is ügyelni kell arra, hogy a lehetséges függvényhívási láncokban mindig csak egy helyre szőjünk.

Hivatkozások

- [1] D. S. Dantas, D. Walker, G. Washburn, S. Weirich, Poly aml: a polymorphic aspect-oriented functional programming language, *ACM SIGPLAN Notices*, 40(9) (2005), pp. 306–319.
- [2] B. Jonas, Real-world scala: Managing cross-cutting concerns using mixin composition and aop, <http://jonasboner.com/2008/12/09/real-world-scala->

managing-cross-cutting-concerns-using-mixin-composition-and-aop/.

- [3] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An overview of aspectj, *ECOOP 2001 – Object-Oriented Programming, LNCS 2072*, Springer, pp. 327–354.
- [4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, J. Irwin, Aspect-oriented programming, *ECOOP'97 – Object-Oriented Programming, LNCS 1241*, Springer, pp. 220–242.
- [5] J. Machacek, Aspectj with akka, scala,
<http://www.cakesolutions.net/teamblogs/2013/08/07/aspectj-with-akka-scala/>
- [6] H. Masuhara, H. Tatsuzawa, A. Yonezawa, Aspectual caml: an aspect-oriented functional language, *ACM SIGPLAN Notices*, 9 (2005), pp. 320–330.
- [7] D. Spiewak, T. Zhao, Method proxy-based aop in scala, *Journal of Object Technology*, 8(7) (2009), pp. 149–169.
- [8] M. Sulzmann, M. Wang, Aspect-oriented programming with type classes, *Proceedings of the 6th workshop on Foundations of aspect-oriented languages*, 2007, pp. 65–74.
- [9] D. Walker, S. Zdancewic, J. Ligatti, A theory of aspects, *ACM SIGPLAN Notices*, 9 (2003), pp. 127–139.

Mély neuronhálók a magyar nyelvű beszédfelismerésben

Grósz Tamás*, Tóth László**

*Eötvös Loránd Kollégium, Informatika Műhely
Szegedi Tudományegyetem

**MTA-SZTE Mesterséges Intelligencia Kutatócsoport
Magyar Tudományos Akadémia, Szegedi Tudományegyetem

{groszt, tothl}@inf.u-szeged.hu

2006-os megjelenésük óta egyre nagyobb népszerűségnek örvendenek az akusztikus modellezésben az ún. mély neuronhálók. A hagyományos neuronhálókkal ellentétben a mély hálók sok rejtett réteget tartalmaznak, emiatt a hagyományos módszerekkel tanítva őket nem lehet igazán jó eredményeket elérni. Cikkünkben röviden bemutatunk négy új tanítási módszert a mély neuronhálókhoz, majd a mély neuronhálókra épülő akusztikus modelleket beszédfelismerési kísérletekben értékeljük ki¹.

1. Bevezetés

Az elmúlt néhány évtizedben a mesterséges neuronhálók számos változatát kipróbálták a beszédfelismerésben - annak függvényében, hogy éppen mi volt az aktuálisan felkapott technológia. Általános elismertséget azonban csak a többrétegű perceptron-hálózatokra (MLP) épülő ún. hibrid HMM/ANN modellnek sikerült elérnie, főleg a Bourlard-Morgan páros munkásságának köszönhetően [2]. Bár kisebb felismerési

¹Jelen mű a 2014-es Magyar Számítógépes Nyelvészeti Konferencián publikált cikkünk [1] kibővített változata

feladatokon a neuronhálós modellek jobb eredményt adnak, mint a sztenderd rejtett Markov-modell (HMM), alkalmazásuk mégsem terjedt el általánosan, részben mivel technikailag nehezebb a használatuk, másrészt mivel nagyobb adatbázisokon az előnyük elvész, köszönhetően a HMM-ekhez kifejlesztett trifón modellezési és diszkriminatív tanítási technikáknak. Így a hibrid modell az elmúlt húsz évben megmaradt a versenyképes, de igazi áttörést nem hozó alternatíva státuszában.

Mindez megváltozni látszik azonban az ún. mély neuronhálók (deep neural nets) megjelenésével. A mély neuronhálót (pontosabban tanítási algoritmusát) 2006-ban publikálták először [3], és a kezdeti cikkek képi alakfelismerési tesztekhez használták demonstrációként. Legjobb tudomásunk szerint a mély hálók első beszédfelismerési alkalmazása Mohamed 2009-es cikke [4] volt – mely cikkben rögtön sikerült megdönteni a népszerű TIMIT benchmark-adatbázison elért összes korábbi felismerési pontosságot.

2013-as cikkünkben [5] mi is bemutattuk az eredeti módszer alapötletét, és publikáltuk az első mély neuronhálós felismerési eredményeket magyar nyelvű adatbázisokon. A technológia iránti érdeklődés azóta sem csökkent, példának okáért az MIT „Tech Review’s” listája a mély neuronhálókat beválogatta a 2013-as év 10 legfontosabb technológiai áttörést jelentő módszere közé. Mindeközben sorra jelennek meg az újfajta mély hálózati struktúrákat vagy tanítási módszereket publikáló cikkek. Jelen anyagunkban néhány olyan új ötletet mutatunk be, amelyekkel az eredeti tanítási algoritmus eredményei még tovább javíthatók.

A mély neuronhálók hatékony betanításához az eredeti szerzők az ún. DBN előtanítási módszert javasolták [3], ami egy elég komplex és műveletigényes algoritmus. A módszer az ún. korlátos Boltzmann-gépeken (Restricted Boltzmann Machine, RBM) és azok tanító algoritmusán, a CD-algoritmuson (kontrasztív divergencia) [3] alapszik. A korlátos Boltzmann-gép lényegében a neuronháló egy rétegpárjának felel meg, így a betanítás rétegenként haladva történik felügyelet nélkül módon.

A DBN előtanítási módszer alternatívájaként vetették fel nemrég az ún. discriminative pre-training („diszkriminatív előtanítás”) algoritmust [6]. Ezen módszer esetén az előtanítás felügyelt módon történik: kezdetben egy hagyományos (egy rejtett réteges) hálóból indulunk ki, néhány iteráción keresztül tanítjuk, ezután egy új rejtett réteget illesztünk be a kimeneti réteg alá és a kimeneti réteget újrainicializáljuk. Az így kapott neuronhálót újra tanítjuk néhány iteráción keresztül, az

új rétegek hozzáadását pedig addig ismételjük, amíg a rejtett rétegek száma el nem éri a kívánt mennyiséget. A módszer előnye, hogy a tanítás során - mindkét fázisban - csak a backpropagation algoritmust kell használnunk.

Egy másik mostanában javasolt, előtanítást nem igénylő módszer az ún. rectified („egyenirányított”) neuronok használata. Ezek nevüket onnan kapták, hogy egyenletükben a szokásos szigmoid aktivációs függvény le van cserélve egy olyan komponensre, amelynek működése egy egyenirányító áramkörre hasonlít (matematikailag a $\max(0, x)$ függvényt valósítja meg). A rectifier neuronokra épülő mély neuronhálók használatát eredetileg képfeldolgozásban vetették fel, csoportunk az elsők között próbálta ki őket beszédfelismerésben [7]. Eredményeink egybevágnak a más kutatók által velünk párhuzamosan publikált eredményekkel: úgy tűnik, hogy az egyenirányított mély neuronhálók hasonló vagy kicsit jobb felismerési pontosságot tudnak elérni, mint hagyományos társaik, viszont a betanításuk jóval egyszerűbb és gyorsabb [8, 9].

Egy negyedik nemrégén publikált módszer a neuronháló backpropagation tanítási algoritmusát módosítja. Az ún. dropout („kiejtéses”) tanulás lényege, hogy a neuronháló tanítása során minden egyes tanítópélda bevitelekor véletlenszerűen kinullázzuk („kiejtjük”) a hálót alkotó neuronok kimenetének valahány (általában 10-50) százalékát [10]. Ennek az a hatása, hogy az azonos rétegbe eső neuronok kevésbé tudnak egymásra hagyatkozni, így a probléma önálló megoldására vannak kényszerítve. Ennek köszönhetően lényegesen csökken a túltanulás veszélye. A módszert eredetileg javasló cikkben kiugró, 10-20 százaléknyi relatív hibacsökkenéseket értek el képi alakfelismerési és beszédfelismerési feladatokon.

A javasolt módszerek hatékonyságát először az angol nyelvű TIMIT adatbázison szemléltetjük, mivel ezen számtalan összehasonlító eredmény áll rendelkezésre. Ezután két magyar nyelvű adatbázissal kísérletezünk. Az egyik egy híradós adatbázis, amelynek méretét tavaly óta jelentősen sikerült megnövelnünk. A másik pedig a „Szindbád történetei” című hangoskönyv hangzóanyaga.

2. Mély neuronhálók

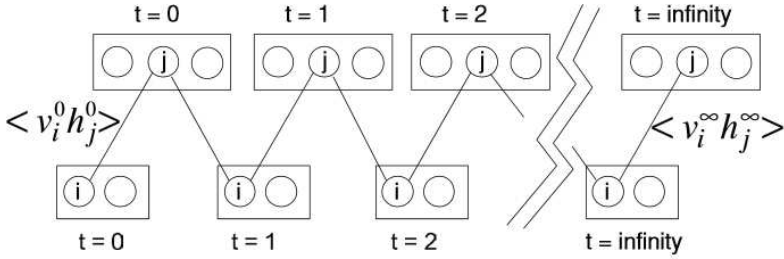
A hagyományos neuronhálók és a mély hálók között az alapvető különbség, hogy utóbbiak több (általában 3-nál több) rejtett réteggel rendelkeznek. Ezen mély struktúrájú neuronhálók használatát igazolják a legújabb matematika érvek és empirikus kísérletek, melyek szerint adott neuronszám mellett a több rejtett réteg hatékonyabb reprezentációt tesz lehetővé. Ez indokolja tehát a sok, relatíve kisebb rejtett réteg alkalmazását egyetlen, rengeteg neuront tartalmazó réteg helyett.

A sok rejtett réteges mély neuronhálók tanítása során több olyan probléma is fellép, amelyek a hagyományos egy rejtett réteges hálók esetén nem vagy alig megfigyelhetők, és ezen problémák miatt a betanításuk rendkívül nehéz. A hagyományos neuronhálók tanítására általában az ún. backpropagation algoritmust szokás használni, ami tulajdonképpen a legegyszerűbb, gradiensalapú optimalizálási algoritmus neuronhálókhoz igazított változata. Több rejtett réteg esetén azonban ez az algoritmus nem hatékony. Ennek egyik oka, hogy egyre mélyebbre hatolva a gradiensek egyre kisebbek, egyre inkább „eltűnnek” (ún. „vanishing gradient” effektus), ezért az alsóbb rétegek nem fognak kellőképp tanulni [11]. Egy másik ok az ún. „explaining away” hatás, amely megnehezíti annak megtanulását, hogy melyik rejtett neuronnak mely jelenségekre kellene reagálnia [3]. Ezen problémák kiküszöbölésére találták ki az alább bemutatásra kerülő módszereket.

2.1. DBN előtanítás

A mély neuronhálók legelső tanítási módszerét 2006-ban publikálták [3], lényegében ez volt az a módszer, amely elindította a mély neuronhálók kutatását. A módszer lényege, hogy a tanítás két lépésben történik: egy felügyelet nélküli előtanítást egy felügyelt finomhangolási lépés követ. A felügyelt tanításhoz használhatjuk a backpropagation algoritmust, az előtanításhoz azonban egy új módszer szükséges: a DBN előtanítás.

A DBN előtanítással egy ún. „mély belief” hálót (Deep Belief Network, DBN) tudunk tanítani, amely rétegei korlátos Boltzmann gépek (RBM). A korlátos Boltzmann gépek a hagyományosaktól annyiban térnek el, hogy a neuronjaik egy páros gráfot kell hogy formázzanak. A két réteg közül a látható rétegen keresztül adhatjuk



1. ábra. Gibbs mintavételezés

meg a bemenetet, a rejtett réteg feladata pedig az, hogy az inputnak egy jó reprezentációját tanulja meg.

Az RBM-ek tanításához a kontrasztív divergencia algoritmust (CD) használhatjuk, amely egy energiafüggvény alapú módszer. Egy RBM a következő energiát rendeli egy látható (v) és a rejtett réteg (h) állapotvektor-konfigurációhoz:

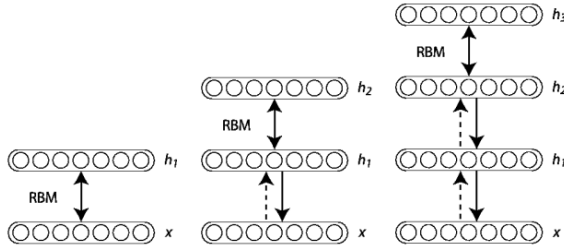
$$E(v, h; \Theta) = - \sum_{i=1}^V \sum_{j=1}^H w_{ij} v_i h_j - \sum_{i=1}^V b_i v_i - \sum_{j=1}^H a_j h_j. \quad (1)$$

A kontrasztív divergencia algoritmus (CD) esetén a következő update szabályt alkalmazzuk a látható-rejtett súlyokra:

$$\Delta w_{ij} \propto \langle v_i h_j \rangle_{input} - \langle v_i h_j \rangle_k, \quad (2)$$

ahol $\langle \cdot \rangle_k$ a látható és a rejtett rétegek Gibbs mintavételezővel k alkalommal történő mintavételezése utáni kovarianciája. Gibbs mintavételezés alatt a következőt kell érteni: miután az input alapján meghatároztuk a rejtett réteg állapotait, azok alapján mintavételezni tudjuk (a kapcsolatok súlyait felhasználva) a látható réteg állapotait. A mintavételezés után még szükséges, hogy az így kapott látható réteg alapján újra meghatározzuk a rejtett réteg neuronjainak állapotait. A rekonstrukciót tetszőleges alkalommal megismételhetjük az 1. ábrán látható módon.

Mivel a rekonstrukciós lépések rendkívül időigényesek, ezért általában csak k db rekonstrukciót végzünk. A CD mohó algoritmus a $k = 1$ rekonstrukciót végez, és az alapján tanulja a súlyokat, általánosan ez a módszer terjedt el viszonylag kis időigénye és jó teljesítménye miatt. A



2. ábra. Korlátozott Boltzmann-gép, illetve a belőle felépített DBN

mohó előtanítás során a súlyok frissítését a következő módon végezzük:

$$\Delta w_{ij} \propto \langle v_i h_j \rangle_{\text{input}} - \langle v_i h_j \rangle_{t=1}. \quad (3)$$

Habár az RBM energiafüggvénye rendkívül jól működik bináris neuronok esetén, beszédfelismerésben azonban valós bemeneteink vannak, ennek kezelésére szükséges az energiafüggvény (1) módosítása. A valós bemenetekkel rendelkező RBM-et Gaussian-Bernoulli korlátozott Boltzmann gépnek (GRBM) nevezzük, energiafüggvénye:

$$E(v, h | \Theta) = \sum_{i=1}^V \frac{(v_i - b_i)^2}{2} - \sum_{i=1}^V \sum_{j=1}^H w_{ij} v_i h_j - \sum_{j=1}^H a_j h_j. \quad (4)$$

Ezen új energiafüggvény esetén a CD-1 algoritmusban csupán a Gibbs mintavételezés módját kell módosítani, a súlyok frissítése pedig továbbra is (2) szerint történik.

A DBN előtanítás során a hálót rétegpáronként tanítjuk. Az első lépésben az inputot és a legelső rejtett réteget egy GRBM-nek tekintve a CD-1 algoritmussal tanítjuk. A továbbiakban a következő RBM-nek a látható rétege az előzőleg tanított RBM rejtett rétege lesz, az új rejtett rétege pedig a következő rejtett réteg a hálóban, ezt illusztrálja a 2. ábra.

Az előtanítás után a hálózatot átalakítjuk hagyományos neuronhálónak, ami egyszerűen csak a súlyok átvitelével, illetve egy softmax kimeneti réteg felhelyezésével történik. Innentől a háló teljesen szokványosan tanítható felügyelt módon a backpropagation-algoritmus segítségével.

2.2. Diszkriminatív előtanítás

A diszkriminatív előtanítást (Discriminative pre-training, DPT) a DBN előtanítás alternatívájaként javasolták [6]. Ahogy az elnevezésből

sejthető, ez a módszer is két fázisból áll, a különbség, hogy az előtanítást is felügyelt tanítással, a backpropagation algoritmussal valósítjuk meg. Az algoritmus kezdetben egy hagyományos egy rejtett réteges neuronhálóból indul ki, amit néhány iteráción keresztül tanítunk. A következő lépésben egy új rejtett réteget illesztünk be a kimenet és a legfelső rejtett réteg közé, a kimeneti réteg súlyait újrainicializáljuk, majd az egész hálót tanítjuk néhány iteráción keresztül. Mindezt addig ismételjük, amíg a rejtett rétegek száma a kívánt mennyiséget el nem éri. A módszer előnye, hogy nem igényel külön tanítási algoritmust.

A tanítás során felmerül egy fontos kérdés, mégpedig, hogy az előtanítás során meddig tanítunk. Az eredeti cikk [6] szerint az eredmények romlanak, ha minden előtanítási lépésben a teljes konvergenciáig tanítunk. Javasolt csak néhány iterációt tanítani - a szerzők 1 iterációt javasolnak - mi a 4 iterációnyi előtanítást találtuk a legeredményesebbnek, azonban megemlítjük, hogy ha a tanító adatbázis mérete megnő, akkor az 1 iterációnyi előtanítás is elegendőnek tűnik.

2.3. Rectifier neuronhálók

Tekintve, hogy az előző két előtanítási módszernek rendkívül nagy az időigénye, sok kutató olyan módszereket próbált kidolgozni, amelyek nem igényelnek előtanítást. Az egyik ilyen javaslat nem a tanítóalgoritmust módosítja, hanem a hálót felépítő neuronokat. Az ún. rectified („egyenirányított”) neuronok nevüket onnan kapták, hogy a szokásos szigmoid aktivációs függvény le van cserélve egy olyan komponensre, amelynek működése egy egyenirányító áramkörre hasonlít (matematikailag a $\max(0, x)$ függvényt valósítja meg). A rectifier neuronokra épülő mély neuronhálók használatát eredetileg képfeldolgozásban javasolták, csoportunk az elsők között próbálta ki őket beszédfelismerésben [7]. Eredményeink egybevágnak a más kutatók által velünk párhuzamosan publikált eredményekkel [8, 9]: úgy tűnik, hogy az egyenirányított mély neuronhálók előtanítás nélkül is hasonló vagy kicsit jobb felismerési pontosságot tudnak elérni, mint hagyományos társaik előtanítással.

A rectifier függvény két alapvető dologban tér el a szigmoid függvénytől: az első, hogy az aktivációs érték növekedésével a neuronok nem „telítődnek”, ennek köszönhetően nem jelentkezik az eltűnő gradiens effektus. A rectifier neuronok esetén emiatt egy másik probléma

jelentkezhet, mégpedig hogy a gradiens értékek „felrobbannak” (ún. „exploding gradient” effektus), azaz egyre nagyobb értékeket vesznek fel [11]. A probléma kiküszöbölése céljából a neuronok súlyait a tanítás során időről időre normalizálni szokták, mi a kettes norma szerint normalizáltunk. A másik fontos különbség, hogy negatív aktivációs értékekre 0 lesz a neuronok kimenete, aminek következtében a rejtett rétegeken belül csak a neuronoknak egy része lesz aktív adott input esetén. Ez utóbbi tulajdonságról az is gondolhatnánk, hogy megnehezíti a tanulást, hiszen megakadályozza a gradiens visszaterjesztését, azonban a kísérleti eredmények ezt nem támasztják alá. A kísérletek azt igazolták, hogy az inaktív neuronok nem okoznak problémát mindaddig, amíg a gradiens valamilyen úton visszaterjeszthető.

Összefoglalva: a rectifier hálók nagy előnye, hogy nem igényelnek előtanítást, és a hagyományos backpropagation algoritmussal gyorsan taníthatók.

2.4. Dropout módszer

Az ún. dropout („kiejtés”) tanulás lényege, hogy a neuronháló tanítása során minden egyes tanítópélda bevitelekor véletlenszerűen kinullázzuk („kiejtjük”) a hálót alkotó neuronok kimenetének valahány (általában 10-50) százalékát [10]. Ennek az a hatása, hogy az azonos rétegbe eső neuronok kevésbé tudnak egymásra hagyatkozni, így a probléma önálló megoldására vannak kényszerítve. Ennek köszönhetően lényegesen csökken a túltanulás veszélye. A módszert eredetileg javasló cikkben kiugró, 10-20 százaléknyi relatív hibacsökkenéseket értek el képi alakfelismerési és beszédfelismerési feladatokon.

A dropout technika előnye, hogy roppant egyszerűen implementálható, és elvileg minden esetben kombinálható a backpropagation algoritmussal. Az eredeti cikkben előtanított szigmoid hálók finomhangolása során alkalmazták, de azóta többen megmutatták, hogy rectifier neuronhálók tanításával kombinálva is remekül működnek [8]. További javulás érhető el az eredményekben, ha tanítás során minden inputvektort többször (2-3-szor) is felhasználunk egy iteráción belül, különböző neuronkieséssel. Ugyan ez némileg javít az eredményeken, de az algoritmus futásidejét sokszorosára növeli, ezért mi csak egyszer használtunk fel minden inputvektort egy tanítási iterációban.

3. Kísérleti eredmények

A továbbiakban kísérleti úton vizsgáljuk meg, hogy a mély neuronhálók különböző tanítási módszerekkel milyen pontosságú beszédfelismerést tesznek lehetővé. Az akusztikus modellek készítése és kiértékelése az ún. hibrid HMM/ANN sémát követi [2], azaz a neuronhálók feladata az akusztikus vektorok alapján megbecsülni a rejtett Markov-modell állapotainak valószínűségét, majd ezek alapján a teljes megfigyeléssorozathoz a rejtett Markov-modell a megszokott módon rendel valószínűségeket. Mivel a neuronhálóknak állapotvalószínűségeket kell visszaadniuk, ezért minden esetben első lépésben egy rejtett Markov-modellt tanítottunk be a HTK programcsomag használatával [12], majd ezt kényszerített illesztés üzemmódban futtatva kaptunk állapotcímkeket minden egyes spektrális vektorhoz. Ezeket a címkeket kellett a neuronhálónak megtanulnia, amihez inputként az aktuális akusztikus megfigyelést, plusz annak 7-7 szomszédját kapta meg.

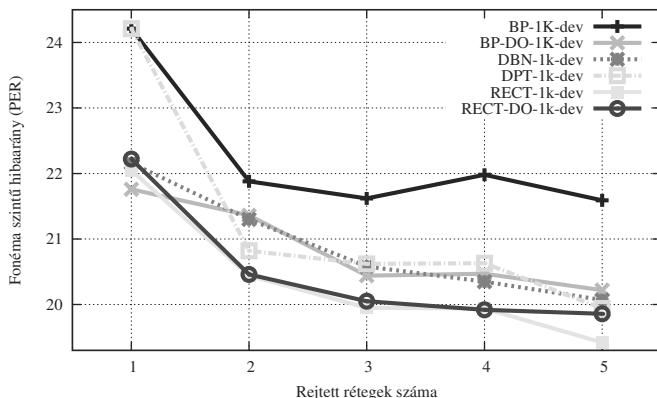
A modellek kiértékelését háromféle adatbázison végeztük el. Mindhárom esetben azonos volt az előfeldolgozás: e célra a jól bevált melkepsztrális együtt-hatókat (MFCC) használtuk, egész pontosan 13 együtthatót (a nulladikat is beleértve) és az első-második deriváltjaikat. A híradós adatbázis esetében szószintű nyelvi modellt is használtunk, a többi adatbázis esetén pusztán egy beszédhang bigram támogatta a beszédhang szintű felismerést.

Mindegyik módszer esetében 128-as batch-eken tanítottunk, a momentumot 0.9-re állítottuk és backpropagation algoritmus esetén a korai leállást használtuk, a betanított mély hálók minden rejtett rétege 1024 neuronból állt.

A DBN előtanítás esetén a paraméterezés annyiban változott a 2013-as cikkünkben közölthöz képest, hogy lényegesen kevesebb epochon keresztül futtattuk a kontrasztív divergencia algoritmust az egyes RBM-ekre: 5 epoch a GRBM esetén és 3 a többi esetén a tavalyi 50-20 helyett. Tapasztalataink szerint ez volt az az iterációszám, amely során a rekonstrukciós hiba lényegesen csökkent, az ezt követő epochokban a súlyok is már csak minimálisan változtak. Az epochszám jelentős csökkentésével a tanításhoz szükséges idő is számottevően csökkent.

A diszkriminatív előtanítás esetén minden új rejtett réteg hozzáadása után 4 iteráción keresztül előtanítottunk 0.01-es fix tanulási rátával.

A rectifier neuronhálók estében a tanulási ráta 0.001 volt, illetve



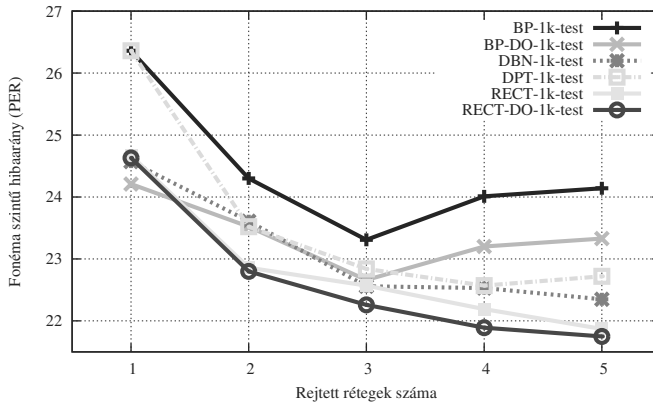
3. ábra. A különböző módszerek eredményei a TIMIT validációs halmazán a rejtett rétegek számának függvényében

minden iteráció végén a súlyokat normalizáltuk, hogy az egy neuronhoz tartozó súlyok 2-es normája 1 legyen.

A dropout módszer esetén szigmoid hálókra a 10%-os neuronkiesési valószínűséget találtuk a legjobbnak, rectifier hálók estén pedig a 20%-ot. A tanítási iterációk végén a [10]-ben javasolt módon a súlyokat csökkentjük 10 illetve 20%-kal (a neuronkiesési valószínűséggel), hogy kompenzáljuk az a tényt, hogy tesztelés során a neuronok nem „esnek ki” véletlenszerűen.

3.1. TIMIT

A TIMIT adatbázis a legismertebb angol nyelvű beszédatadtbázis [13]. Habár mai szemmel nézve már egyértelműen kicsinek számít, a nagy előnye, hogy rengeteg eredményt közöltek rajta, továbbá a mérete miatt viszonylag gyorsan lehet kísérletezni vele, ezért továbbra is népszerű, főleg ha újszerű modellek első kiértékeléséről van szó. Esetünkben azért esett rá a választás, mert több mély neuronhálós módszer eredményeit is a TIMIT-en közzétették, így kézenfekvőnek tűnt a használata az implementációnk helyességének igazolására. A TIMIT adatbázis felosztására és címkézésére a tavalyi cikkünkben [5] ismertetett (és amúgy sztenderdnek számító) módszert használtuk. A továbbiakban csak monofón eredményeket közlünk.



4. ábra. A különböző módszerek eredményei a TIMIT core teszt halmazon a rejtett rétegek számának függvényében

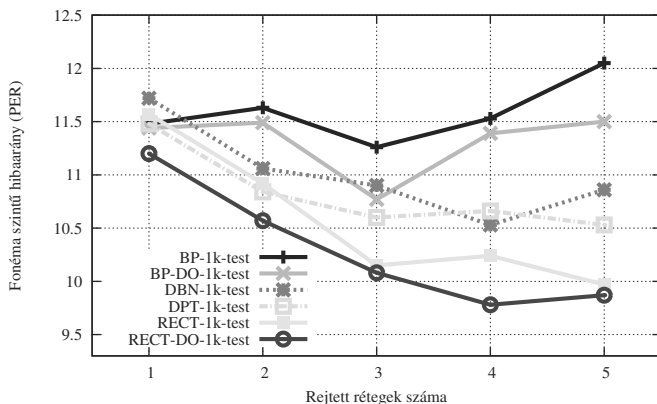
A TIMIT adatbázison elért beszédhang szintű eredményeket láthatjuk a 4. és 3. ábrán². Jól látható, hogy a hagyományos backpropagation tanítóalgoritmusnál mindegyik ismertett módszer jobban teljesített, ezen felül az is megfigyelhető, hogy a két előtanításos módszer nagyjából azonos eredményeket ért el.

A 4. és 3. ábrát összevetve megállapítható, hogy a dropout módszert alkalmazva a rectifier hálók esetén csökkent a túltanulás, szigmoid hálónál azonban növekedett.

A teszt halmazon a legjobb eredményeket a rectifier hálókkal tudtuk kihozni: 21.75%, ami nagyjából 3%-os relatív javulás az előtanításos módszerekhez képest, illetve 7%-os relatív javulás a legjobb hagyományos (előtanítás nélküli) módszerhez képest. A korábbi cikkünkben a core teszt halmazhoz közölt legjobb monofón eredményünkhöz (22.8%) képest a legjobb módszerrel több mint 1%-os javulást sikerült elérnünk.

A 4. ábrán megfigyelhető a dropout módszer hatékonysága is: míg szigmoid hálók esetén átlagosan 1%-os javulást hozott, ami 4%-os relatív javulásnak felel meg, addig a rectifier hálók esetén lényegesen kisebb a javulás. Ez utóbbinak az oka abban keresendő, hogy megfigyeléseink

²Jelmagyarázat: **BP**: backpropagation, **BP-DO**: backpropagation+dropout, **DBN**: DBN előtanítás, **DPT**: diszkriminatív előtanítás, **RECT**: rectifier háló, **RECT-DO**: rectifier háló+dropout



5. ábra. A különböző módszerek eredményei a hangoskönyv adatbázis tesztalmazán a rejtett rétegek számának függvényében

szerint a rectifier hálók neuronjainak átlagosan 70%-a inaktív tanítás során, ezt a dropout módszerrel kb. 75%-ra tudtuk növelni, ami nem hozott jelentős javulást az eredményekben.

Megvizsgáltuk továbbá, hogy a legjobban teljesítő mély neuronhálónkkal megegyező paraméterszámú hagyományos, egy rejtett réteges háló milyen eredményeket tud elérni. Az így kapott 23.5% a teszt halma-zon lényegesen rosszabb mint a mély struktúrával elérhető eredmények, ami igazolja, hogy célszerű azonos paraméterszám esetén a mély struk-túrájú hálót választani.

3.2. Hangoskönyv

A hangoskönyv adatbázisunk megegyezik a tavaly használttal (Krúdy Gyula: Szindbád kalandjai). Tanításra kb. 2 órányi anyag állt rendelkezésre, tesztelésre pedig 20 percnyi. Az 5. ábrán a különböző rétegszámmal elért eredményeket láthatjuk.

Megfigyelhető, hogy a különböző tanítási módszerek eredményei már jobban eltérnek, mint a korábbi adatbázison, viszont továbbra is megállapíthatjuk, hogy ha 2 vagy annál több rejtett réteget használunk, akkor a hagyományos módszer mindig a legrosszabb. A TIMIT-en elért eredményekhez hasonlóan itt is a rectifier hálók teljesítettek a legjobban,

a legjobb eredményt (9.78%-ot) 4 rejtett réteggel dropout módszerrel tanítva értük el, ez több mint 13%-os hibacsökkenést jelent.

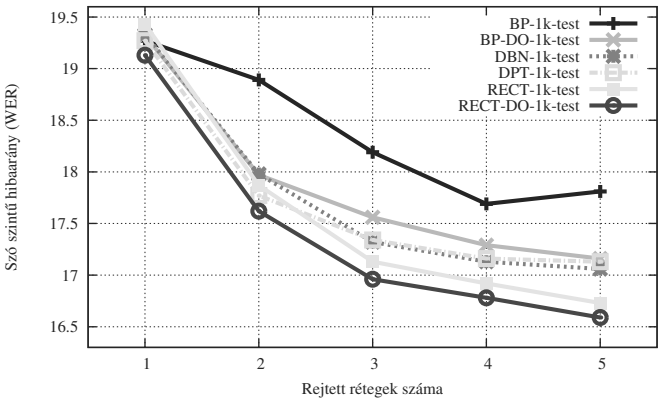
Az adatbázis sajátosságai miatt az figyelhető meg, hogy a hagyományosan tanított hálók esetén a rétegszám növelésével nem tudunk jelentős javulást elérni. A dropout módszer szigmoid hálók esetén 3 rejtett réteggel teljesített a legjobban – nagyjából 0.5%-kal jobb eredményt ért el –, rectifier háló esetén pedig 4 rejtett réteg esetén javított jelentősebben. A hagyományosan (azaz csak backpropagation algoritmussal) tanított illetve a backpropagation+dropout módszerrel tanított szigmoid hálók kivételével mindegyik módszer esetén 4 vagy 5 rejtett réteggel értük el a legjobb eredményt. A tavalyi legjobb monofón eredményhez (10.62%) képest idén jelentős javulást tudtunk elérni (9.78%).

3.3. Híradós adatbázis

A magyar nyelvű híradós adatbázis, amely méretét tavaly óta sikerült jelentősen megnövelnünk, nagyjából 28 órányi hanganyagot tartalmaz. Az adatbázis felosztása: 22 órányi anyag a betanítási rész, 2 órányi a fejlesztési halmaz és a maradék 4 órányi hanganyag pedig a tesztelő blokk.

A híradós adatbázison szószintű felismerést tudtunk végezni, az ehhez szükséges nyelvi modellt az origo (www.origo.hu) hírportál szövegei alapján készítettük. Az így előálló korpusz nagyjából 50 millió szavas, mivel a magyar nyelv agglutináló (toldalékoló) nyelv. A korpusz lecsökkentése érdekében csak azokat a szavakat használtuk, amelyek legalább kétszer előfordultak a híryananyagban, így 486982 szó maradt. A szavak kiejtését a Magyar Kiejtési Szótár-ból [14] vettük. A trigram nyelvi modellünket a HTK [12] nyelvi modellező eszközei segítségével hoztuk létre. Ezen adatbázis esetén környezetfüggő (trifón) modelleket használtunk, ennek eredményeképp az adatbázis mérete miatt 2348 állapot adódott, azaz ennyi osztályon tanítottuk a neuronhálókat.

A 6. ábrán láthatóak az elért szószintű eredmények különböző rejtett réteg-szám mellett. Ezen adatbázis esetén is elmondható, hogy a hagyományos módszer adja a legrosszabb eredményt, továbbá az is megfigyelhető, hogy a tanító adatbázis megnövekedése miatt a különböző tanítási módszerek eredményei jóval kevésbé térnek el. Továbbra is a rectifier hálók adják a legjobb eredményt (16.6%), ez a hagyományos módszerrel elérhető legjobb eredményhez (17.7%) képest 6%-os relatív



6. ábra. A különböző módszerek eredményei a híradós adatbázis tesztalmazán a rejtett rétegek számának függvényében

Módszer	Előtanítási idő	Finomhangolási idő
Hagyományos	—	4.5 óra
Dropout	—	5.5 óra
DBN előtanítás	1 óra	4 óra
Diszkriminatív előtanítás	2.5 óra	3 óra
Rectifier háló	—	4 óra
Rectifier háló + Dropout	—	4.5 óra

1. táblázat. Az 5 rejtett réteges háló különböző módszerekkel történő tanításához szükséges idők

hibacsökkenés.

A híradós adatbázishoz közölt korábbi legjobb eredményünkhöz (16.9%) [15] képest is sikerült javítanunk, a rejtett rétegek neuronszáma 2048-ról 1024-re csökkentése mellett.

Végül megvizsgáltuk az egyes módszerek időigényét: az 1. táblázatban az 5 rejtett réteges mély hálók különböző módszerekkel történő betanításához szükséges időket láthatjuk a híradós adatbázisra, GeForce GTX 560 Ti grafikus kártyát használva. Megállapítható, hogy a rectifier hálók nem csak jobb eredményeket adnak, de a betanításukhoz is

kevesebb idő szükséges, mint a többi módszer esetén.

4. Konklúzió

Cikkünkben bemutattuk a mély neuronhálókra épülő akusztikus modelleket, illetve a betanításukhoz legújabban javasolt algoritmusokat. A kísérleti eredmények egyértelműen igazolják, hogy az új algoritmusok jobb eredményeket tudnak adni, miközben egyszerűbbek és/vagy kisebb időigényűek, mint az eredeti DBN előtanításra alapuló megoldás. Az eredményeket és a tanítási időket figyelembe véve megállapíthatjuk, hogy a legjobb módszer – az itt ismertetettek közül – a rectifier hálók dropout módszerrel történő tanítása.

Hivatkozások

- [1] Grósz T., Kovács Gy., Tóth L., Új eredmények a mély neuronhálós magyar nyelvű beszédfelismerésben, *Proc. MSZNY* (2014), pp. 3–13.
- [2] H. Bourlard, N. Morgan, *Connectionist speech recognition: a hybrid approach*, Kluwer Academic, 1994.
- [3] G. E. Hinton, S. Osindero, Y-W. Teh, A fast learning algorithm for deep belief nets, *Neural Computation*, 18(7) (2006), pp. 1527–1554.
- [4] A-R. Mohamed, G. E. Dahl, G. E. Hinton, Acoustic modeling using deep belief networks, *IEEE Trans. Audio, Speech, and Language Processing*, 20(1) (2012), pp. 14–22.
- [5] Grósz T., Tóth L., Mély neuronhálók az akusztikus modellezésben, *Proc. MSZNY* (2013), pp. 3–12.
- [6] F. Seide, G. Li, X. Chen, D. Yu, Feature engineering in context-dependent deep neural networks for conversational speech transcription, *Proc. ASRU* (2011), pp. 24–29.
- [7] L. Tóth, Phone Recognition with Deep Sparse Rectifier Neural Networks, *Proc. ICASSP* (2013), pp. 6985–6989.

- [8] G. E. Dahl, T. N. Sainath, G. Hinton, Improving deep neural networks for LVCSR using rectified linear units and dropout, *Proc. ICASSP* (2013), pp. 8609–8613.
- [9] M. D. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. V. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean, G. Hinton, On rectified linear units for speech processing, *Proc. ICASSP* (2013), pp. 3517–3521.
- [10] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, A. A. Salakhutdinov, Improving neural networks by preventing co-adaptation of feature detectors, *CoRR*, *abs/1207.0580* (2012).
- [11] X. Glorot, Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, *Proc. AISTATS* (2010), pp. 249–256.
- [12] S. Young et al., *The HTK book*, Cambridge Univ. Engineering Department, 2005.
- [13] L. Lamel, R. Kassel, S. Seneff, Speech database development: Design and analysis of the acoustic-phonetic corpus, *DARPA Speech Recognition Workshop* (1986), pp. 121–124.
- [14] Abari K., Olaszy G., Zainkó Cs., Kiss G., Hungarian Pronunciation Dictionary on Internet (in Hungarian), *Proc. MSZNY* (2006), pp. 223–230.
- [15] L. Tóth, T. Grósz, A Comparison of Deep Neural Network Training Methods for Large Vocabulary Speech Recognition, *TSD* (2013), pp. 36–43.

Típusbiztos szkriptnyelvek generálása funkcionális beágyazott nyelvekből

Horváth Gábor¹, Kozár Gábor¹, Szűgyi Zalán²

¹Eötvös József Collegium*

² Eötvös Loránd Tudományegyetem, Informatikai Kar

{xaxax.hun, kozargabor}@gmail.com; lupin@ludens.elte.hu

1. Bevezetés

Ez a cikk az Informatics'2013 konferencián publikált „Generating type-safe script languages from functional APIs” cikkünk átdolgozása és lefordítása magyarra.

A fordított nyelvek számos előnnyel rendelkeznek az értelmezett nyelvekkel szemben. Többek között általában az ilyen nyelven megírt programok hatékonyabbak és a fordító fordítási időben több potenciális hibát talál meg. Ugyanakkor a forráskód lefordítása gyakran jelentős időt vehet el a fejlesztőktől, ezáltal hátráltatva a munkát. További problémát jelent, hogy ahhoz, hogy egy módosítást le lehessen tesztelni, újra kell indítani magát a programot is. Ez a folyamat gyakorta még több időt igényel, mint a fordítás. Egy lehetséges megoldás a dinamikus könyvtárak használata lenne, ugyanakkor a dinamikus könyvtárak kicserélése sem triviális feladat, miközben fut a program. Ráadásul ez a megoldás nagyban függne az adott platformtól is.

Az értelmezett nyelvek esetében az értelmező gyakran a típusellenőrzést csak közvetlen egy adott kódrészlet végrehajtása előtt végzi el. Ez a programozók számára nagyobb flexibilitást nyújt, azonban ez azt

*Horváth Gábor 2011–, Kozár Gábor 2011–

jelenti, hogy a futó programban lehetnek rosszul típusozott részek is. Ugyanakkor, mivel nincs szükség a forráskód gyakori újrafordítására, ezért a fejlesztők gyors fejlesztés-tesztelés iterációkat tudnak végezni. Viszont éppen azért, mert kevesebb garanciát nyújt az értelmező a kód helyességével kapcsolatban, mint a statikusan típusozott nyelvek esetén, ezért a fejlesztőknek még nagyobb hangsúlyt kell fektetniük a tesztelésre. Sajnos az értelmezett nyelvek nem elég hatékonyak ahhoz, hogy rendszerprogramozási nyelveknek használjuk őket.

Bár léteznek hibrid megoldások, mint például a szkript előfordítása byte kódra, vagy a Just In Time fordítási modell, de ez a cikk nem ezekről a megoldásokról szól.

Igen gyakori, hogy egy alkalmazás esetén a teljesítmény szempontjából kritikus részeket egy fordított és statikusan típusozott nyelvben írják, majd egy API-t készítenek egy szkriptnyelv számára, amivel egyszerűen és gyorsan lehet a kevésbé teljesítmény kritikus részeket lefejleszteni. Sajnos ennek az API-nak az elérhetővé tétele a szkriptnyelv értelmezője számára gyakran nagyon sok munkát jelent. Emellett az elkészült szkriptek tesztelésére is rengeteg időt kell fordítani. Ez a cikk egy olyan megoldásról szól, ami segítségével jelentősen csökkenthető az előbb említett tevékenységekre fordított idő.

1.1. Motiváció

A Clang [4] egy modern C/C++/Obj-C fordító, ami moduláris felépítésű. Úgy tervezték meg, hogy könnyen lehessen különböző eszközöket fejleszteni ezekhez a programnyelvekhez a fordító könyvtárainak a felhasználása segítségével. Az egyik ilyen könyvtár egy beágyazott domain specifikus nyelvet kínál a felhasználó számára [2, 3, 5]. Ezt a nyelvet *ASTMatcher*nek hívják. Ennek a nyelvnek a segítségével mintákat lehet definiálni, amiket később a programkód szintaxis fájlra lehet illeszteni. Ez a nyelv funkcionális megközelítést alkalmaz.

A C++ az egyik legösszetettebb gyakorlatban is használt programnyelv. Ezt a bonyolultságot a C++ programok szintaxis fája természetes módon tükrözi. Ebből kifolyólag, ha egy bizonyos mintát akarunk megtalálni ezeken a szintaxis fákon, akkor elég kicsi a valószínűsége, hogy első próbálkozásra a megfelelő mintát fogjuk leírni. Ebből kifolyólag egy tipikus munkafolyamat, hogy egy a mintán apró változtatásokat végzünk, majd kipróbáljuk azt. Sajnos a C++ kódok elemzése elég sokáig

tart, ráadásul az alkalmazást ami az illesztést végzi újra kell fordítani a minta minden egyes módosítása után. Ebből kifolyólag a folyamat, ameddig sikerül a megfelelő mintát meghatározni, szükségtelenül sokáig tart.

Ezt a problémát megoldandó egy olyan értelmezett környezetre lenne szükségünk, ahol egyből megkapjuk a visszajelzést, hogy a minta mennyire felelt meg a követelményeknek [9]. Az értelmező felelős azért, hogy elemezze a mintát, amit kap, majd lefordítsa ezt `ASTMatcher`ek formájában megfogalmazott mintára. Ugyanakkor fontos, hogy az illegális lekérdezések ne kerülhessenek végrehajtásra, mivel ez esetben a környezet leállhat, és az újraindításnál a fejlesztő kénytelen újra kivárni a kódok elemzésével eltöltött időt. Emellett a `Clang` fordító gyorsan változik, ezért nem ritka, hogy a `ASTMatcher` könyvtár elemei jönnek és mennek. Ezért az értelmező által elfogadott nyelv nyelvtanját is folyton változtatni kellene.

A cikkben bemutatott megoldás célja az, hogy az értelező által elfogadott szkriptnyelv nyelvtanát generáljuk le automatikusan a típus-információkból, ezáltal nem szükséges a programozóknak lekövetnie a szkript nyelvben az `ASTMatcher` könyvtárban bekövetkezett változásokat. Emellett a szkriptnyelvben ne legyen lehetséges illegális minták megfogalmazása.

Számos alternatíva létezik ugyan, amivel szkriptnyelvek számára elérhetővé lehet tenni egy API-t. Ilyen például a `Simplified Wrapper and Interface Generator (SWIG)` [12] is. Azonban ezek az eszközök nem kezelik jól a `C++` nyelvben megtalálható *template*-eket. A *template*-ek viszont gyakran alapjául szolgálnak a beágyazott nyelveknek, ezért a már létező megoldások nem alkalmasok erre a feladatra.

1.2. `C++` és a Template Metaprogramozás

A bemutatott megoldás `C++`-ban [7] került megvalósításra, és sok metaprogramot tartalmaz. A `C++` *template*-ek egy Turing-teljes beágyazott nyelvet alkotnak a `C++`-ban. A *template*-ek segítségével megírt programok a `C++` kód fordítása közben futnak le. Ez az alapja a nyelvben a generatív programozásnak. Teljes szoftverkomponensek kigenerálása is lehetséges, például akár értelmezők generálása is. A generatív programozásra nem csak a `C++` alkalmas, tehát minden itt bemutatott módszer átvihető bármely nyelvre, ahol ugyanezek az

eszközök rendelkezésre állnak. Sőt, bizonyos részei a metaprogramoknak kiválthatóak reflexió segítségével is.

Az alapötlet az, hogy minden típusinformáció a fordító rendelkezésére áll, ismeri a konverziós szabályokat. A template specializációkkal történő mintaillesztés és a Substitution Failure Is Not An Error (SFINAE) [8] technika segítségével lehetséges ezeknek a konverziós szabályoknak a megállapítása és eltárolása fordítási időben. Ezekből a konverziós szabályokból állítható elő az a nyelvtan, amit az értelmező el fog fogadni. Bár ebben a megoldásban nem generáljuk ki az értelmező teljes elemzőprogramját, az is lehetséges lenne [6].

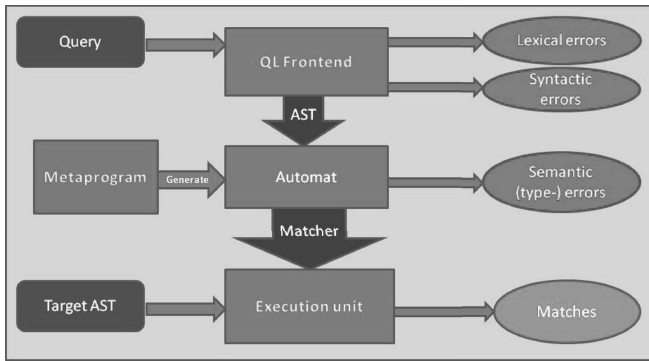
1.3. Architektúra áttekintése

Az eszköz, amit kifejlesztettünk, egy értelmezett lekérdezőnyelven alapszik, amivel lekérdezéseket lehet megfogalmazni a C++ kódbázisra. A lekérdezések gyakorlatilag minták, amiket a C++ kód szintaxisfájára próbálunk illeszteni. Az architektúra nagy vonalakban az 1. ábrán megtekinthető. Az eszköz bemenete egy ASTMatcher kifejezés szövegeként. Az értelmező elemzője elkészíti a szintaxis fáját a lekérdezésnek magának. A lexikális és szintaktikus hibák ebben a fázisban derülnek ki. Ezután a szemantikus analízis jön, ezt a részt végző kódot generáljuk ki metaprogramok segítségével. Itt derül ki, ha egy lekérdezés nem jól formált. Ezután kódgenerálás helyett a tényleges ASTMatcherok legenerálása történik, amit utána a végrehajtó egység fog végrehajtani (elvégezni a mintaillesztést). A végrehajtó egység egyik bemenete az ASTMatcher kifejezés, a másik pedig a C++ kód szintaxis fája.

A szemantikus analízisért felelős modul típusbiztos automatikus kiegészítésre is rendkívül egyszerűen felhasználható.

2. Implementáció

A megoldásunk függvényeket és funktorokat (függvényként viselkedő osztály) tud kezelni. A Clang [4] ASTMatcher könyvtárában a kifejezések [10] általánosított parancs mintát [1] megvalósító funktorokból épülnek fel. Ahhoz, hogy az értelmező végre tudja hajtani az értelmezett kifejezést, valahogy példányosítania kell ezeket a függvényobjektumokat. Ehhez a futási idejű polimorfizmust használja fel. Ehhez azonban a funktoroknak egy öröklődési hierarchiában kell lenniük. Amennyiben



1. ábra. Architektúra

nincsenek, a metaprogramnak mesterségesen kell becsomagolnia a funktorokat egy ilyen hierarchikus szerkezetbe. Sok problémát okoznak a túlterhelt függvények, ugyanis név alapján nem lehet eldönteni ebben az esetben, hogy pontosan melyik függvényről van szó. Ezekben az esetekben egy konverzió segítségével lehet meghatározni a pontos függvényt, ugyanakkor ennek a szintaxisa nagyon sok gépelést igényel.

2.1. Karakterláncok a metaprogramokban

Az első kihívás a karakterláncok kezelése. Ugyanis technikai okokból kifolyólag egy karakterlánc az nem legális template paraméter. A probléma az, hogy a szemantikus ellenőrző név alapján tudja csak azonosítani a függvényeket, ezért valahogy a metaprogramban mégis csak muszáj kezelni a karakterláncokat. Az erre kidolgozott módszerünk Sinkovics Ábel munkáján [11] alapszik. Az ő megoldását viszont módosítani kellett, mivel nálunk jelentős szempont volt, hogy minél egyszerűbben tudjunk fordítási idejű karakterláncból futási idejű objektumot generálni. Ábel megközelítésének a lényege az volt, hogy egy makró metaprogram segítségével feldarabolja a karakterláncot karakterek sorozatára, és a fel nem használt karakterek helyére nullákat rak.

```
#define DO(z, n, s) at(s, n),
```

```
#define _S(s) \
```

```

BOOST_PP_REPEAT(String_Max_Length, \
                DO, s)

template <int N>
constexpr char
at(char const(&s)[N], int i)
{
    return i >= N ? '\0' : s[i];
}

```

Sinkovics Ábel a Boost MPL metaprogramozáshoz gyakran használt könyvtárban lévő vektor típust használta fel a karakterek tárolására. Mi azonban ehelyett a C++11-ben megjelent variadikus (változó paraméterszámú) template-ek paraméter csomagjaiban tároltuk a karaktereket. Ez azért előnyös, mert az új inicializációs szintaxis segítségével könnyen és hatékonyan tudunk futási idejű karakterláncokat létrehozni. A `MetaStringImpl` metaprogram eltávolítja a fölösleges záró nullákat is a karakterlánc végéről. A `GetRuntimeStr` metóduson látszik, hogy ebben a reprezentációban milyen egyszerű a futási idejű karakterlánc létrehozása.

```

template <char... cs>
struct Accumulator {
    static std::string GetRuntimeStr() {
        return {cs...};
    }
};

template <typename T, char...>
struct MetaStringImpl;

template <char... cs>
struct MetaString {
    typedef typename
        MetaStringImpl<
            Accumulator<>,
            cs...>::result str;

    static std::string GetRuntimeStr() {
        return str::GetRuntimeString();
    }
};

```

2.2. Típusinformáció tárolása

A következő akadály, hogy hogyan tároljuk el a konverziós adatokat úgy, hogy azt későbbiekben a legkönnyebben fel tudjuk használni. Makrók segítségével egyszerűvé tettük a függvények és funktorok regisztrációját a szemantikus elemző generátorához. Eltárolásra kerül a függvény szignatúrája, a neve, valamint az objektum típusa, ami ha példányosításra kerül, akkor utána függvényobjektumként is használható. Ezeket az információk a standard könyvtárban is megtalálható `std::tuple` osztályokban tároljuk.

```
template<typename T, T* ptr>
struct FunctionPointer;

template<typename Ret,
        typename... Args,
        Ret (*f)(Args...)>
struct FunctionPointer<Ret(Args...), f> {
    Ret operator()(Args... args) {
        return f(args...);
    }
};

#define FUNCTION(x) \
    std::tuple<decltype(x), \
        FunctionPointer<decltype(x), &x>, \
        MetaString<_S( #x )>>

#define FUNCTOR(x) \
    std::tuple< \
        decltype(&x::operator()), x, \
        MetaString<_S( #x )>>
```

A metaprogramok ki és bemenetei elsősorban típusok. Funktorok esetében egyszerű minden információt biztosítani, azonban függvények esetén ahhoz, hogy kapjunk példányosítható objektumot is, a függvény mutatójából készítünk egy egyedi típust. Ezt szolgálja a `FunctionPointer` template. Ezek a rendezett *n*-esek egy lista szerű fordítási idejű adat-szerkezetbe kerülnek.

2.3. A típusellenőrző kigenerálása

A metaprogram ezután generálni fog számos mátrixot a lista tartalma alapján. A mátrixok száma az meg fog egyezni a legnagyobb arítású függvény arításával. Mindegyik mátrix a függvények illetve funktorok neveivel van indexelve. Valójában a szintaxisfa csúcsai mind függvénykompozíciót jelölnek, például: $A(\dots, B(\dots), \dots)$, ahol B függvény eredményét tovább adjuk A -nak az i -edik paramétereként. Legyen az i -edik mátrix neve M_i . Az előbbi szintaxisfabeli csúcs akkor típushelyes, hogyha az $M_i[A, B]$ értéke igaz. Tehát a $M_i[A, B]$ hordozza azt az információt, miszerint a B visszatérési értéke implicit módon konvertálódik-e A -nak az i -edik formális paraméterének a típusára. A metaprogram ezeket a mátrixokat fordítási időben generálja ki.

Egy példán keresztül szemléltetve lehet a legkönnyebben látni, hogy mit is csinál ez a metaprogram. Tekintsük a következő egyszerű API-t:

```
struct A {};
struct B : A {};
struct C {};
struct E : B {};

A* func1(A*) { return 0; }
B* func2(A*, B*) { return 0; }
C* func3(A*) { return 0; }

struct D
{
    E* operator() (A*) { return 0; }
};
```

A szemantikus ellenőrzést végző automata generálásához elég beregisztrálnunk a függvényeinket és funktorainkat:

```
typedef typename Automata<
    FUNCTION(func1),
    FUNCTION(func2),
    FUNCTION(func3),
    FUNCTOR(D)
>::result GA;

std::set<std::string>
    expected {"func1", "func2", "D"};

auto tmp =
```

```
GA::GetComposables("func1", 0);

std::set<std::string>
    result(tmp.begin(), tmp.end());

EXPECT_EQ(expected, result);
```

Sajnálatos módon a felsorolás nem feltétlen triviális. Amennyiben túl van terhelve egy függvény, akkor minden túlterhelt változatát máshogy kell elnevezni a beregisztráláskor. Valójában a fentebb leírt makrók minden esetben a név mezőt is kitöltik a tuple-ban, tehát túlterhelt függvények esetén nincs lehetőség ezen makrók használatára.

Ezek után, ha meg akarjuk határozni, hogy a `func1` nevű függvény első paramétereként mely függvényhívások fordulhatnak elő, egy `GetComposables` hívással megkaphatjuk a neveiket. Gyakorlatilag ebben az esetben minden olyan függvény megfelel, aminek a visszatérési értéke az A , vagy A egy leszármazottjára mutató mutatóval tér vissza.

A fenti példához hasonló módon dönti el a szemantikus elemző is, hogy az adott csúcs a szintaxisfában sérti-e a típuskonverziós szabályokat.

3. Összefoglalás

A generatív programozás egy nagyon népszerű paradigma számos feladat megoldására. Sok automatizált eszköz létezik bizonyos feladatok elvégzésére. Ugyanakkor, mint kiderült, a C++ alkalmas a saját API-jának a kivezetésére típus biztos nyelvek számára mindenféle külső eszköz nélkül is. Ez is mutatja a statikus típusozás gazdagságát. A majdan megjelenő C++17-es szabványnak már része lesz a fordítási idejű statikus reflexió is. Ennek a segítségével jelentősen hatékonyabbá tudjuk majd tenni a megoldásunkat, valamint ki fogjuk tudni terjeszteni imperatív beágyazott nyelvekre és API-kra is. Összességében egy olyan módszert és eszközt alkottunk meg, amelynek hála, néhány esetben nagyban növelhető a fejlesztők hatékonysága.

Hivatkozások

- [1] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison Wesley, 2001.
- [2] M. Fowler, *Domain-Specific Languages*, Addison-Wesley, 2010.
- [3] P. Hudak, Building domain-specific embedded languages, *ACM Comput. Surv.*, 28(4) (1996).
- [4] C. Lattner, *LLVM and Clang: Next Generation Compiler Technology*, The BSD Conference, 2008.
- [5] M. Mernik, J. Heering, A. Sloan, When and how to develop domain-specific languages, *ACM Computing Surveys*, 37(4) (2005), pp. 316–344.
- [6] Z. Porkoláb, Á. Sinkovics, Domain-specific Language Integration with Compile-time Parser Generator Library, *Proc. 9th international conference on Generative programming and component engineering (GPCE 2010)*, ACM, 2010., pp. 137–146.
- [7] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 4th edition, 2013.
- [8] D. Vandervoorde, N. M. Josuttis, *C++ Templates: The Complete Guide*, Addison-Wesley Professional, 2002.
- [9] M. J. Varanda, M. Mernik, D. Da Cruz, P. R. Henriques, Program comprehension for domain-specific languages, *Comput. Sci. Inf. Syst.*, 5(2) (2008), pp. 1–17.
- [10] Clang AST matcher library
<http://clang.llvm.org/docs/LibASTMatchers.html>
(11 June 2013.)
- [11] Á. Sinkovics, D. Abrahams, *Using strings in C++ template metaprograms*
<http://cpp-next.com/archive/2012/10/using-strings-in-c-template-metaprograms/>
(02 June 2013.)

-
- [12] D. M. Beazley, SWIG: an easy to use tool for integrating scripting languages with C and C++, *Proc. 4th conference on USENIX Tcl/Tk Workshop*, 4 (1996), pp. 15.

Bevezetés a homotópia-típuselméletbe

Kaposi Ambrus*

Eötvös Loránd Tudományegyetem, Informatikai Kar

`kaposi.ambrus@gmail.com`

1. Bevezetés

A számítógéptudományban sokféle módszer használatos a programok helyes működésének biztosítására: a program futtatása különböző bemenetekkel és a kimenetek ellenőrzése; a program futás idejű monitorozása; a program egy egyszerűsített modelljének szimulálása, és annak ellenőrzése, hogy a szimulált világ lehetséges állapotaiban megfelelő programműködést tapasztalunk-e; a program részleges specifikálása típusokkal, és a program fordítása során ennek a specifikációnak való megfelelés ellenőrzése; a program helyességének bizonyítása formális eszközökkel vagy bizonyítottan helyes program szintézise.

A fentiek közül a típusokkal való specifikálás az egyik legelterjedtebb módszer, hiszen a fordítás egyik fázisaként a fejlesztés folyamatába egyszerűen beépíthető. A fordítóprogram automatikusan elvégzi a típusellenőrzést, a nem típushelyes programot visszautasítja, és ezáltal sokféle, gyakran előforduló programozási hibát képes kiszűrni.

Ha egy típusrendszer nem elég kifejező, az az absztrakció rovására megy, pl. egy egyszerű típusrendszerrel rendelkező programozási nyelvben külön programra van szükség ahhoz, hogy egész számok listájának hosszát ill. karakterek listájának hosszát meghatározzuk, mert ezeknek különböző típusa van, $\text{List Int} \rightarrow \text{Int}$ ill. $\text{List Char} \rightarrow \text{Int}$. Egy lehetőség ilyenkor a típusrendszer megkerülése azáltal, hogy kiskapukat teszünk bele. Másik lehetőség, hogy kifinomultabb típusrendszert

*University of Nottingham, United Kingdom

használunk, mely képes polimorf típusokat leírni. Ekkor az előbbi két típus egyesíthető az alábbi típusban: $\forall a : \text{Type} . \text{List } a \rightarrow \text{Int}$, és az ilyen típusú program működni fog mindenfajta listára. A Girard-Reynolds típusrendszer [12] megengedi az ilyen konstrukciókat, és ezzel nemcsak lehetővé teszi az absztrakciót, de a programozót rá is kényszeríti a reprezentáció-független programok írására [32], [27]. Erre a típusrendszerre épül az ML [26] és Haskell [30] programozási nyelv.

Ha az előbb említett típusrendszerben szeretnénk a véges elemszámú típusokat megvalósítani, külön-külön kell definiálnunk a $\text{Fin1} : \text{Type}$ (1-elemű), $\text{Fin2} : \text{Type}$ (2-elemű) stb. típusokat. Ez a példa azt mutatja, hogy a típusrendszer továbbra is akadályozza az absztrakciót, de most a típusok szintjén. Ha nincs szükségünk nagy biztonságra, egy Fin típusba egyesíthetjük az összeset, melyet pl. a természetes számok típusával definiálunk, és a programozóra bízunk, hogy ahol Fin2 -t vár a program, ott véletlenül se adjon egy Fin3 -beli értéket. Ha azonban szükségünk van az elkülönítésre, és nem egy metaprogrammal szeretnénk a típus-definíciókat generálni, mert pl. az elemszámra nincs felső korlátunk, az alábbi típusnak megfelelő programra van szükségünk: $\text{Fin} : \text{Nat} \rightarrow \text{Type}$. Ez a Fin típusoknak egy indexelt családja, minden egyes természetes számra egy-egy külön típus, az indexelő típus a Nat . $\text{Fin } 3$ pl. a három-elemű típus. Azzal, hogy értékek (egy természetes szám) jelentek meg a típusban, függő típusrendszerhez jutottunk. Függő típusok használatával a Curry–Howard izomorfizmuson [19] keresztül tetszőleges, matematikailag leírható tulajdonság kifejezhető típusokkal (lásd 2.8. pont). A függő típusrendszer az elképzelhető legkifejezőbb típusrendszer, hiszen minden tulajdonság, amit valaha le szeretnénk írni egy programról, matematikai képlettel kifejezhető.

Egy ilyen típusrendszer a programozónak sokféle lehetőséget kínál: annak megfelelően, hogy mekkora biztonságot kíván a feladat, pl. az egészek listájának rendezését végző `sort` programot az alábbi típusokkal szerelheti fel, biztonságosság szerint növekvő sorrendben:

- `sort : List Int → List Int`
- `sort : (xs : List Int) → (ys : SortedList Int)`
- `sort : (xs : List Int) → (ys : List Int) × (sorted ys)`
 $\times (\text{length } xs = \text{length } ys)$

• `sort : (xs : List Int) → (ys : List Int) × (sorted ys)
 × (ys 'permutationOf' xs)`

Egy másik lehetőségként a programozó a `sort` program típusát meghagyhatja a legelsőnek, és egy külön programot írhat az alábbi típussal:

`(xs : List Int) → let ys = sort xs in
 (sorted ys) × (ys 'permutationOf' xs)`

Ez a program annak bizonyításának felel meg, hogy a `sort` program egy rendezett listát ad vissza, és ez a rendezett lista a bemeneti lista egy permutációja. Ilyen módon a programok bizonyításokat is tartalmazhatnak, és a programok helyességének (valamilyen szinten való) bizonyítása a programozás egy lépése lehet.

Függő típusrendszerrel rendelkező programozási nyelvek az Agda [29] és Idris [7]. Bevezető jellegű könyvek a típuselméletbe magyarul [10], angolul a logika felől közelítve [13], a programozás felől közelítve [15].

A matematika konstruktív megalapozására függő típuselméleteket régóta használnak [33]. A Curry–Howard izomorfizmuson keresztül a matematikai állítások típusoknak, az állítás bizonyításai adott típusú programoknak felelnek meg. A legnépszerűbb, Curry–Howard izomorfizmust használó számítógépes tételbizonyító rendszer (más szavakkal függő típusrendszerű programozási nyelv) a Coq [24]. Ennek alapja az intenzionális Martin-Löf típuselmélet [22]. Bár nagy és bonyolult matematikai tételeket bizonyítottak ebben a rendszerben ([14], ¹), használata mégis nehézkes, mert olyan, a matematikában (és emiatt a programokról való érvelésben) gyakran használt alapelvek, mint a pontonként egyenlő függvények egyenlősége (függvény extenzionalitás) ill. az izomorf halmazok (típusok) egyenlősége nem teljesülnek benne. Az ekvivalencia-osztályokkal való műveletek is kényelmetlenek, pl. tiszta Martin-Löf típuselméletben a valós számok nem definiálhatók [21]. Ezek a problémák mind a típuselmélet egyenlőség-fogalmával kapcsolatosak. A homotópia-típuselmélet [31] új megvilágításba helyezi az egyenlőség típust, és választ ad a fenti problémákra, egy matematikára és programozásra egyaránt kényelmesebben használható típuselmélet formájában.

¹<http://www.msr-inria.fr/news/feit-thomson-proved-in-coq>

Az alábbiakban bevezetjük a Martin-Löf típuselméletet, rávilágítunk az egyenlőség típus néhány tulajdonságára, bemutatjuk azokat az extensionális alapelveket, amiket használni szeretnénk, majd megmutatjuk, hogy a típusok topologikus terekként való értelmezése hogyan teszi érthetővé az egyenlőség különleges tulajdonságait és teszi lehetővé az előbbi alapelvek használatát.

Ezen írás megértéséhez egy számítástudományi alapképzésnek megfelelő matematikai tudás elégséges, valamely funkcionális programozási nyelv ismerete előny. Néhány témakört helyhiány miatt csak nagy vonalakban tudunk érinteni, a részleteket hivatkozásokkal igyekszünk pótolni. A magyar szóhasználatban [10]-t igyekszünk követni.

A típuselméletet leggyakrabban a mienkhez hasonló módon, szintaktikusan vezetik be, ami első olvasásra nagyon töménynek, esetleg rejtélyesnek tűnhet. A megadott levezetési szabályok között azonban mély szimmetriák vannak, melyeknek a kifejezésére még nem találtuk meg a legjobb formát – talán a kategóriaelmélet [3] lesz az, de mivel kevesen ismerik az alapfogalmakat, maradunk a szintaktikus bemutatásnál. A kategóriaelmélet nyelvét használó szép bevezetés a típuselméletbe pl. [17].

2. Martin-Löf típuselmélet

Ebben a pontban a Martin-Löf típuselméletet vezetjük be, aki ismerős a témával, az gyorsan átfuthatja a szabályokat, aki nem, annak ez egy gyorstalpaló bevezető lesz. A szabályok olvasásakor az elsőrendű logikában tanultakkal kapcsolatos intuícióra érdemes támaszkodni.

Martin-Löf típuselmélete [22] egy formális matematikai rendszer², mely következtetések levezetésére alkalmas. A formális rendszer ábécéjét, nyelvtani szabályait a levezetési szabályokon keresztül implicit módon adjuk meg.

Négyféle következtetési formát különböztetünk meg:

²Martin-Löf típuselméletének intenzionális változatát adjuk meg implicit helyettesítéssel, Russel-féle univerzumokkal és definicionális β és η szabályokkal rendelkező Π és Σ típusokkal.

$\Gamma \vdash$	Γ egy érvényes környezet
$\Gamma \vdash t : A$	Γ környezetben t kifejezés típusa A
$\Gamma \equiv \Delta \vdash$	Γ és Δ környezetek definicionálisan egyenlők
$\Gamma \vdash u \equiv v : A$	u és v , Γ környezetben A típusú kifejezések definicionálisan egyenlők

Utóbbi esetben a környezetet és a típust gyakran elhagyjuk, és egyszerűen $u \equiv v$ -t írunk.

A kifejezésekre gondolhatunk programokként, melyek valamilyen típussal rendelkeznek. A típusra matematikai állításként is gondolhatunk, a kifejezés ennek bizonyítása. Két program definicionálisan egyenlő, ha futtatásuk ugyanarra a végeredményre jut. Pl. $4 + 3 \equiv 7$. A típusok is kifejezések, melyeknek típusa van, így a típusok és kifejezések egy szintaktikus kategóriában vannak, az intuíció miatt különítjük el őket informálisan.

A környezet a kifejezésben és a típusban levő szabad változók típusait adja meg, típusok listája: $x_1 : A_1, x_2 : A_2, \dots, x_n : A_n$. Ez a környezet az $A_1 \dots A_n$ típusokat tartalmazza, a változónevek csak címkék, azért van rájuk szükség, hogy könnyű legyen hivatkozni a típusokra. Emiatt a pontos név nem érdekes, két környezet, mely csak az elnevezésekben különbözik, definicionálisan egyenlő; hasonlóképpen két kifejezés, melyben a változók ugyanazokra a helyekre mutatnak, de különböző nevűk van, definicionálisan egyenlő (ezt a problémakört α -konverciónak nevezik, mi nem foglalkozunk vele). A_i -ben előfordulhatnak x_j szabad változók, ahol $j < i$. A környezetekre vonatkozó levezetési szabályok az alábbiak:

$$\frac{}{\vdash} \text{üres környezet} \qquad \frac{\Gamma \vdash A : U_i}{\Gamma, x : A \vdash} \text{környezet hosszabbítás}$$

$$\frac{\Gamma, x : A, \Delta \vdash}{\Gamma, x : A, \Delta \vdash x : A} \text{változó bevezetés}$$

A környezet hosszabbításban x -nek egy olyan változónak kell lennie, ami Γ -ban nem szerepel. U_i minden i természetes számra egy típus. Zárt kifejezéseknek nevezzük azokat, melyekben nincs szabad változó, tehát egy olyan t , melyre valamely A -ra a $\vdash t : A$ következtetés levezethető.

A definicionális egyenlőség ekvivalencia-reláció, és definicionálisan egyenlő környezetek és kifejezések bármely esetben felcserélhetőek. A következő (nem túl érdekes) levezetési szabályok ezeket fejezik ki, első

olvasásra ezek nyugodtan átugorhatóak, a teljesség kedvéért tesszük őket ide:

$$\begin{array}{c}
 \frac{\Gamma \vdash}{\Gamma \equiv \Gamma \vdash} \quad \frac{\Gamma \equiv \Delta \vdash}{\Delta \equiv \Gamma \vdash} \quad \frac{\Gamma \equiv \Delta \vdash \quad \Delta \equiv \Theta \vdash}{\Gamma \equiv \Theta \vdash} \\
 \\
 \frac{\Gamma \vdash t : A}{\Gamma \vdash t \equiv t : A} \quad \frac{\Gamma \vdash u \equiv v : A}{\Gamma \vdash v \equiv u : A} \quad \frac{\Gamma \vdash u \equiv v : A \quad \Gamma \vdash v \equiv w : A}{\Gamma \vdash u \equiv w : A} \\
 \\
 \frac{\Gamma \equiv \Delta \vdash \quad \Gamma \vdash A \equiv B : \mathcal{U}_i \quad \Gamma \vdash t : A}{\Delta \vdash t : B} \\
 \\
 \frac{\Gamma \equiv \Delta \vdash \quad \Gamma \vdash A \equiv B : \mathcal{U}_i}{\Gamma, x : A \equiv \Delta, x : B \vdash} \quad \frac{\Gamma \equiv \Delta \vdash \quad \Gamma \vdash A \equiv B : \mathcal{U}_i \quad \Gamma \vdash u \equiv v : A}{\Delta \vdash u \equiv v : B}
 \end{array}$$

Az utolsó előtti szabályhoz szintén hozzátartozik, hogy x változó friss legyen (ne szerepeljen Γ -ban és Δ -ban).

Ha a $\Gamma \vdash t : A$ következtetés levezethető, akkor azt mondjuk, hogy t az A típus eleme. A típusok típusát univerzumnak hívjuk, a kis típusok $\mathcal{U}_0$ -ban vannak, $\mathcal{U}_0$ -t magát nagy típusnak hívjuk, mert elemei olyan kifejezések, melyeknek vannak elemei. A legalsó szinten levő kifejezéseknek nincsenek további elemei, ezeket kifejezés-konstruktoroknak (vagy egyszerűen konstruktoroknak) hívjuk (eddig még egy ilyennel sem találkoztunk). Az univerzumokra a következő szabályaink vannak (i bármely természetes szám):

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathcal{U}_i : \mathcal{U}_{i+1}} \text{ univ. képzés} \quad \frac{\Gamma \vdash A : \mathcal{U}_i}{\Gamma \vdash A : \mathcal{U}_{i+1}} \text{ univ. kumulativitás}$$

Az univerzumok megszámlálhatóan végtelen hierarchiájára azért van szükség, mert az $\mathcal{U}_0 : \mathcal{U}_0$ szabály inkonzisztenciához vezetne (Burali-Fori paradox, [11]) – az inkonzisztencia itt azt jelenti, hogy a $\cdot \vdash t : 0$ következtetés levezethető valamely t -re, ahol 0 az üres típus, lásd később.

A típuselméletben egy-egy adott típushoz tartozó levezetési szabályok a következő formában jelennek meg: típusképző szabály, bevezető szabály, eliminációs szabály, számítási (β) szabály, egyediség (unicitás, η) szabály, a konstruktorok definicionális egyenlőséggel való kompatibilitását kifejező szabályok. Ilyen formában adjuk meg a függvény, a függő pár (szigma), a 0-elemű típus, a 2-elemű típus, a természetes számok és az egyenlőség típus levezetési szabályait. Végül röviden megemlítjük, hogy általános induktív típusokat milyen módon képezhetünk.

2.1. A függvény típus levezetési szabályai

A $\prod_{x:A} B$ függvény típusra gondolhatunk úgy is, mint a $\forall_{x:A}.B$ állításra, melyben az univerzális kvantor az A típus (halmaz) elemeire vonatkozik, tehát azt mondja, hogy bármely x -nek nevezett A -beli elemre teljesül B (amely tartalmazhatja x -et). Pl. $\prod_{x:\mathbb{N}}(n+2=2+n)$ azt fejezi ki, hogy bármely n természetes számra $n+2$ egyenlő $2+n$ -el (a természetes számokat, összeadást és egyenlőséget később definiáljuk).

$$\frac{\Gamma \vdash A : \mathcal{U}_i \quad \Gamma, x : A \vdash B : \mathcal{U}_i}{\Gamma \vdash \prod_{x:A} B : \mathcal{U}_i} \Pi \text{ képzés}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : \prod_{x:A} B} \Pi \text{ bev.} \quad \frac{\Gamma \vdash f : \prod_{x:A} B \quad \Gamma \vdash a : A}{\Gamma \vdash fa : B[a/x]} \Pi \text{ elim.}$$

$$\frac{\Gamma, x : A \vdash t : B \quad \Gamma \vdash a : A}{(\lambda x.t)a \equiv t[a/x] : B[a/x]} \Pi \beta \quad \frac{\Gamma \vdash f : \prod_{x:A} B}{\Gamma \vdash f \equiv \lambda y.fy : \prod_{x:A} B} \Pi \eta$$

$t[a/x]$ egy meta-jelölés arra a kifejezésre, melyet úgy kapunk, hogy t -ben az x változó összes előfordulását a -ra cseréljük. B -t indexelt családnak (típuscsaládnak) nevezzük, a Γ környezetben $B[a/x]$ egy típus minden $a : A$ -ra, A az indexelő típus.

$\prod_{x:A} B$ -re további három jelölés $\prod_{x:A} B$, $\Pi A B$ és $(x : A) \rightarrow B$. Π és λ változót kötnek meg, melyek csak hatáskörükben látszanak. Mindkettő hatásköre a lehető legtovább terjed, tehát pl. $\lambda x.\lambda y.t \equiv \lambda x.(\lambda y.t)$, vagyis t -ben x és y is szerepelhet. Ha x nem szerepel B -ben, $\prod_{x:A} B$ helyett $A \rightarrow B$ -t írhatunk. $A \rightarrow B \rightarrow C$ jobbra zárójeleződik, tehát $A \rightarrow (B \rightarrow C)$ -t jelent. $A \rightarrow B$ -re gondolhatunk úgy, mint az A -ból következik B logikai állításra. λ egy konstruktor, Π típuskonstruktor, a függvényalkalmazás, melyet csak egymás mellé írással jelölünk, eliminátor.

A konstruktorok respektálják a definicionális egyenlőséget:

$$\frac{\Gamma \vdash A \equiv A' : \mathcal{U}_i \quad \Gamma, x : A \vdash B \equiv B' : \mathcal{U}_i}{\Gamma \vdash \prod_{x:A} B \equiv \prod_{x:A'} B' : \mathcal{U}_i} \quad \frac{\Gamma, x : A \vdash t \equiv r : B}{\Gamma \vdash \lambda x.t \equiv \lambda x.r : \prod_{x:A} B}$$

2.2. A szigma típus levezetési szabályai

A $\sum_{x:A} B$ típusra úgy gondolhatunk, mint a $\exists_{x:A}.B$ állításra. Annak bizonyítása, hogy létezik egy A -beli elem, melyre B igaz, egy függő pár,

mely egy x -nek nevezett A -beli elemből áll, és egy bizonyításból, hogy x -re teljesül B . Az alábbi bevezetési szabály ezt fejezi ki.

$$\frac{\Gamma \vdash A : \mathbb{U}_i \quad \Gamma, x : A \vdash B : \mathbb{U}_i}{\Gamma \vdash \sum_{x:A} B : \mathbb{U}_i} \Sigma \text{ képzés}$$

$$\frac{\Gamma \vdash u : A \quad \Gamma \vdash v : B[u/x]}{\Gamma \vdash (u, v) : \sum_{x:A} B} \Sigma \text{ bevezetés}$$

$$\frac{\Gamma \vdash t : \sum_{x:A} B}{\Gamma \vdash \pi_1 t : A} \Sigma \text{ elimináció}_1 \quad \frac{\Gamma \vdash t : \sum_{x:A} B}{\Gamma \vdash \pi_2 t : B[\pi_1 t/x]} \Sigma \text{ elimináció}_2$$

$$\frac{\Gamma \vdash u : A \quad \Gamma \vdash v : B[u/x]}{\Gamma \vdash \pi_1(u, v) \equiv u : A} \Sigma \beta_1 \quad \frac{\Gamma \vdash u : A \quad \Gamma \vdash v : B[u/x]}{\Gamma \vdash \pi_2(u, v) \equiv v : B[u/x]} \Sigma \beta_2$$

$$\frac{\Gamma \vdash t : \sum_{x:A} B}{\Gamma \vdash t \equiv (\pi_1 t, \pi_2 t) : \sum_{x:A} B} \Sigma \eta$$

A konstruktorok (Σ és $-$, $-$) respektálják a definicionális egyenlőséget, ezeket a levezetési szabályokat rövidített formában adjuk meg:

$$\frac{A \equiv A' \quad B \equiv B'}{\sum_{x:A} B \equiv \sum_{x:A'} B'} \quad \frac{u \equiv u' \quad v \equiv v'}{(u, v) \equiv (u', v')}$$

A nem-függő pár (Descartes-szorzat, logikai és) a függő pár speciális esete, ahol a második elem típusa nem függ az elsőtől: $A \times B$ -t egyszerűen $\sum_{x:A} B$ rövidítéseként definiáljuk abban az esetben, ha B -ben nem szerepel x .

$\sum_{x:A} B$ -re egy másik elterjedt jelölés $(x : A) \times B$.

2.3. Az üres típus levezetési szabályai

Az üres típus különleges, csak képzés és eliminációs szabálya van:

$$\frac{}{\Gamma \vdash 0 : \mathbb{U}_0} 0 \text{ képzés} \quad \frac{\Gamma \vdash t : 0 \quad \Gamma \vdash A : \mathbb{U}_i}{\Gamma \vdash \text{ind}_0 t : A} 0 \text{ elimináció}$$

Az üres típusra azért van szükségünk, mert a logikai negáció vele fejezhető ki: a $\neg A$ állítást az $A \rightarrow 0$ típussal fejezzük ki.

2.4. A kételemű típus levezetési szabályai

A kételemű típus szabályainak olvasásakor a számítástudományi intuíciónak érdemes hagyni: a `Bool` típusnak felel meg, eliminációs szabálya egy (függő típusú) `if then else` alkalmazásának, β szabályai pedig egy elágazást tartalmazó program futtatásának. η szabálya nincsen.

$$\frac{}{\Gamma \vdash 2 : \mathbf{U}_0} \text{ 2 képz.} \quad \frac{}{\Gamma \vdash \text{ff} : 2} \text{ 2 bev}_1. \quad \frac{}{\Gamma \vdash \text{tt} : 2} \text{ 2 bev}_2.$$

$$\frac{\Gamma, x : 2 \vdash A : \mathbf{U}_i \quad \Gamma \vdash u : A[\text{ff}/x] \quad \Gamma \vdash v : A[\text{tt}/x] \quad \Gamma \vdash t : 2}{\Gamma \vdash \text{ind}_2 u v t : A[t/x]} \text{ 2 elim.}$$

$$\frac{}{\text{ind}_2 u v \text{tt} \equiv u} \text{ 2 } \beta_1 \quad \frac{}{\text{ind}_2 u v \text{ff} \equiv v} \text{ 2 } \beta_2$$

$\text{ind}_2 u v t$ -re úgy lehet gondolni, mint `if t then u else v`.

A kételemű típus és a függő pár segítségével definiálhatjuk az összeg típusot bármely két A és B típusra az alábbi módon:

$$\cdot \vdash \lambda A. \lambda B. \sum_{x:2} (\text{ind}_2 A B x) : \mathbf{U}_i \rightarrow (\mathbf{U}_i \rightarrow \mathbf{U}_i)$$

Bevezetjük az alábbi rövidítést:

$$A + B \equiv \sum_{x:2} (\text{ind}_2 A B x)$$

Az $A + B$ típus elemei vagy `ff` és egy A -beli elem, vagy `tt` és egy B -beli elem. A bal és jobb injekciókat az alábbi rövidítésekkel adhatjuk meg:

$$\text{inl} \equiv \lambda a. (\text{ff}, a)$$

$$\text{inr} \equiv \lambda b. (\text{tt}, b)$$

Szintén definiálható³ rövidítésként az alábbi ind_{A+B} függvény, melyre az $A + B$ eliminátoraként gondolhatunk (ha minden $a : A$ -ra tudjuk

³ ind_{A+B} a következő kifejezést rövidíti: $\lambda ml. \lambda mr. \lambda t. \text{ind}_2 ml mr (\pi_1 t) (\pi_2 t)$, ahol a 2 elim szabályhoz szükséges A típus a következőképp van definiálva: $\prod_{v:\text{ind}_2 A B x} C(x, v)$. Érdemes a β -szabályokat is ellenőrizni.

$C(\text{inl } a)$ -t és minden $b : B$ -re tudjuk $C(\text{inr } b)$ -t, akkor minden $t : A + B$ -re tudjuk $C\ t$ -t):

$$\begin{aligned} & A : \mathcal{U}_i, B : \mathcal{U}_i, C : A + B \rightarrow \mathcal{U}_j \vdash \text{ind}_{A+B} \\ & : \left(\prod_{a:A} C(\text{inl } a) \right) \rightarrow \left(\prod_{b:B} C(\text{inr } b) \right) \rightarrow \prod_{t:A+B} C\ t \end{aligned}$$

$A + B$ -re az $A \vee B$ diszjunkcióként gondolhatunk.

2.5. A természetes számok típusának levezetési szabályai

A természetes számokat Peano-számokként adjuk meg, két konstruktorral: **zero** és **suc**. Az eliminációs szabály a természetes számokon alkalmazott teljes indukciónak felel meg, ez alapján az analógia alapján hívjuk **ind**-nek az eliminátorokat.

$$\frac{}{\Gamma \vdash \mathbb{N} : \mathcal{U}_0} \mathbb{N} \text{ képz.} \quad \frac{}{\Gamma \vdash \text{zero} : \mathbb{N}} \mathbb{N} \text{ bev}_1. \quad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \text{suc } n : \mathbb{N}} \mathbb{N} \text{ bev}_2.$$

$$\begin{aligned} & \Gamma, x : \mathbb{N} \vdash A : \mathcal{U}_i \\ & \Gamma \vdash mz : A[\text{zero}/x] \quad \Gamma, n : \mathbb{N}, p : A[n/x] \vdash ms : A[\text{suc } n/x] \\ & \Gamma \vdash t : \mathbb{N} \\ & \hline & \Gamma \vdash \text{ind}_{\mathbb{N}} mz ms t : A[t/x] \quad \mathbb{N} \text{ elim.} \\ & \frac{}{\text{ind}_{\mathbb{N}} mz ms \text{ zero} \equiv mz} \mathbb{N} \beta_1 \\ & \frac{}{\text{ind}_{\mathbb{N}} mz ms (\text{suc } m) \equiv ms[m/n, \text{ind}_{\mathbb{N}} mz ms m/p]} \mathbb{N} \beta_2 \end{aligned}$$

A természetes számoknak nincs η -szabályuk.

Az összeadást pl. az alábbi kifejezéssel definiálhatjuk (az üres környezetben):

$$\cdot \vdash \lambda x. \lambda y. \text{ind}_{\mathbb{N}} y (\text{suc } p) x : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$$

Az $\text{ind}_{\mathbb{N}}$ második argumentuma az n és p változókkal kibővített környezetben értelmezett, emiatt van értelme p -re hivatkozni a fenti $(\text{suc } p)$ kifejezésben. Bevezetjük az alábbi rövidítést: $x + y \equiv \text{ind}_{\mathbb{N}} y (\text{suc } p) x$.

Ha $x \equiv \text{zero}$, akkor $x + y \equiv \text{ind}_{\mathbb{N}} y (\text{suc } p) \text{zero}$, ami $\mathbb{N} \beta_1$ miatt definicionálisan egyenlő y -al. Ha $x \equiv \text{suc } m$, akkor $x + y \equiv$

$\text{ind}_{\mathbb{N}} y (\text{suc } p) (\text{suc } m)$, mely a $\mathbb{N} \beta_2$ szabály miatt definicionálisan egyenlő $\text{suc} (\text{ind}_{\mathbb{N}} y (\text{suc } p) m)$ -el stb. Így gondolhatunk a program végrehajtására (további részletek a 2.9. pontban).

2.6. Egyenlőség típus

Két kifejezés egyenlőségének leírására szolgál a definicionális egyenlőség következtetés-típus, pl. $u \equiv v$. Ez az egyenlőség azonban nem szerepelhet magukban a kifejezésekben (programokban, típusokban, matematikai állításokban, bizonyításokban). A definicionális egyenlőség internalizálására vezetjük be a (propozicionális) egyenlőséget⁴. Ez egy típus, mely azt fejezi ki, hogy két kifejezés egyenlő.

Levezetési szabályai:

$$\frac{\Gamma \vdash A : \mathbb{U}_i \quad \Gamma \vdash u : A \quad \Gamma \vdash v : A}{\Gamma \vdash u =_A v : \mathbb{U}_i} = \text{képz.} \quad \frac{\Gamma \vdash a : A}{\Gamma \vdash \text{refl } a : a =_A a} = \text{bev.}$$

$$\frac{\begin{array}{l} \Gamma \vdash A : \mathbb{U}_i \\ \Gamma, x : A, y : A, p : x =_A y \vdash C : \mathbb{U}_j \\ \Gamma, a : A \vdash m : C[a/x, a/y, \text{refl } a/p] \\ \Gamma \vdash u : A \quad \Gamma \vdash v : A \quad \Gamma \vdash r : u =_A v \end{array}}{\Gamma \vdash \text{J } C \text{ } t \text{ } r : C[u/x, v/y, r/p]} = \text{elim.}$$

$$\overline{\text{J } C \text{ } t (\text{refl } u) \equiv t[u/a]} = \beta$$

A reflexivitás (refl) az egyetlen módja az egyenlőség bevezetésének. Az eliminációs szabályban C egy olyan típus, amely függ két A -beli elemtől (x és y), és ezek egyenlőségének bizonyításától (p). Az eliminációs szabály pedig azt fejezi ki, hogy ha C -t be tudjuk látni bármely A -beli elemre és refl-re, akkor bármilyen egyenlőségre (r -re) is be tudjuk látni. Néha $u =_A v$ helyett egyszerűen csak $u = v$ -t írunk.

A definicionális egyenlőséggel való kompatibilitást az alábbi szabályok biztosítják:

$$\frac{A \equiv A' \quad u \equiv u' : A \quad v \equiv v' : A}{(u =_A v) \equiv (u' =_{A'} v')} \quad \frac{a \equiv a'}{\text{refl } a \equiv \text{refl } a'}$$

A bal oldali szabályból következik, hogy ha két kifejezés definicionálisan egyenlő, akkor propozicionálisan is egyenlők (internalizálás). Fordítva

⁴A “propozicionális” jelzőt gyakran elhagyjuk.

ez nem igaz: pl. a fenti definícióval az $u : \mathbb{N}, v : \mathbb{N}, w : \mathbb{N}$ környezetbeli $(u + v) + w$ és $u + (v + w)$ kifejezések nem definicionálisan egyenlők, de propozicionálisan igen (lásd lejjebb). Ha viszont tetszőleges környezetben egy egyenlőség bizonyítása egyszerűen $\text{refl } t$ valamely t -re, akkor az egyenlőség két oldalán szereplő kifejezések definicionálisan is egyenlők. Az üres környezetben egyenlő kifejezések mind definicionálisan is egyenlők.

A $\forall f:A \rightarrow B. (u = v \rightarrow fu = fv)$ tételt az alábbi módon tudjuk pl. bebizonyítani:

$$\begin{aligned} A : \mathbb{U}_i, B : \mathbb{U}_i, u : A, v : A \vdash \lambda f. J(fx =_B fy) (\text{refl}(fa)) \\ : \prod_{f:A \rightarrow B} (u =_A v \rightarrow fu =_B fv) \end{aligned}$$

A fenti kifejezésre bevezetjük az **ap** rövidítést.

Az **ap** segítségével bebizonyíthatjuk, hogy a természetes számok fentebb definiált összeadása asszociatív:

$$\begin{aligned} \cdot \vdash \lambda u. \lambda v. \lambda w. \text{ind}_{\mathbb{N}}(\text{refl}(v + w)) (\text{apsuc } p) x \\ : \prod_{u:\mathbb{N}} \prod_{v:\mathbb{N}} \prod_{w:\mathbb{N}} (u + v) + w =_{\mathbb{N}} u + (v + w) \end{aligned}$$

A bizonyítást így írhatjuk le: teljes indukcióval bizonyítunk x szerint. Az $\mathbb{N}$ elim szabályban szereplő x -től függő A típusnak $(x + v) + w =_{\mathbb{N}} x + (v + w)$ -t választunk. Ha x értéke 0 ($x \equiv \text{zero}$), akkor az $\mathbb{N}$ elim szabályban szereplő $mz : (\text{zero} + v) + w =_{\mathbb{N}} \text{zero} + (v + w)$, ennek típusa pedig a $+$ rövidítés behelyettesítésével és a $\mathbb{N} \beta_1$ szabály alapján definicionálisan egyenlő $v + w =_{\mathbb{N}} v + w$ -vel, amit $\text{refl}(v + w)$ bizonyít. Ha $x \equiv \text{suc } n$, akkor az ms környezetében levő p típusa $(n + v) + w =_{\mathbb{N}} n + (v + w)$, ez az indukciós hipotézisünk, ha erre alkalmazzuk az **apsuc** függvényt, $\text{suc}((n + v) + w) =_{\mathbb{N}} \text{suc}(n + (v + w))$ -ot kapunk, ami viszont a $+$ rövidítés behelyettesítésével és $\mathbb{N} \beta_2$ alkalmazásával definicionálisan egyenlő $(\text{suc } n + v) + w =_{\mathbb{N}} \text{suc } n + (v + w)$ -el, és ez az, amit az indukciós lépésben bizonyítani szeretnénk (ms típusa ez).

Egy további érdekes tétel, melyet **transport**-al rövidítünk, szintén egyszerűen bizonyítható **J** segítségével:

$$\begin{aligned} A : \mathbb{U}_i, u : A, v : A \vdash \lambda P. \lambda r. J(\lambda x. \lambda y. \lambda p. P x \rightarrow P y) (\lambda a. \lambda s. s) r \\ : \prod_{P:A \rightarrow \mathbb{U}_j} u =_A v \rightarrow P u \rightarrow P v \end{aligned}$$

transport a P tulajdonság A -ban egyenlő elemek közötti transzportálását fejezi ki: ha $u = v$, és u rendelkezik a P tulajdonsággal, akkor v is.

Szintén a J eliminátor segítségével bizonyítható, hogy egy adott A típusra $_ =_A _ : A \rightarrow A \rightarrow U_i$ ekvivalencia-reláció, refl egységelem, és a tranzitivitás asszociatív:

$\text{refl } a : a = a$	reflexivitás
$_^{-1} : a = b \rightarrow b = a$	szimmetria
$_ \cdot _ : a = b \rightarrow b = c \rightarrow a = c$	tranzitivitás
$p \cdot \text{refl } b = p$	jobb oldali egységelem
$\text{refl } a \cdot p = p$	bal oldali egységelem
$(p \cdot q) \cdot r = p \cdot (q \cdot r)$	asszociativitás

adott $a, b, c, d : A$, $p : a = b$, $q : b = c$, $r : c = d$ esetén.

2.7. Általános induktív típusok

A természetes számok, bináris fák, listák stb. mind induktív típusok. Ezeket funkcionális programozási nyelvekben konstruktoraik listázásával adjuk meg, és eliminálásukra mintaillesztéssel adunk meg függvényeket. Pl. a feljebb definiált természetes számokat Haskell-ben így adjuk meg: `data Nat = Zero | Suc Nat`, Agda-ban ekképp:

```
data Nat : Set where
  zero : Nat
  suc  : Nat -> Nat
```

Az összes induktív típus megadható egy konténerből számított W -típusként [1]. Egy konténert az $S : U_i$ alakjával és a $P : S \rightarrow U_i$ pozícióival adunk meg. Az alak reprezentálja a konstruktorok halmazát a nem-rekurzív paraméterekkel együtt, míg a pozíciók a rekurzív előfordulásokat adják meg. A természetes számok, A -beli elemek listája és leveleknél L -beli elemet, elágazásoknál N -beli elemet tartalmazó bináris fák konstruktoraik ill. a nekik megfelelő konténerek (1 az egyelemű típus):

$\text{zero} : \mathbb{N}$	$S \equiv 2$
$\text{suc} : \mathbb{N} \rightarrow \mathbb{N}$	$P \equiv \lambda s. \text{ind}_2 0 1 s$
$\text{nil} : \text{List}_A$	$S \equiv 1 + A$
$\text{cons} : A \rightarrow \text{List}_A \rightarrow \text{List}_A$	$P \equiv \lambda s. \text{ind}_{1+A} (\lambda x. 0) (\lambda x. 1) s$
$\text{leaf} : L \rightarrow \text{Tree}_{L,N}$	$S \equiv L + N$
$\text{node} : N \rightarrow \text{Tree}_{L,N} \rightarrow \text{Tree}_{L,N} \rightarrow \text{Tree}_{L,N}$	$P \equiv \lambda s. \text{ind}_{L+N} (\lambda x. 0) (\lambda x. 2) s$

Tehát a természetes számoknál $s \equiv \text{ff}$ jelképezi a zero konstruktort, $s \equiv \text{tt}$ a suc konstruktort, és $P \text{ff} \equiv 0$, tehát a zero konstruktornak nincs rekurzív paramétere, míg $P \text{tt} \equiv 1$, tehát a suc konstruktornak egy rekurzív ($\mathbb{N}$ típusú) paramétere van. Listáknál a nil konstruktornak nincs rekurzív paramétere ($s \equiv \text{inl} *$ jelzi, ahol $*$ az egyelemű típus eleme), míg a cons konstruktor igazából A -nyi különböző konstruktor, annak megfelelően, hogy a cons első paramétere micsoda, és függetlenül ettől az értéktől mindig 1 db rekurzív (List_A típusú) paramétere van.

Ha adott egy S, P konténer, a hozzá tartozó WSP típus az alábbi bevezető szabállyal rendelkezik:

$$\frac{\Gamma \vdash s : S \quad \Gamma \vdash f : P s \rightarrow WSP}{\Gamma \vdash \text{sup } s f : WSP} \text{W bev}$$

Tehát ha egy WSP típusú értéket szeretnénk konstruálni, egy formára és egy függvényre van szükség, utóbbi a formának megfelelő összes pozícióra ad egy újabb WSP típusú értéket. Az eliminációs és egyéb szabályok is megadhatók, részletek a W -típusokat bevezető [23]-ban. A konténerek kategória-elméleti megfelelői a szigorúan pozitív funktorok [1], a W -típusok ezek iniciális algebrái (legkisebb fixpontjai). Terminális koalgebráikat M -típusoknak nevezik [8], ezek maximális fixpontoknak felelnek meg, végtelen listák pl. így írhatók le. A konténerek kiterjesztései az indexelt konténerek [28], illetve az ezekhez tartozó indexelt W -típusok, melyekkel indexelt típusok leírhatók, pl. az n -hosszú vektorok típusa, $\text{Vec}_A n$, ahol $n : \mathbb{N}$.

2.8. Curry–Howard izomorfizmus

Az előbb megadott típuselméletet a következőképp használhatjuk logikai érvelésre: a proposíciók halmazának U_0 -t vesszük, egyéb halmazokat típusokkal értelmezzük, pl. a természetes számok halmazát

az $\mathbb{N}$ induktív típussal értelmezzük. Az elsőrendű logikai összekötőket így értelmezzük:

$\text{Prop}^c \equiv \mathsf{U}_0$	(propozíciók halmaza)
$A \rightarrow \text{Prop}^c \equiv A \rightarrow \mathsf{U}_0$	A -n értelmezett állítások halmaza
$\perp \equiv 0$	logikai hamis
$\top \equiv 1$	logikai igaz
$\neg P \equiv P \rightarrow 0$	negáció
$P \wedge Q \equiv P \times Q \equiv \sum_{p:P} Q$	(Q -ban nem szerepel p)
$P \vee^c Q \equiv P + Q \equiv \sum_{x:2} (\text{ind}_2 A B x)$	
$P \Rightarrow Q \equiv P \rightarrow Q \equiv \prod_{p:P} Q$	(Q -ban nem szerepel p)
$\forall_{x \in A} P_x \equiv \prod_{x:A} P$	
$\exists_{x \in A}^c P_x \equiv \sum_{x:A} P$	
$a = b \equiv a =_A b$	(a és b az A típus elemei)

Ez a logikai rendszer néhány fontos különbséget mutat a klasszikus matematikához képest:

- Halmazok helyett típusokat használunk, emiatt nincs értelmezve az unió és metszet művelet, hiszen ezek tetszőleges típusok között sem értelmesek. Minden kifejezésnek van típusa, nincs olyan helyzet, hogy egy elem többféle halmazba (típusba) is tartozhat, a típus valamely kifejezésnek el nem idegeníthető és meg nem változtatható tulajdonsága (ami egyébként elősegíti nemcsak a biztonságos programozást, de a biztonságos bizonyítást és absztrakciót is).
- $\text{Prop}^c \equiv \mathsf{U}_0$ bizonyítás-relevanciával (proof-relevance) rendelkezik: ugyanannak az állításnak több különböző bizonyítása is lehet,

és ezek nem kell, hogy egyenlők legyenek – a bizonyítások információt hordoznak (ennek kiküszöbölésére a típuselméletben gyakran bevezetnek egy külön Prop^c univerzumot, melynek elemei definicionálisan egyenlők). Heyting nyomán egy állításra gondolhatunk úgy, mint bizonyításainak halmazára (típusára).

- A $\exists^c$ kvantor erős (vagy konstruktív, emiatt jelöltük c -vel): ha van egy bizonyításunk arra, hogy létezik A -nak valamely P tulajdonsággal rendelkező eleme, akkor ebből a bizonyításból ezt az elemet mindig képesek vagyunk kinyerni (nem csak a pusztán létezésről tudunk). Hasonlóképpen $P \vee^c Q$ bizonyításából extrahálható, hogy P vagy Q az igaz. A klasszikus változataik megadásához homotópia-típuselmélet szükséges, lásd a 7. pont.
- Ehhez szorosan kapcsolódik, hogy a kizárt harmadik elve nincs validálva, vagyis nem igaz, hogy minden állítás eldönthető. Ha indirekten bizonyítjuk P -t, akkor valójában $\neg\neg P$ -t bizonyítottunk. Mivel a dupla negáció művelet monádként [3] viselkedik, hasonlóan ahhoz, ahogy a Haskell programozási nyelvben IO -t kell írni a mellékhatásokat használó függvények típusa elé, itt $\neg\neg$ -et kell írni a kizárt harmadik elvét használó bizonyítások típusa elé.

2.9. Metaelméleti tulajdonságok

Az ún. helyettesítési és gyengítési szabályok [10] a levezetési fákon végzett indukcióval bizonyíthatók. Hasonlóképp bizonyítható, hogy ha $a \equiv b : A$, akkor mind a -nak, mind b -nek a típusa A (tehát a definicionális egyenlőség megőrzi a típust).

A fent bevezetett formális rendszer azért használható jól programozásra, mert a típusellenőrzés eldönthető. Tehát, ha adott egy Γ környezet, t kifejezés és A típus, akkor létezik egy olyan program, mely eldönti, hogy $\Gamma \vdash t : A$ következtetés levezethető-e⁵. Úgy is fogalmazhatunk, hogy ha van egy matematikai állításunk, és egy jelöltünk ennek bizonyítására, akkor a számítógép felismeri, hogy a bizonyítás helyes-e.

⁵Ehhez az egyes kifejezéseknek több információt kell hordozniuk, mint amennyit informálisan leírunk, pl. a λ kifejezéseket fel kell díszíteni a változó típusával stb. A kényelmes, tömör írásmódból teljes információval rendelkező kifejezéseket létrehozó folyamatot elaborációnak hívjuk, további információ pl. az Agda programozási nyelv működését leíró PhD-dolgozatban [29] vagy az Epigram elaborációját leíró cikkben [25] található.

Továbbá a rendszer konzisztens, amin azt értjük, hogy nem létezik olyan t kifejezés, melyre $\cdot \vdash t : 0$ levezethető lenne. Ezt természetesen csak egy másik, erősebb rendszerhez képest relatíve tudjuk kijelenteni, ilyen rendszer pl. a ZFC megfelelő nagyságú kardinálisokkal.

A konzisztencia bizonyítása normalizáláson keresztül történik. Ha a definicionális egyenlőség szerint ekvivalencia-osztályokat képzünk, minden ilyen osztályból kiválasztható egy reprezentáns, az ún. normálforma, és létezik egy algoritmus, mely bármely kifejezést átalakít az ekvivalencia-osztályának reprezentálására. Ezt a folyamatot normalizálásnak hívjuk. A normalizálás történhet kis lépésekben [13], nagy lépésekben [9], vagy a normalizálás kiértékeléssel technikával [5]. A normálformák úgy vannak meghatározva, hogy az üres környezetben mindig konstruktorral kezdődnek, emiatt, mivel az üres típusnak nincs konstruktora, az üres környezetben nincs 0 típusú kifejezés. Pl. természetes számok normálformái az üres környezetben $\text{suc}^n \text{zero}$, tehát a suc konstruktor n -szer alkalmazva a zero konstruktorra, ahol n egy természetes szám.

A fent megadott típusrendszerre (kivéve általános induktív típusok és az egyelemű típus) a normálformákat v adja meg (a normálformák típusozottak, ettől most eltekintünk):

$$\begin{aligned} v ::= & \text{U}_i \mid \prod_{x:v} v \mid \lambda x.v \mid \sum_{x:v} v \mid (v, v) \mid 0 \mid 2 \mid \text{ff} \mid \text{tt} \mid \mathbb{N} \mid \text{zero} \mid \text{suc } v \mid v =_v v \mid \text{refl } v \\ n ::= & x \mid n v \mid \pi_1 n \mid \pi_2 n \mid \text{ind}_2 v v n \mid \text{ind}_{\mathbb{N}} v v n \mid \text{J } v v n \end{aligned}$$

n az ún. neutrális kifejezéseket jelöli, x egy változó. A neutrális kifejezések bár nem konstruktorral kezdődnek, mégsem egyszerűsíthetők, mert β szabály nem alkalmazható rájuk. Zárt kifejezés nem lehet neutrális, mert nincs benne szabad változó.

3. Extenzionalitás

A fent megadott típuselméletben az egyenlőség túlságosan finom: olyan kifejezéseket is megkülönböztet, melyeket a matematikában nem szeretnénk megkülönböztetni. Ilyenek a pontonként egyenlő, de definíció szerint különböző függvények, pl. a $\lambda x.x : \mathbb{N} \rightarrow \mathbb{N}$ és a $\lambda x.x + \text{zero}$ függvények. Az extenzionalitás elve azt mondja, hogy két objektum nem lehet

egyszerre (a meglevő egyenlőség használata nélkül) megkülönböztethetetlen és nem-egyenlő. Mivel függvényeket csak úgy lehet használni, hogy alkalmazzuk őket, ha tetszőleges kifejezésre alkalmazva őket, egyenlő eredményt adnak, akkor megkülönböztethetetlenek. Ennek megfelelően szeretnénk, hogy az alábbi szabály része legyen a típuselméletünknek:

$$\frac{f : \prod_{x:A} B \quad g : \prod_{x:A} B \quad t : \prod_{a:A} f a =_{B[a/x]} g a}{\text{funext } t : f = \prod_{x:A} B g}$$

Ha azonban ezt, mint új levezetési szabályt hozzáadjuk a rendszerhez, elveszítjük a normalizálást: mivel az egyenlőség β szabálya csak olyankor működik, ha az egyenlőség a **refl** konstruktorral lett létrehozva, ha J -t egy **funext** által létrehozott egyenlőségre alkalmazzuk, elakad a normálformára hozás algoritmus. Az egyenlőség más fajta definíciójával, és egy újabb levezetési szabály, a **K** (lásd később) hozzáadásával megadható egy olyan típuselmélet, ami normalizáló és a függvény extenzionalitást is validálja [2]. Ez az elmélet azonban inkonzisztens a következő extenzionalitási alapelvvel, az izomorf típusok egyenlőségével.

A és B típusok izomorfak, ha léteznek $f : A \rightarrow B$ és $g : B \rightarrow A$ függvények, melyekre igaz, hogy $\prod_{a:A} g (f a) =_A a$ és $\prod_{b:B} f (g b) =_B b$. Ha két típus izomorf, akkor ha egyikben kapunk egy elemet, áttehetjük a másikba, és tudjuk, hogy ez a lépés nem változtatja meg lényegesen az elemet, mert mindig visszakapjuk az eredetit, ha a másik irányú függvényt alkalmazzuk. Ha $A \cong B$ jelöli az A és B közötti izomorfizmusok típusát, akkor szeretnénk egy **isotoid** $: A \cong B \rightarrow A = B$ függvényt. Ha van egy ilyenünk, ez azzal az előnnyel jár, hogy bármely műveletet vagy tulajdonságot, amivel az egyik típus rendelkezik, transzportálni tudunk a másikra. Pl. ha van egy $m : A \rightarrow A \rightarrow A$ műveletünk és egy $i : A \cong B$ izomorfizmusunk, akkor az ennek megfelelő $m' : B \rightarrow B \rightarrow B$ műveletet megkaphatjuk a 2.6 pontban megadott **transport** függvénnyel: $m' \equiv \text{transport } (\lambda X. X \rightarrow X \rightarrow X)$ (**isotoid** i) m .

Ha egy szótárt szeretnénk implementálni, megtehetjük ezt nem-hatékony módon, egy listával (**List**, **_::_**, ...), vagy hatékony módon piros-fekete keresőfákkal (**RBT**, **insert**, ...) (mindkettő a következő egzisztenciális típus eleme: $\sum_{T:U_i} (A \rightarrow T \rightarrow T) \times \dots$). A hatékony implementációról szeretnénk bizonyítani bizonyos tulajdonságokat, pl. hogy kétszer egymás után ugyanazt az **insert** parancsot végrehajtva ugyanazt kapjuk, mint ha csak egyszer tettük volna meg. A hatékony implementációról azonban nehéz formálisan

érvelni. Ha viszont bizonyítunk egy izomorfizmust a hatékony és a nem-hatékony implementáció között, onnantól elég a tulajdonságokat az egyszerű implementációról belátni, és **transport** segítségével ezek mind igazak lesznek a bonyolultra is.

Ha **isotoid**-t egyszerűen levezetési szabályként hozzáadjuk az elméletünkhöz, ugyanabba a problémába ütközünk, mint az előbb, elveszítjük a normalizálást. A típuselméletünk nem válik inkonzisztenssé, mert Voevodsky szimpliciális halmaz modellje [20] validálja ezeket a levezetési szabályokat.

Az extenzionális tulajdonságok intenzionális típuselméletbe való beillesztéséről szól [16] (az izomorfizmusokra nem tér ki).

4. Típusok mint topologikus terek

Hagyományosan a típusokra halmazokként gondolunk, és a Curry–Howard izomorfizmus (2.8. pont) sem változtat ezen, hiszen az egy típust a neki megfelelő állítás bizonyításainak halmazaként értelmezi.

Az egyenlőség típus nem úgy viselkedik, ahogyan azt egy halmaztól elvárnánk. A természetesnek tűnő szabály, miszerint ha van két bizonyításunk a és b egyenlőségére, akkor ez a két bizonyítás (propozicionálisan) egyenlő, a Martin–Löf típuselméletben nem bizonyítható. Ezt Hofmann és Streicher mutatták meg groupoid modelljükkel [18] (a groupoid egy olyan kategória, melyben minden morfizmus izomorfizmus [3]). Ez a szabály ekvivalens az egyenlőség típus ún. K eliminációs szabályával, mely újabb levezetési szabályként hozzávehető a típuselmülethez, és mellé tehető egy β szabály is, amellyel a típuselmélet normalizáló marad. Hogy az induktív típusként megadható egyenlőség normális eliminációs szabálya (J) miért nem elégséges az egyenlőség leírására, sokáig a típuselmélet egy titokzatos tulajdonsága volt.

A groupoid modellt Awodey [4] és tőle függetlenül Voevodsky [34] fejlesztette tovább a típusokat topologikus tereként értelmező modellé. Ha egy típust topologikus térként értelmezünk, a típus elemeit pedig annak pontjaiként, akkor egy $a = b$ típusú kifejezés egy a és b közötti útnak felel meg. Az, hogy $p, q : a = b$ esetén nem feltétlenül igaz, hogy $p = q$, annak felel meg, hogy két út nem feltétlenül egyenlő, vagyis nem feltétlenül igaz, hogy létezik egy homotópia p és q között.

A topologikus tér és topologikus terek közötti folytonos függvény definícióját nem adjuk meg, bármely topológia könyvben megtalálhatóak

ezek a fogalmak. Egy térre intuitíve gondolhatunk úgy, mint pl. egy háromdimenziós objektum belsejére (vagy felszínére stb.), folytonos függvényre pedig mint egy vonalra, melyet ceruzánk felemelése nélkül megrajzolhatunk.⁶ Ezek általánosításai a topológiai alapfogalmak. Igazából nem is érdekes, hogy pontosan mik a definíciók, mert a típuselméletnek egyfajta “szintetikus” topológia felel meg, ahol az alapfogalmak definíciója absztrakt. Emiatt minden definiálható függvény folytonos, és nem tudjuk a terek topológiáját megváltoztatni.

4.1. Definíció (Homotópia (topológiában)). X, Y topologikus terek, az $f, g : X \rightarrow Y$ folytonos függvények közötti homotópia egy folytonos $h : X \times [0, 1] \rightarrow Y$ függvény, melyre minden $x : X$ -re $h(x, 0) = f x$ és $h(x, 1) = g x$. Jelölés: $h : f \sim g$.

Egy homotópiára úgy gondolhatunk, mint f képének folytonos deformálására g képébe.

Egy X topologikus tér két pontja megadható mint $a : 1 \rightarrow X$ és $b : 1 \rightarrow X$ (folytonos) függvények. Egy a és b közötti homotópia így egy $f : 1 \times [0, 1] \rightarrow X$ folytonos függvény lesz, melynek típusa izomorf⁷ $[0, 1] \rightarrow X$ -el, és így $f 0 = a$ és $f 1 = b$ (ahol pongyolán a -t írunk $a *$ helyett, hiszen $(1 \rightarrow X) \cong X$). Típuselméletben f megfelelője egy $f : a = b$ kifejezés. Ezzel adja magát a következő definíció, mely megfelel a topológiai definíciónak.

4.2. Definíció (Homotópia (típuselméletben)). $f, g : \prod_{x:A} B$ függvények közötti homotópiát

$$f \sim g := \prod_{x:A} (f x = g x)$$

definiálja.

Ha adott két folytonos függvény, $f, g : [0, 1] \rightarrow X$, melyek 0-ban a -t, 1-ben b -t vesznek föl, akkor az f és g közötti homotópia egy $h : [0, 1]^2 \rightarrow X$ folytonos függvény, melyre minden $z : [0, 1]$ -re $h(z, 0) = f z$ és $h(z, 1) = g z$. h típuselméleti megfelelője egy $h : (a, b, f) = \sum_{x,y:X} x=y (a, b, g)$, ami nem kifejezetten érdekes (J kétszeri

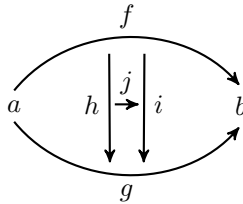
⁶Főként $[0, 1]^n$ értelmezési tartományú függvényekkel fogunk foglalkozni, emiatt az analógia elég jól használható.

⁷A matematikában, mint ebben az érvelésben is, gyakran használjuk az “izomorf típusok egyenlőek” alapelvet. Típuselméleti megfelelőjéért lásd a 6. pontot.

alkalmazásával bármely két ilyen hármas triviálisan egyenlő lesz). Mi lesz tehát egy típuselméletbeli magasabb egyenlőség, $h' : f = g$ topológiai megfelelője?

4.3. Definíció (Relatív homotópia (topológiában)). X, Y topológikus terek, az $f, g : X \rightarrow Y$ folytonos függvények közötti, $A \subseteq X$ halmazhoz relatív homotópia egy olyan $h : f \sim g$, hogy minden $a : A$ -ra a $h(a, t)$ érték független t -től.

Természetesen f és g között relatív homotópia csak akkor adható meg, ha $f a = g a$ minden $a : A$ -ra. Egy relatív homotópiára úgy gondolhatunk, mint f képének folytonos deformálására g képébe úgy, hogy közben az A -beli pontok fixen maradnak. Az előbbi f és g közötti $h' : f \sim g$, $\{0, 1\}$ -re relatív homotópia típuselméletben egy $h' : f = g$ kifejezésnek felel meg.



1. ábra. Egyenlőségek közötti egyenlőségek.

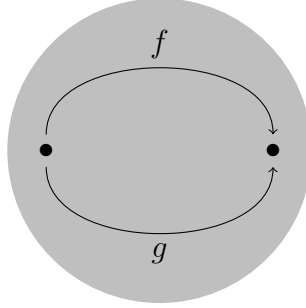
A 1. ábrán látható elrendezésben levő egyenlőségeket típuselméletben ill. topológiában így jelölhetjük:

$$\begin{array}{ll}
 a, b : X & a : 1 \rightarrow X, b : 1 \rightarrow X \\
 f, g : a =_X b & f, g : a \sim b \\
 h, i : f =_{a=b} g & h, i : f \sim g, \text{ melyek } \{0, 1\}\text{-re relatívok} \\
 j : h =_{f=g} i & j : h \sim i, \\
 & \text{mely } \{(t, 0) \mid t \in [0, 1]\} \cup \{(t, 1) \mid t \in [0, 1]\}\text{-re relatív}
 \end{array}$$

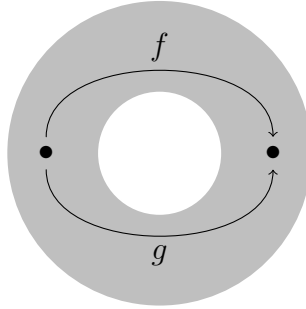
A 2. ábra egy olyan teret (korong) ábrázol, melyben minden pont között van út (minden pont egyenlő), és bármely két út között van $[0, 1]$ -re relatív homotópia (bármely két egyenlőség, pl. az ábrán f -el és g -vel jelölt is, egyenlő).

A 3. ábra egy olyan teret (gyűrű) ábrázol, melyben bár minden pont

között van út (minden pont egyenlő), de az ábrán jelölt f, g utak között nincs $[0, 1]$ -re relatív homotópia ($f = g$ -nek nincs eleme).



2. ábra. Korong.



3. ábra. Gyűrű.

A $\text{refl } a : a =_X a$ kifejezésnek a konstans út $(\lambda t. a : [0, 1] \rightarrow X)$ felel meg; a $p : a = b$ -ből kapott $p^{-1} : b = a$ egyenlőségnek a p -nek megfelelő homotópia és a $\lambda t. 1 - t : [0, 1] \rightarrow [0, 1]$ függvény kompozíciója; a tranzitivitás az alábbi homotópiával adható meg (ha adott f, g , melyekre $f \cdot 1 = g \cdot 0$):

$$(f \cdot g)(z) := \begin{cases} f(z * 2) & z < \frac{1}{2} \\ g((z - \frac{1}{2}) * 2) & \text{egyébként} \end{cases}$$

5. Homotópia-szintek

Léteznek olyan típusok, melyekre teljesül, hogy bizonyos szinttől kezdve az egyenlőségeik triviálisak. Pl. a fentebb mutatott korong típus bármely két pontja között van egy egyenlőség, és ezek mind egyenlők. A gyűrű típus bármely két pontja között van egyenlőség, de azok nem feltétlenül egyenlők, de ha igen, akkor csak egyféleképpen lehetnek azok. Azok a típusok, melyeket halmaznak tekinthetünk, azzal a tulajdonsággal rendelkeznek, hogy két pont között vagy van egyenlőség, vagy nincs, de két nem-egyenlő egyenlőség nem fordulhat elő. A fentieket általánosítják a homotópia-szintek.

5.1. Definíció (Kontraktibilitás). *Egy X típus kontraktibilis, ha az*

$$\text{isContr } X := \sum_{x:X} \left(\prod_{x':X} x = x' \right)$$

típusnak van eleme.

5.2. Definíció (Homotópia-szint). *Ha $n \geq -2$, a homotópia szinteket a következőképp definiáljuk:*

$$\text{is-}n\text{-type} : \mathcal{U}_i \rightarrow \mathcal{U}_i$$

$$\text{is-}(-2)\text{-type} \equiv \text{isContr}$$

$$\text{is-}(n+1)\text{-type} \equiv \lambda X. \prod_{x,x':X} \text{is-}n\text{-type}(x = x')$$

Tehát a -2 . homotópia szinten vannak a kontraktibilis típusok. A -1 . szinten levő típusokat propozícióknak nevezzük: 0 vagy 1 elemük van. Az ilyen típusok csak annyi információt hordoznak, hogy van-e elemük vagy nincsen (bebizonyíthatóak-e vagy sem); ők felelnek meg a matematikai logikában szokványos (bizonyítás-irreleváns) állításoknak. Ha a Curry–Howard izomorfizmust ilyen állításokra szorítjuk meg, a klasszikus logikához közelebb álló logikát kapunk, mint a 2.8. pontban megadott logika. Ehhez szükséges a logikai összekötők propozicionális csonkítása, lásd a 7. pontban. Egyébként maga $\text{is-}n\text{-type } X$ minden X típusra propozicionális. A homotópia-szintek kumulatívák, tehát ha egy X típusra $\text{is-}n\text{-type } X$ igaz, akkor $\text{is-}(n+1)\text{-type } X$ is.

A 0. szinten levő típusokat halmazoknak nevezzük. Az 1. szinten levőket groupoidoknak, a 2. szinten levőket 2-groupoidoknak stb.

Ezen szintek rendszere csak egy új fajta csoportosítása a Martin-Löf típuselmélet típusainak, egyelőre nem vezettünk be semmilyen új levezetési szabályt.

Az előbbi példák közül a korong típus kontraktibilis, -2 . szinten levő típus. A gyűrű groupoid, az egyenlőségeinek egyenlőségei propozíciók. A legtöbb, programozásban használt adattípus (pl. 2 , $\mathbb{N}$, természetes számok listája) halmaz, melynek egyenlőségei pozicionálisak. Bármely, konténerből W -val képzett típus is halmaz. Bizonyítható, hogy Π és Σ megőrzi a homotópia-szinteket: ha $\text{is-}n\text{-type } X$ és $\prod_{x:X} \text{is-}n\text{-type } Y$, akkor $\text{is-}n\text{-type } (\sum_{x:X} Y)$; amennyiben $\prod_{x:X} \text{is-}n\text{-type } Y$, akkor $\text{is-}n\text{-type } (\prod_{x:X} Y)$ (nem kell, hogy X n -szintű legyen). Ezzel belátható, hogy az összes, U_0 -beli típus halmaz (ha magasabb induktív típusokat nem engedünk meg, lásd 7. pont).

A homotópia-szinteket nem szabad összetéveszteni az univerzum-szintekkel. Ez két, majdnem teljesen független dimenzió: a homotópia-szintek a magasabb egyenlőségekkel kapcsolatosak, míg az univerzumok a Russel-paradoxon és változatainak elkerülésére lettek bevezetve.

Az eddig bemutatott összes típus (a gyűrűn kívül, melyet még formálisan nem definiáltunk) halmaz. Magasabb egyenlőségekhez meg kell változtatunk az eddig használt Martin-Löf típuselméletet: a homotópia-típuselmélet új levezetési szabályokkal bővíti a típuselméletet, az ekvivalenciákra vonatkozó unvalenciával és a magasabb induktív típusokra vonatkozó szabályokkal. A homotópia-típuselmélet matematikusoknak szóló, összefoglaló tankönyve, mely az összes, ebben az írásban taglalt szempontra kitér, [31].

6. Ekvivalencia

Ahogy a 3. pontban írtuk, szeretnénk, ha izomorf típusok egyenlőek lennének. A két típus közötti izomorfizmusok típusa azonban nem propozicionális, emiatt egy további koherencia feltétellel egészítjük ki azt, s így jutunk az ekvivalencia definíciójához, mely a homotópia-ekvivalencia definíciójára hajaz:

6.1. Definíció (Ekvivalencia). *Adott $A, B : U_i$. Azt mondjuk, hogy egy $f : A \rightarrow B$ függvény ekvivalencia, ha*

$$\text{isEquiv } f := \sum_{g:B \rightarrow A} \sum_{\alpha:g \circ f \sim \text{id}_A} \sum_{\beta:f \circ g \sim \text{id}_B} \text{ap } f(\alpha x) = \beta(f x)$$

ahol id_X az X típuson értelmezett $\lambda x.x$ identikus függvény, $— \circ —$ pedig a szokásos függvény-kompozíció.

Az alábbi rövidítést használjuk:

$$A \simeq B := \sum_{f:A \rightarrow B} \text{isEquiv } f$$

$A \simeq B$ bármely A, B -re propozicionális típus. Ha van egy izomorfizmusunk, mindig képezhetünk belőle egy ekvivalenciát, tehát a fenti ötös ötödik tagját megkaphatjuk az első négyből.

Minden $A, B : \mathcal{U}_i$ -re definiálható egy $\text{idtoeqv} : A = B \rightarrow A \simeq B$ függvény. Az ekvivalencia $f : A \rightarrow B$ tagja $\text{transport}(\lambda x. \mathcal{U}_i)$ -ként adható meg, az egyenlőségek J-vel bizonyíthatók.

6.2. Definíció (Univalence). Azt mondjuk, hogy egy $\mathcal{U}_i$ univerzum univalens, ha

$$\prod_{A, B : \mathcal{U}_i} \prod_{p : A = B} \text{isEquiv}(\text{idtoeqv } p)$$

Más szavakkal: minden $A, B : \mathcal{U}_i$ -re $(A = B) \simeq (A \simeq B)$.

Ha a Martin-Löf típuselmélethez axiómaként hozzávesszük, hogy minden $\mathcal{U}_i$ univerzum univalens, nem csak az izomorf típusok egyenlőségét, hanem a függvény extenzionalitást is validáljuk [31] a 3. pontban megadott extenzionalitással kapcsolatos tulajdonságok közül.

Ahogy inductív típusokra a refl konstruálja az egyenlőségeket, az univerzumokra a univalence teszi ugyanezt. Ezzel új egyenlőségeket vezet be a meglévők mellé az univerzumokban: a $2 \simeq 2$ típusnak két eleme van, az egyikhez $f = id_2$, a másikhoz $f = \lambda x. \text{not } x$ tartozik, ahol a not a boolean negáció. Ez a két izomorfizmus nem egyenlő, ha egyenlő lenne (ahogy az K-ből következne⁸), ellentmondásra jutnánk.

A univalence axióma az alábbi, naív kizárt harmadik elvével is inkompatibilis:

$$\text{LEM}_\infty := \prod_{A : \mathcal{U}_i} A + \neg A$$

Azonban, ha az eldönthetőséget megszorítjuk a propozíciókra, ahogyan a klasszikus matematikában teszik, kompatibilis axiómát kapunk, ami lehetővé teszi a klasszikus érvelést:

$$\text{LEM} := \prod_{A : \mathcal{U}_i} \text{is-}(-1)\text{-type } A \rightarrow A + \neg A$$

⁸Emiatt K inkompatibilis a univalence axiómával.

7. Magasabb induktív típusok

Az induktív típusokat (2.7. pont) konstruktoraikkal adjuk meg; ha tekként értelmezzük őket, akkor a konstruktorok pont-konstruktoroknak felelnek meg, és természetesen adódik az általánosítás, hogy lehessen út-konstruktorokat (egyenlőség-konstruktorokat) is megadni. Az $\mathbb{S}^1$ típust pl. az alábbi konstruktorok adják meg:

$$\begin{aligned} \text{base} &: \mathbb{S}^1 \\ \text{loop} &: \text{base} =_{\mathbb{S}^1} \text{base} \end{aligned}$$

Ez a korábban bemutatott gyűrű (vagy kör) típus, melynek egy pontja van (**base**), és egy nemtriviális (nem-refl) hurok **base**-ből **base**-be.

Magasabb induktív típusoknak azokat az induktív típusokat nevezzük, melyeknek (magasabb) egyenlőség-konstruktorai is vannak. A konténerek elmélete egyelőre még nincs kiterjesztve a magasabb induktív típusokra, és általánosságban nem adható meg egy ilyen típus eliminátora, de $\mathbb{S}^1$ -nek pl. a következő az eliminátora:

$$\frac{\begin{array}{l} \Gamma, x : \mathbb{S}^1 \vdash P : \mathbf{U}_i \\ \Gamma \vdash b : P[\text{base}/x] \\ \Gamma \vdash l : \text{transport } P \text{ loop } b =_{P[\text{base}]} b \\ \Gamma \vdash t : \mathbb{S}^1 \end{array}}{\Gamma \vdash \text{ind}_{\mathbb{S}^1} b l t : P[t/x]}$$

Tehát, ha adott a P család egy b eleme **base**-nél, és egy l egyenlőség b és b között, mely P -ben **loop** fölött helyezkedik el, akkor bármely $t : \mathbb{S}^1$ -re kapunk egy $P[t/x]$ elemet.

A β szabályok $\text{ind}_{\mathbb{S}^1} b l \text{base} \equiv b$ és $\text{apd } (\lambda x. \text{ind}_{\mathbb{S}^1} b l x) \text{loop} = l$. A második szabály egy propozicionális számítási szabály, mely az **ap** függvény függő típusú függvényekkel is működő változtatával van megadva.

A magasabb induktív típusok arra is használhatók, hogy egy meglevő típust valamilyen homotópia-szintre csonkítsunk. Pl. a propozicionális csonkítás propozíciót képez valamely típusból, tehát azon kívül, hogy a típusnak van-e eleme, minden információt elfelejt, melyet a típus eredetileg hordozott.

7.1. Definíció (Propozicionális csonkítás). *Adott A típusra az*

$||A||$ típus egy magasabb induktív típus az alábbi konstruktorokkal:

$$\begin{aligned} &|_| : A \rightarrow ||A|| \\ &\text{prop-eq} : \prod_{x,y: ||A||} x =_{||A||} y \end{aligned}$$

Ebből a típusból akkor tudunk eliminálni (akkor tudunk információt kinyerni belőle), ha biztosítjuk, hogy a g függvény, melyet erre használunk, nem tesz különbséget a típus különböző elemei között:

$$\frac{\begin{array}{l} \Gamma, x : ||A|| \vdash P : \mathbf{U}_i \\ \Gamma \vdash g : \prod_{x:A} P \\ \Gamma, a, a' : ||A||, u : P[a/x], u' : P[a'/x] \vdash \\ \quad q : \text{transport } P (\text{prop-eq } a a') u = v \\ \Gamma \vdash t : ||A|| \end{array}}{\Gamma \vdash \text{ind}_{||A||} g q t : P[t/x]}$$

q azt fejezi ki, hogy a P indexelt család tagjai mind egyenlőek: tetszőleges $u : P[a/x]$ -re és $u' : P[a'/x]$ -ra u egyenlő u' -vel, de mivel a típusuk különbözik, transzportálnunk kell közöttük.

A propozicionális csonkítással kifejezhető a klasszikus diszjunkció és a klasszikus egzisztenciális kvantor, így a 2.8. pontban megadott logikai összekötők kiegészíthetők az alábbiakkal:

$$\begin{aligned} \text{Prop} &:= \sum_{P:\mathbf{U}_i} (\text{is-}(-1)\text{-type } P) \\ P \vee Q &:= ||P + Q|| \\ \exists_{x \in A} P_x &:= ||\sum_{x:A} P|| \end{aligned}$$

Mivel a függvény típusnál elég, ha az értékészlet propozíció, a függvény típus maga is propozíció lesz, azt nem szükséges csonkítani ahhoz, hogy jól működjön a homotópia-szinttel megadott propozíciókkal.

A magasabb induktív típusok egy további alkalmazása a típus valamely ekvivalencia-reláció szerinti osztályfelbontása; ennek az extrém használata a propozicionális csonkítás is, ahol mindent egy osztályba sorolunk. A természetes számok használatával az egész számok pl. az alábbi konstruktorokkal jellemzett magasabb induktív típusként adhatók

meg:

$$\text{minus} : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{Z}$$

$$\text{quot} : \prod_{a,b,c,d:\mathbb{N}} a + d = c + b \rightarrow \text{minus } a \, b =_{\mathbb{Z}} \text{minus } c \, d$$

$$\text{set} : \prod_{(x,y:\mathbb{Z})} \prod_{(p,q:x=_{\mathbb{Z}}y)} p =_{x=_{\mathbb{Z}}y} q$$

Az utolsó konstruktor azt biztosítja, hogy nincsenek magasabb egyenlőségeink, tehát $\mathbb{Z}$ egy halmaz.

8. Homotópia-típuselmélet

A homotópia-típuselmélet az intenzionális Martin-Löf típuselmélet 2. pontban ismertetett szabályai mellé hozzáveszi a univalence axiómát (6. pont) és a magasabb induktív típusok bevezetését lehetővé tevő szabályokat (melyek még nincsenek formalizálva, néhány példa a 7. pontban). Ezzel a típuselméleten belül használható (propozicionális) egyenlőség kényelmesebbé válik: pontonként egyenlő függvények egyenlőek, ahogyan izomorf típusok is azok. A propozíciók, bizonyítás-releváns állítások vagy halmazok külön homotópia-szintekbe különülnek, és elkülöníthető az információt nem hordozó, klasszikus $\exists$ kvantor a konstruktív Σ -tól; tehát probléma nélkül használható a klasszikus érvelés: ha azt bizonyítjuk, hogy két egész számnak létezik legnagyobb közös osztója, akkor a konstruktív σ -t használjuk, ha analízist formalizálunk, a klasszikus (propozicionálisan csonkított) $\exists$ -et.

A homotópia-típuselmélet szimpliciális halmaz modellje [20] által tudjuk, hogy ezek az új levezetési szabályok nem teszik inkonzisztenssé a típuselméletet. Azonban a 3. pontban leírtakhoz hasonlóan a típuselmélet elveszíti normalizáló tulajdonságát. Konstruktív modellek használatával [6] a normalizálás visszanyerhető, és reményeink szerint a közeljövőben a típuselméletet közvetlenül is ki tudjuk olyan szabályokkal egészíteni, melyek normalizálónak teszik azt. Ehhez az egyenlőség 2.6. pontbeli definíciója helyett valószínűleg egy rekurzív definícióra van szükség, mely típuskonstruktoroktól függően határozza meg, hogy az adott típus egyenlősége hogyan van megadva: pl. függvények egyenlősége pontbeli egyenlőség, párok egyenlősége egyenlőségek párja, univerzumok egyenlősége ekvivalencia stb.

További részletek a homotópia-típuselmélet összefoglaló tankönyvében [31] és a <http://homotopytypetheory.org> weboldalon található.

Hivatkozások

- [1] M. Abbott, T. Altenkirch, N. Ghani, Containers - constructing strictly positive types, *Theoretical Computer Science*, 342 (2005), pp. 3–27.
- [2] T. Altenkirch, C. McBride, W. Swierstra, Observational equality, now!, *PLPV '07: Proceedings of the 2007 workshop on Programming languages meets program verification*, ACM, 2007., pp. 57–58.
- [3] S. Awodey, *Category Theory*, Oxford Logic Guides, OUP Oxford, 2010.
- [4] S. Awodey, M. A. Warren, *Homotopy theoretic models of identity types*, 2007.
- [5] U. Berger, H. Schwichtenberg, An inverse of the evaluation functional for typed λ -calculus, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press, 1991., pp. 203–211.
- [6] M. Bezem, T. Coquand, S. Huber, *A model of type theory in cubical sets*, Unpublished, 2013.
- [7] E. Brady, Idris, a general-purpose dependently typed programming language: Design and implementation, *J. Funct. Program.*, 23(5) (2013), pp. 552–593.
- [8] V. Capretta, Coalgebras in functional programming and type theory, *Theoretical Computer Science*, 412(38) (2011), pp. 5006–5024.
- [9] J. Chapman, Type theory should eat itself, *Electron. Notes Theor. Comput. Sci.*, 228 (2009), pp. 21–36.
- [10] Z. Csörnyei, *Bevezetés a típusrendszerek elméletébe*, ELTE Eötvös Kiadó, 2012.

-
- [11] J-Y. Girard, Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types, 63 (1971), pp. 63–92.
 - [12] J-Y. Girard, The system F of variable types, fifteen years later, *Theor. Comput. Sci.*, 45(2) (1986), pp. 159–192.
 - [13] J-Y. Girard, P. Taylor, Y. Lafont, *Proofs and types*, Cambridge University Press, 1989.
 - [14] G. Gonthier, A computer-checked proof of the Four Colour Theorem, 2005.
 - [15] R. Harper, *Practical Foundations for Programming Languages*, 2009.
 - [16] M. Hofmann, *Extensional Concepts in Intensional Type Theory*, Thesis, University of Edinburgh, Department of Computer Science, 1995.
 - [17] M. Hofmann, Syntax and semantics of dependent types, *Semantics and Logics of Computation*, Cambridge University Press, 1997., pp. 79–130.
 - [18] M. Hofmann, T. Streicher, The groupoid interpretation of type theory, *Venice Festschrift*, Oxford University Press, 1996., pp. 83–111.
 - [19] W. A. Howard, The formulas-as-types notion of construction, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, Academic Press, 1980., pp. 479–490.
 - [20] C. Kapulkin, P. L. Lumsdaine, V. Voevodsky, The simplicial model of univalent foundations, 2012.
<http://arxiv.org/abs/1211.2851>/arXiv:1211.2851.
 - [21] N. Kraus, Non-normalizability of cauchy sequences, Unpublished, 2014.
 - [22] P. Martin-Löf, An intuitionistic theory of types: predicative part, *Proceedings of the Logic Colloquium '73, Studies in Logic and the Foundations of Mathematics*, 80 (1975), pp. 73–118.

- [23] P. Martin-Löf, *Intuitionistic type theory*, *Studies in Proof Theory*, 1, Bibliopolis, 1984.
- [24] The Coq development team, *The Coq proof assistant reference manual*, Version 8.0., LogiCal Project, 2004.
- [25] C. McBride, J. McKinna, The view from the left, *J. Funct. Program.*, 14(1) (2004), pp. 69–111.
- [26] R. Milner, M. Tofte, R. Harper, D. MacQueen, *The Definition of Standard ML*, MIT Press, 1997.
- [27] J. C. Mitchell, G. D. Plotkin, Abstract types have existential type, *ACM Trans. Program. Lang. Syst.*, 10(3) (1988), pp. 470–502.
- [28] P. Morris, T. Altenkirch, Indexed containers, *Twenty-Fourth IEEE Symposium in Logic in Computer Science (LICS 2009)*, 2009.
- [29] U. Norell, *Towards a practical programming language based on dependent type theory*, PhD thesis, Chalmers University of Technology, 2007.
- [30] S. Peyton Jones, J. Hughes and al., Haskell 98 – A non-strict, purely functional language, <http://www.haskell.org/definition/>, February 1999.
- [31] The Univalent Foundations Program, Homotopy type theory: Univalent foundations of mathematics, Technical report, Institute for Advanced Study, 2013.
- [32] J. C. Reynolds, Types, abstraction and parametric polymorphism, *Proceedings of the IFIP 9th World Computer Congress*, Elsevier, 1983., pp. 513–523.
- [33] A.S. Troelstra, D. van Dalen, *Constructivism in Mathematics: An Introduction*, *Studies in logic and the foundations of mathematics*, North-Holland, 1988.
- [34] V. Voevodsky, A very short note on the homotopy λ -calculus, http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/Hlambd_short_current.pdf, 2006.

A fordítóprogramokról

Kovács Györgyi

Eötvös József Collegium*

kovacsgyorgyi0528@gmail.com

A számítógépes nyelvészet, és ezen belül a gépi fordítás interdiszciplína, a nyelvészetet, az informatikát és a matematikát köti össze. Ha a szöveget nemcsak tárolni és megjeleníteni kell, hanem fel kell ismerni a benne lévő nyelvi szerkezeteket is, belépnek a nyelvtechnológia eszközei [1]:171–173. A gépi fordítást három részre szokás osztani. Az első a teljesen automatizált gépi fordítás, amivel a továbbiakban foglalkozni szeretnék. A második a géppel támogatott fordítás, ami két különböző rendszert foglal magában: az ember által támogatott gépi fordítást, amikor az ember és a gép interaktív kapcsolatban vannak a fordítás során, és a gép által támogatott emberi fordítás, amikor egy számítógépes rendszer van az ember segítségére a fordítás során, és nem egy szótár. Az utolsót alkotják a terminológiai adatbankok, melyek egy adott szakterület szókincsét tartalmazzák [2]:401.

„*Valódi gépi fordítás alatt (machine translation) azt értjük, hogy a számítógépes program a forrásszöveg mondatait tulajdonképpen felügyelet nélkül, a felhasználói interakció kihagyásával fordítja a célnyelvre*” [4]:277. A gépi fordítás ma egyik élő nyelvről fordít mondatokat egy másik élő nyelvre egy program segítségével, a szöveg hosszúságától függően több vagy kevesebb idő szükséges hozzá, de mindenképpen az emberi fordításhoz képest ez elenyészően kevés.

A gépi fordítás eredménye azonban nem tökéletes, szükséges, hogy egy ember is átnézze és kijavítsa utána a szöveget, ami viszont sok időt igényel. Ennek oka, hogy az emberi fordítás mögött ott van a nyelvhez a nyelvi elemek jelentése útján kapcsolódó kognitív háttér.

*2009–

Ezért nem is alkalmas irodalmi szövegek fordítására, mert az sok kulturális háttérismeretet igényel, szakmai szövegeket, melyek jóval kötöttebbek, könnyebben és kevesebb hibával fordít [4]:277–278. A gépi fordító rendszerek jelentették eleinte az egyetlen nyelvészeti szoftvert, később pedig a nyelvészeti szoftverek legspeciálisabbjává váltak, hiszen rengeteg bonyolult elemző és generáló részrendszert tartalmaznak [2]:401. Különösen a gyorsan fejlődő tudományágak, például az informatika terén szokás, hogy egy újonnan megjelent szakirodalmat fordítóprogrammal fordítanak le, és ezt emberek igazítják ki fejezetekre leosztva, mert önmagában, gép nélkül az emberi fordítás túlságosan lassú lenne.

1. A fordítóprogramok története

A gépi fordítás születését a második világháború utáni időszaktól számítjuk, ekkor jelentek meg a végrehajtására ténylegesen is alkalmas szerkezetek, a számítógépek. Ekkor lehetségesnek tartották a háborúbeli kódmegfejtő programok továbbfejlesztésével a gépi fordítást, ez az elképzelés Weaver nevéhez kötődik, a géppel való fordítást is dekódolásnak fogta fel. Ez az első gépi fordítási időszak (körülbelül 1946 és 1954 között), amit mai értelemben nem is nevezhetünk fordításnak, inkább csak többnyelvű szótárban való keresésnek [3].

Ezt váltotta fel az úgynevezett „optimista” korszak, ez már alkalmazta olykor a szerkezeti megfelelést. 1954-ben az amerikai Georgetown Egyetemen bemutatták az első gépi fordítást, a rendszer egyszerű orosz mondatokat fordított angolra 6 szabály és egy 250 szavas szótár segítségével. Célja annak bizonyítása volt, hogy a gépi fordítás lehetséges, az alapelvei világosak, csak technikai jellegű munkára van szükség a jó minőségű fordítások előállításához. Ebben az időszakban az Egyesült Államokban 17 különböző intézmény összesen 20 millió dollárt költött ennek a kutatására, a várakozásokkal ellentétben azonban nem került sor ugrásszerű fejlődésre [3].

1964-ben létrejött az ALPAC, az amerikai nyelvfeldolgozás tanácsadó bizottsága, aminek 1966-os jelentése szerint a gépi fordítás lassabb és pontatlanabb az emberi fordításnál, de legalább még egyszer annyiba kerül, ezért nem javasolják az ez irányú kutatások további támogatását. Ebben az időben a fordítóprogramok stratégiája a közvetlen fordítás volt [3].

Az ALPAC jelentése ellenére a kutatás tovább folytatódott, és egyre

jobb minőségű, a gyakorlatban is alkalmazható rendszerek születtek, közülük az egyik legjelentősebb a SYSTRAN. Az 1960-as években megjelentek a közvetítőnyelvre épülő első rendszerek, majd annak a hibáit próbálták kiküszöbölni a transzfer alapú rendszerek, ilyen például a TAUM. A gépi fordítás egyre népszerűbb lett, amit az is bizonyít, hogy a közvetlen fordítást alkalmazó SYSTRAN átállt a transzfer alapú módszerre, így más nyelveket is be tudott vonni a fordító hálózatba. Megjelentek az úgynevezett résznyelvek, azaz a korlátozott szintaxisú és szemantikájú nyelvrészletek [3].

A 80-as évek elején kezdődött az EUROTRA-projekt, az Európai Közösség országainak nyelvei között tetszőleges fordítást biztosító, transzfer alapú rendszer kidolgozása, melyhez felhasználták a szemantika és a mesterséges intelligencia kutatásának legújabb eredményeit. Az EU nem látta elég sikeresnek az EUROTRA-t, és leállt a támogatásával, ezért nem fejeződtek be a munkálatok [3]. 1984-ben összesen félmillió oldalt fordítottak le számítógépes rendszerek segítségével [2]:407.

2. A fordítóprogramok osztályozása

Megkülönböztetünk produktív és minta-alapú fordítóprogramokat. A produktív fordítóprogram maga szintetizálja a célnyelvi mondatot, a minta-alapú fordítóprogram kikeresi a forrásnyelv mondatai közül a leghasonlóbbat, és annak tárolt fordítását adja elő, minimális módosítással. Ma minden kereskedelembe kapható fordítórendszer produktív [4]:278–279.

Az átváltási művelet absztrakciós szintje szerint háromféle produktív fordítási technikát különböztetünk meg:

- közvetlen (direct),
- közvetítőnyelves (interlingual),
- transzfer (transfer).

A közvetlen fordítás kizárólag a forrásnyelv és a célnyelv egyedi tulajdonságaira épül. A szintaktikai elemzés az azonos alakú szavak azonosítására szolgált. Szemantika a mai értelemben nem is volt a rendszerben, csak néhány szemantikai jellegű jegy a már formalizált mondatokban [3].

A közvetítőnyelves modell a hatvanas évek terméke. Itt a forrásnyelvi szöveg analízise és a célnyelvi szöveg szintézise teljesen elválik egymástól, a rendszer a forrásnyelvi szöveget úgynevezett közvetítő nyelvre fordítja le, és a közvetítő nyelvből állítja elő a célnyelvi szöveget. Az elemző és generáló komponensek függetlenek egymástól, a rendszer külön forrásnyelvi és célnyelvi forrásokat tartalmaz, ennek a modellnek az a célja, hogy további nyelveket a meglevő stratégiák módosítása nélkül lehessen a rendszerbe kapcsolni. A közvetítőnyelv elsősorban szintaktikai szerkezet, szemantikai elemek beépítésére csak kevés példa volt. A szintaktikai szerkezet azonban gyakran többértelmű, ezért könnyen fordíthatott mellé. Az analízáló folyamat bármely szintjén végrehajtott rossz alternatívaválasztás pedig kihatott az összes további szintre [3].

A transzfer stratégiában a forrásnyelv és a célnyelv önálló, egymástól független mélyszerkezeti reprezentációkkal rendelkezik, ezért a fordítás három lépésből áll:

- analízis,
- transzfer,
- szintézis.

A szintaktikai elemzés itt nem olyan mély, mint a közvetítőnyelves fordítások esetében, hisz az ott tárolandó információk egy részét a transzfer fázis viszi a rendszerbe [3].

A gépi fordítórendszereket másképpen is lehet osztályozni, megkülönböztethetünk szabály-alapú és statisztika-alapú rendszereket. A szabály-alapú rendszerek jellemzése: „a számítógép programjába olyan szabályokat írnak, amelyek az ember nyelvi vagy nyelvészeti tudását tükrözik, leképezve a számítógép programozási nyelveinek lehetőségeire.” „Ekkor a számítógépes nyelvész a saját nyelvérzéke vagy nyelvészeti tudása – megfelelő forrásmunkák – alapján fogalmazza meg a szabályokat. A szabályok gépi megfogalmazása általában többé-kevésbé megfelel valamelyik matematikai nyelvmodellnek.” Előzetes hipotézist tartalmaz arról, milyen szerkezetek lehetnek a szövegben [1]:174. A szabály-alapú rendszerek az esetek kisebb részét kezelik, azokat viszont hibátlanul (kis fedés, nagy pontosság – (low recall, high precision) [3]. A statisztika-alapú rendszerek esetében a számítógépes nyelvész nem ad előzetes tudást a számítógépnek, a gépnek kell felismernie a szövegben megjelenő szabályosságokat, ismétlődő mintákat, és ezt statisztikai számításokkal

érik el, az eredményt gyakran formalizált nyelvészeti információvá alakítják. Azt vizsgálja, milyen jelenségek vannak a szövegben, ebből fogalmaz meg szabályszerűségeket. A szabály-alapú és statisztika-alapú rendszerek közti különbséget Kenesei István a performancia és kompetencia közti különbséggel azonosítja [1]:174. A statisztika-alapú rendszerek minden esetet igyekeznek kezelni, de triviális esetekben is hibázhatnak, jellemzőik a nagy fedés és a kis pontosság (high recall, low precision) [3].

Fordítási minták, azaz szinkronizált szövegpárok nemcsak a statisztikai, hanem a szabály-alapú rendszerekben is alkalmazhatóak: a szabályok létrehozása történhet minták alapján, például az induktív logikai programozás eszközeivel. A szabály-alapú rendszerek alternatívái lehetnek a példa-alapú rendszerek, ez azon a felfedezésen alapul, hogy a rendszerek minőségét a valós életből vett példák segítségével lehet javítani, nem a szabályok további aprólékos részletezésével [3].

Azóta nem jött létre olyan módszertan, melynek segítségével tovább lehetne lépni, a ma hozzáférhető fordítóprogramok általában ezeket a módszereket használják. Ma a magáncégek általában kiválasztanak egy nyelvpárt, és erre szakosodva próbálnak meg elfogadható eszközöket produkálni, ilyen például az orosz-angol ProMT, a dán-angol PaTrans fordítóprogram, vagy a Kielikone cég finn-angol fordítórendszere [3]. Magyarországon a MorphoLogic céghez kapcsolható a MetaMorpho angol-magyar gépi fordítórendszer. Kenesei István szerint: „A közeljövőben arra számíthatunk, hogy néhány nyelvpárhoz megjelennek jó minőségű gyorsfordítók, amelyek mindig akkora egység fordítását jelenítik meg, amekkorát a nyelvtani elemzőjük még felismert. Azonban nem várható, hogy ilyen programok tetszőleges nyelvpárhoz rendelkezésre álljanak.” [1]:187.

3. Zátonyok közt

A továbbiakban az angol-magyar gépi fordítás lehetőségeit vizsgálom. A kutatáshoz a webforditas.hu szövegfordítóját használtam fel. Kiválasztottam egy angol nyelvű, viszonylag egyszerű nyelvezetű könyvet, Agatha Christie-től a *Zátonyok közt*-et, és a mondatait elkezdtem egyesével lefordítani magyarra. Magyarról aztán visszafordítottam angolra, és mellette megnéztem, hogy a magyar kiadásban hogyan szerepel.

A cím:

Taken at the flood

Magyarra fordítva:

Vett az árvíznél

Vissza:

Had bought the flood

Irodalmi fordítás:

Zátonyok közt

A magyar fordítás szó szerinti: a *taken* a *take* ige harmadik alakja, ezt vagy befejezett melléknévi igenévnek, vagy múlt idejű igének fordította a program, a két alak a magyarban megegyezik. A *flood* árvizet jelent, az *at*-nek valóban *-nál*, *-nél* az elsődleges jelentése, de itt nem erre van szükség. A visszafordítás során a program múlt idejű igének tekintette a *vett*-et, és past perfect igeidőnek fordította, a fordításból kimaradt a helyhatározó rag, így az angol nyelvtan szabályai szerint a *the flood*-ot itt tárgynak kell értenünk.

Az irodalmi fordítás visszaütal arra, hogy a cím egy Shakespeare-idézet (Brutus mondja ezt a Julius Caesarban), amit a könyvben meg is neveznek, és annak magyar fordításából veszi a cím fordítását, érdekes módon nem a *taken at the flood* szerkezetet fordítva, hanem az *in shallows*-t.

„There is a tide in the affairs of men.

Which, taken at the flood, leads on to fortune;

Omitted, all the voyage of their life

Is bound in shallows and in miseries.

On such a full sea are we now afloat,

And we must take the current when it serves,

Or lose our ventures.”

Magyarul:

„Az emberek dolgának árja van,

mely habdagállyal boldogságra visz.

De elmulasztva, teljes életök

nyomorban, s zátonyok közt zárva teng.

Ily duzzadt tenger víz most minket is.

Használni kell, míg áradatja tart,

vagy veszítjük a sors kedvezéseit.”

(Vörösmarty Mihály fordítása)

Első két mondat (azért veszem egybe, mert a magyar kiadásban egy mondatnak fordították):

Angol:

In every club there is a club bore. The Coronation Club was no exception; and the fact that an air raid was in progress made no difference to normal procedure.

Fordítás:

Minden ott levő klubban van egy klubfurat. A Coronation Club nem volt kivétel; és a tény, hogy egy légitámadás haladásban volt, nem csinált problémát a normál eljárásnak.

Vissza: There is a club bore in all clubs there. Coronation Club was not an exception; and the fact that he was an air raid in progress did not do a problem for the normal procedure.

Irodalmi fordítás:

A Koronázási Klub ugyanolyan klub, mint a többi: vannak benne unalmas, lerázhatatlan alakok és olyan bevett szokások, amelyeken az sem változtat, ha történetesen légiriadó van.

Az első mondatban a helyhatározó az *in every club*, ettől elkülönül a *there*, ami a *there is* szerkezet része. A program ezt nem érzékelte, és együtt fordította, így lett *minden ott levő klubban*. A *club bore* itt – mint azt később a regény kifejti – egy rendkívül unalmas klubtagot jelent, aki mindenáron untatni akar másokat. Ezzel szemben a *bore* szónak van *furat* jelentése is, és a magyar fordításban is ez jelenik meg.

A tulajdonneveket a program nem fordítja, még akkor sem, ha ez nem személynév, hanem egy klubnak a neve. Az *in progress*-nek a *haladásban* volt egy rendkívül szerencsétlen fordítása. A *nem csinált problémát a normál eljárásnak* szó szerinti fordítása egy kifejezésnek, a *difference* egyébként nem problémát jelent, hanem *különbség*-et.

A *klubfurat*-ot vissza angolra *club bore*-nak fordította a program. A plusz helyhatározó itt is megjelenik. A második mondatban a legfeltűnőbb hiba a *he* alany megjelenése, eszerint megjelent egy hímnemű személy, aki a légiriadóval azonos.

Major Porter, late Indian Army, rustled his newspaper and cleared his throat.

Fordítás:

Major Porter, a késői indiai hadsereg, suhogtatta az újságát és tisztította

a torkát.

Vissza:

Major Porter, the late Indian army, was swishing his newspaper and cleaned his throat.

Irodalmi:

Porter őrnagy, az Indiai Hadsereg nyugalmazott tisztje az újságjával zörgött, s a torkát köszörülte.

A program nem ismerte fel, hogy az őrnagy nem a tulajdonnév szerves része, és nem fordította le, így úgy tűnik, mintha a *Major* a keresznév lenne. A legfeltűnőbb pontatlanság itt, hogy a *late Indian Army*-t a program értelmező jelzőnek vette, és így a magyar fordításban Porter őrnagy és a késői indiai hadsereg mint azonos dolgok jelennek meg. A *rustle* jelent *suhogtatás*-t és *zörgetés*-t is, itt a zörgetés lett volna megfelelő. A *clear one's throat* kifejezés azt jelenti, hogy köszörüli a torkát, nem azt, hogy tisztítja. Ugyanezek a hibák a visszafordításban is megjelennek.

Every one avoided his eye, but it was no use.

Fordítás:

Minden egy elkerülte a szemét, de ez nem volt használat.

Vissza:

On all of them one kept clear of it son of a bitch, but this was not a usage.

Irodalmi:

Senki sem akart tudomást venni róla, de őt ez nem zavarta.

A program megint lefordította egymás után a szavakat. Felismerte, hogy a *his eye* tárgya az előtte álló igének, és ennek megfelelően tárgyragot kapott. A *use* főnévként valóban használatot jelent, de itt egy kifejezés része, így a jelentésében a használat sokkal összetettebben jelenik meg. A visszafordítás során a jelentés tovább távolodik az eredetitől. A *minden egy* úgy jelenik meg, mint egy az összes közül. A *szemét* szót nem tárgyas főnévnek ismerte fel, hanem alanyesetűnek, és ennek megfelelően fordította. A használat szót nem *use*-nak fordította vissza, hanem *usage*-nek.

I see they've got the announcement of Gordon Cloade's death in the Times

Fordítás:

Látom, hogy bevitték Gordon Cloade halálának a közleményét a Times-ba

Vissza:

I see that his communication was brought for Gordon Cloade death into Times

Irodalmi:

Itt olvasom a Times-ban Gordon Cloade halálhírét.

A magyar fordítás jelentése nem azonos az angol eredetivel, emellett mulatságosan hangzik. Az angolban a közlemény ott van a Times-ban, a magyarban pedig odakerül. A visszafordítás során megjelent egy *his*, ami nem utal senkire. A Gordon Cloade halála birtokos szerkezetből eltűnt a birtokos személyjel.

He said.

Fordítás:

mondta

Vissza:

he said it.

Irodalmi:

szólalt meg

Ez egy egyszerű mondat, a magyar fordítás pontos, de a visszafordítás már nem teljesen azonos az eredetivel, mivel a magyar *mondta* kifejezés magában foglalja a tárgyat, az angolban megjelent egy *it*.

Discreetly put, of course.

Fordítás:

Körületekintően tett, persze.

Vissza:

Cautiously act, sure.

Irodalmi:

Persze finoman fogalmaztak.

Ez egy szemléletes példája annak, hogy a két fordítás során hogyan változik az eredeti jelentés. A eredeti mondatból egy szó sem jelenik meg a visszafordítás során kapottban. Az irodalmi fordítás itt nagyon távol van a szószerintiségtől.

On Oct. 5th, result of enemy action.

Fordítás:

Októberen. 5-e, az ellenséges akció eredménye.

Visszafordítás:

On October. 5, the result of the hostile action.

Irodalmi:

Október 15-én, ellenséges tevékenység következtében

Az *Oct* után lévő pontot mondatzáró írásjelnek vette a program. A rövidítést helyesen ismerte fel, de az ötödikét már másik mondathoz tartozónak tekintette, így jött létre az *októberen* szerkezet, és az *ellenséges akció eredménye* mint értelmező jelző, aminek névelője is van.

No address given.

Fordítás:

Nincs cím adott.

Vissza:

There is not a title given.

Irodalmi:

Lakcím nincs.

A lakcím és a cím között különbség van.

4. Összegzés

A példák mutatják, hogy a fordítóprogram rengeteg olyan apróságot nem érzékel, ami kulturálisan meghatározott, így a beszélők számára nyilvánvaló. Helyesen nem fordította le a *Times* újság nevét, mert tulajdonnévnek érzékelte. Az *örnagy* esetében viszont kulturális megegyezés szerint le kellett volna fordítania a *major*-t. Mivel a mondatzáró és a rövidítést jelző pont ugyanaz a karakter, nem tud különbséget tenni a két funkció között, és ez gondokat okoz a nyelvtani szerkezetek felismerésében. Ha egy szónak több jelentése van, akár jelentésárnyalatokról van szó, akár poliszemiáról, a program kénytelen az egyik jelentés mellett dönteni, amihez nem veszi feltétlenül figyelembe a szöveggörnyezetet.

A program a vizsgálthoz hasonló szöveg fordítására önmagában alkalmatlan, de a fordító ember munkáját megkönnyítheti. Léteznek emellett könnyen elérhető, nem fordító-, hanem fordítástámogató programok is, ahol a program nem próbálja meg helyettesíteni az ember munkáját, és érdemi segítséget tud nyújtani.

Hivatkozások

- [1] Kenesei I., *A nyelv és a nyelvek*, Akadémiai Kiadó, Budapest, 2004.
- [2] Prószéky G., *Számítógépes nyelvészet*, Számítástechnika-Alkalmazási Vállalat, Budapest, 1989.
- [3] Prószéky G., *A nyelvtechnológia (és) alkalmazásai*, Aranykönyv Kiadó, Budapest, 2005.
- [4] Prószéky G., Kis B., *Számítógéppel - emberi nyelven. Intelligens szövegkezelés számítógéppel*, Szak Kiadó, Bicske, 1999.

Fotorealisztikus 3D grafika: számítógéppel generált tájak

Kovács Lehel István

Kiss Elemér Szakkollégium
Sapientia Erdélyi Magyar Tudományegyetem
Műszaki és Humántudományi Kar
`klehel@ms.sapientia.ro`

A Kiss Elemér Szakkollégium 2010. novemberében alakult a Sapientia Erdélyi Magyar Tudományegyetem marosvásárhelyi karán a MITIS Egyesület keretében.

Mindjárt megalakulása után együttműködési szerződést kötött a budapesti Eötvös Collegiummal.

Névadónk, Kiss Elemér 1929. augusztus 25-én született Brassóban. Gyermekkorát Csíkmenaságon töltötte. A középiskolát a csíkszeredai gimnáziumban végezte. Egyetemi diplomát a kolozsvári Bolyai Tudományegyetemen szerzett 1951-ben. 1961-ig a Bolyai Farkas Líceumban tanított, majd a Marosvásárhelyi Tanárképző Főiskola matematikai tanszékének adjunktusa volt. Az intézet megszűnése után a jelenlegi Petru Maior Tudományegyetem elődintézetében folytatta munkáját, ahol 1976-tól 1985-ig tanszékvezetői feladatokat is ellátott. Doktori disszertációját, melyben a modern algebra témaköréhez tartózkodó kérdésekkel foglalkozott, 1974-ben védte meg Gh. Pick professzor irányítása alatt.

Résztvett az Erdélyi Magyar Tudományegyetem alapításában, és haláláig professzora volt. Élete utolsó két évtizedében Bolyai Jánosnak a marosvásárhelyi Teleki-Bolyai könyvtárban található kéziratos hagyatékát tanulmányozta. Kutatói munkája eredményeként a Magyar Tudományos Akadémia 2001-ben külső tagjai közé választotta. 2006. augusztus 23-án halt meg Marosvásárhelyen.

A Kiss Elemér Szakkollégium 2013-ban a marosvásárhelyi Műszaki és Humántudományok Kar minden szakára kiterjedt, az addig működő

matematika és informatika szakkollégium pedig műhely lett.

Jelen dolgozat a Kiss Elemér Szakkollégium 2011-ben megtartott egyik előadása és azon kritériumokat és technikákat próbálja meg összefoglalni, amelyek segítségével valósághű grafikát lehet előállítani számítógépen. A modern számítógépes grafika célja a fotorealisztikus, valós ábrázolásmód, ami azt jelenti, hogy a számítógépes grafikával generált képeket gyakorlatilag nem lehet megkülönböztetni a fénykép vagy videófelvételektől.

1. Bevezető

A *generatív számítógépes grafika* a képi információ tartalmára vonatkozó adatok és algoritmusok alapján modelleket állít fel, képeket jelenít meg (*renderel*). Ide tartoznak a speciális effektusok előállítása vagy az animáció is, amely a generált grafikát az időtől teszi függővé. Általában két- (2D) vagy háromdimenziós (3D) grafikus objektumok számítógépes generálását, tárolását, felhasználását és megjelenítését fedi a fogalom.

Nyilvánvaló, hogy az ember által készített mesterséges objektumok könnyűszerrel modellezhetők fotorealisztikusan számítógépen, hisz nem egy már eleve számítógép segítségével volt megtervezve. A nagy kérdés a természet alkotta tájak, élőlények, kövek, sziklák stb. modellezése. Ebben nagy segítségünkre vannak a *fraktálok*.

A fraktálok *önhasznló*, végtelenül komplex matematikai alakzatok, amelyek változatos formáiban legalább egy felismerhető (tehát matematikai eszközökkel leírható) ismétlődés tapasztalható. Az elnevezést 1975-ben Benoît Mandelbrot adta, a latin *fractus* (vagyis törött; törés) szó alapján, ami az ilyen alakzatok tört számú dimenziójára utal. „A természet geometriájának fraktál arculata van” – vallotta Mandelbrot.

2. Általános követelmények

Fotorealisztikus képek előállításának általános követelményei ([1] alapján):

- *Térhatás (depth cueing)*: A 3D-s modelltér jelenete a 2D-s raszteres képen is térhatású legyen. Érvényesüljön a perspektivikus ábrázolási mód. Reálisan ábrázoljuk a tárgyak látható és nem látható

éleit, felületeit. Érvényesüljön a mélység-élesség. A messzeségbe tűnő objektumok legyenek elmosódottabbak, kevésbé kidolgozottak. Használjuk a *mip-mapping* technikát.

- *Felületek megvilágítása, tükröződés, árnyékok*: modellezzük és használjuk fel a természetben is lezajló jelenségeket. A képeken a fényhatások feleljenek meg a természet és a fizika törvényeinek. A természethűség érdekében használjuk természetes (természetutánzó) textúrákat. Érdes, göröngyös térhatású felületeket tudunk elkészíteni a *bump-mapping* technikával, amikor a felületre merőlegesen véletlenszerűen módosítjuk a tárgy felszínét: kiemelünk, le-süllyesztünk. A testek egymásra vetett árnyékait meg kell jelelni.
- *Átlátszóság, áttetszőség, köd, füst modellezése*: figyelembe kell venni a fénytörést, a fény intenzitásának csökkenését. Használjuk az *alpha-blending* technikát.
- *Textúrák alkalmazása*: a valóság-hűség érdekében fényképeket, ábrákat tudunk ráhúzni az egyes grafikus objektumokra.

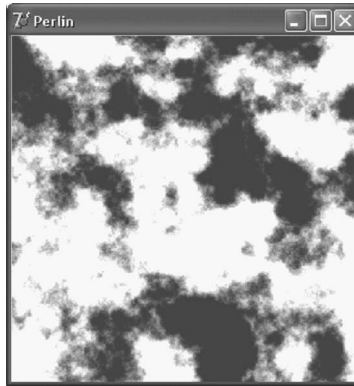
Mindezek az ábrázolási lehetőségeken, követelményeken túl, vizsgáljuk meg, milyen algoritmusok segítségével lehet előállítani a megfelelő természetes objektumokat, itt elsősorban felhőkre, domborzatra, vízre, fákra gondolunk. Megjegyezhető, hogy a nem természetes, mesterséges objektumok nagyon egyszerűen előállíthatók fotorealisztikusan, hisz az utóbbi években ezek megtervezése CAD eszközök segítségével történik (pl. épületek, bútorszat, lámpatestek, autók stb.), amelyek már eleve képesek arra, hogy fotorealisztikus látványtervet készítsenek a modellről.

3. Felhők generálása

Egy kép megalkotásakor elsődleges szempont a háttér létrehozása. A szabadban ez gyakran egy felhős égboltot (is) jelent.

A valóságmodellezéskor is nagy szerephez jutnak a véletlen fraktálok, hisz a természet alkotta valós objektumok nem teljesen szabályosak.

A véletlen fraktálok vagy véletlen halmazokból veszük fel értékeiket, vagy egy generált véletlen-számmal perturbáljuk a fraktál értékét, vagy



1. ábra. Felhőzet Perlin-zajjal

valamilyen más szinten kötődnek a véletlenhez, pl. a Brown-féle mozgás pályájának a fraktál jellegű tulajdonságait használjuk fel.

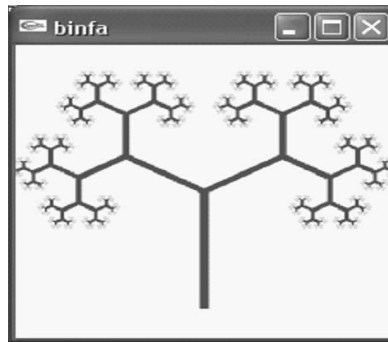
A valóság modellezésében felületeket, felhőzetet, atmoszférikus effektusokat stb. nagyon jól elő tudunk állítani *Perlin-zaj* [2] alkalmazásával.

Perlin zajfüggvénye R^n -en értelmezett ($f : R^n \rightarrow [-1, 1]$), az egész számokban csomópontokat képző rácshoz igazított pszeudo-véletlen spline függvény, amely a véletlenszerűség hatását kelti, ugyanakkor rendelkezik azzal a tulajdonsággal, hogy azonos bemeneti értékekre azonos függvényértéket térít vissza. Az n gyakrabban használt értékei: 1 – animáció esetén, 2 – egyszerű textúrák, 3 – bonyolultabb 3D textúrák, 4 – animált 3D textúrák (pl. mozgó felhők).

A következőképpen generálhatunk Perlin-zajt: adott egy bemeneti pont. Minden környező rács-csomópontra választunk egy pszeudo-véletlen értéket egy előre generált halmazból. Interpolálunk az így megkapott csomópontokhoz rendelt értékek között, valamilyen S görbét használva (pl. $3t^2 - 2t^3$).

Ha a Perlin-zajfüggvényt kifejezésben használjuk, különböző procedurális mintákat és textúrákat hozhatunk létre.

Ha ezeket a kifejezéseket fraktál-összegben használjuk, minden iterációban új adatot vihetünk be, amely valamilyen módon befolyásolja a teljes képet. Például domborzat generálás esetén, az iteráció során a fraktál dimenzióját akarjuk befolyásolni, azaz minden iterációban az amplitúdót osztani fogjuk egy bizonyos értékkel.



2. ábra. Bináris fa

A gyakorlati kísérletek azt mutatják, hogy a Perlin-zajfüggvény a következő együttthatóértékekre ad fotorealistikus felhős égboltot:

1. `r1 := 1000+Random(10000);`
2. `r2 := 100000+Random(1000000);`
3. `r3 := 1000000000+Random(2000000000);`

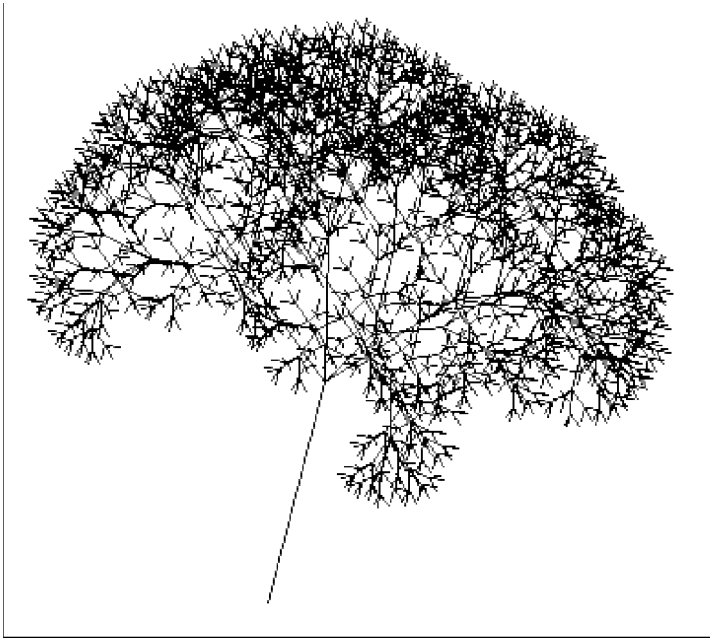
4. Fák, bokrok generálása

A távolban lévő fák, növényzet előállítható egyszerűen *bináris* vagy *kvadrális fák* segítségével, vagy *Barnsley-féle páfrányok* segítségével.

A barna törzsű fákat akár levél-szinten zöldre is színezhethetjük, vagy egy perturbáló faktor segítségével szétrázhatjuk az ágait, mintha szél fújta volna meg őket. A páfrányokat IFS segítségével állíthatjuk elő.

Az IFS az *Iterated Function System* (iterált függvényrendszer) kifejezés rövidítése. Egy IFS nem más, mint kontraktív, $R^2 \rightarrow R^2$ alakú transzformációk kollekcója, mely szintén egy leképezés. Az ilyen típusú leképezéseknek mindig van egy egyedi fixpontja, digitális képekre alkalmazva ez a fixpont általában egy fraktálkép.

A Barnsley-páfrányt [3] úgy állíthatjuk elő IFS-ként, hogy kiindulunk az origóból ($x_0 = 0, y_0 = 0$), kirajzoljuk a pontot, majd véletlenszerűen alkalmazunk egy transzformációt a következő négyből (pl. 300 000-szer), a kapott pontokat kirajzoljuk:



3. ábra. Véletlen perturbáció alkalmazása kvadrális fánál

1. $\begin{cases} x_{n+1} = 0 \\ y_{n+1} = 0,16 \cdot y_n, \end{cases}$ ezt a transzformációt 1%-os valószínűséggel alkalmazzuk.
2. $\begin{cases} x_{n+1} = 0,2 \cdot x_n - 0,26 \cdot y_n \\ y_{n+1} = 0,23 \cdot x_n + 0,22 \cdot y_n + 1,6, \end{cases}$ 7%-os valószínűséggel.
3. $\begin{cases} x_{n+1} = -0,15 \cdot x_n + 0,28 \cdot y_n \\ y_{n+1} = 0,26 \cdot x_n + 0,24 \cdot y_n + 0,44, \end{cases}$ 7%-os valószínűséggel.
4. $\begin{cases} x_{n+1} = 0,85 \cdot x_n + 0,04 \cdot y_n \\ y_{n+1} = -0,04 \cdot x_n + 0,85 \cdot y_n + 1,6, \end{cases}$ 85%-os valószínűséggel.

Ha a fák vagy bokrok az előtérben – tehát közel helyezkednek el, jóval bonyolultabb algoritmusokkal tudjuk ezeket fotorealisztikussá tenni.

Ezek az algoritmusok a fa természetes növekedését követik, véletlen perturbálófaktorok alkalmazásával, a törzs textúrázásával, az ágak levelekkel való ellátásával együtt. Minden egyes levél hű mintázata



4. ábra. Barnsley-páfrány

a természetes leveleknek.

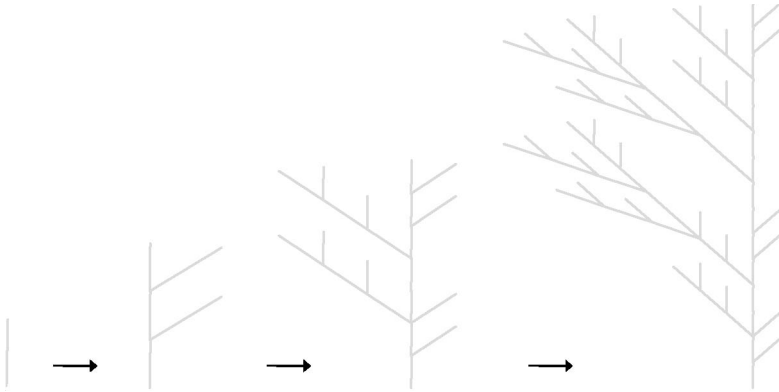
Az egyik módszer a *graftálok* alkalmazása. A graftálok egyszerű szabályokból iteratív eljárással létrehozott alakzatok, amik a növényeket modelleznek.

Példa graftálra:

1. Legyen egy négy jeltől álló nyelv: 0, 1, [,].
2. A [-t mindig követi egy], a] előtt mindig áll egy [.
3. A [] páros között egy vagy több jel is állhat.
4. A 0 és 1 jelentése: lépj előre egy egységnyit.
5. A [jelentése: jegyezd meg az aktuális pozíciót és irányt, majd fordulj el meghatározott szöggel.
6. A] jelentése: menj vissza és fordulj a legutóbb megjegyzett pozícióba és irányba.

„Életet” egy graftálba kicserélési szabályok alkalmazásával lehelhetünk. Például:

1. Cseréljünk ki minden 0-át 1[0]1[0]0-ra.
2. Cseréljünk ki minden 1-et 11-re.



5. ábra. Graftál „növekedése”¹

Fotorealisztikus fa előállítási algoritmusokat ír le Gilles Tran [5]. Ezeket próbáltuk meg továbbfejleszteni és úgy paraméterezni, textúrázni, hogy általános fákat lehessen velük előállítani.

5. Vízfelület, hegyes táj, domborzat generálása

A domborzat modellezése a virtuális valóság és a fotorealisztikus grafika egyik fontos alkotóeleme.

Az egyik legsikeresebb domborzat-modell a fraktál domborzat-modell, amelynek az alapja szintén a Perlin-zaj [6].

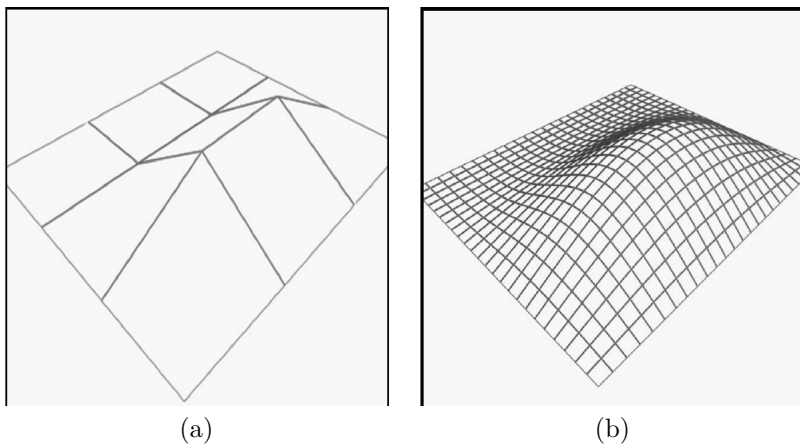
A fraktál domborzat-modell létrehozásához négy elem szükséges:

- egy alapfüggvény, amely megadja a domborzat alakját (Perlin-alap),
- a fraktál dimenziója (az amplitúdó módosulása minden iterációban),

¹Madár János és Abonyi János nyomán, Veszprémi Egyetem

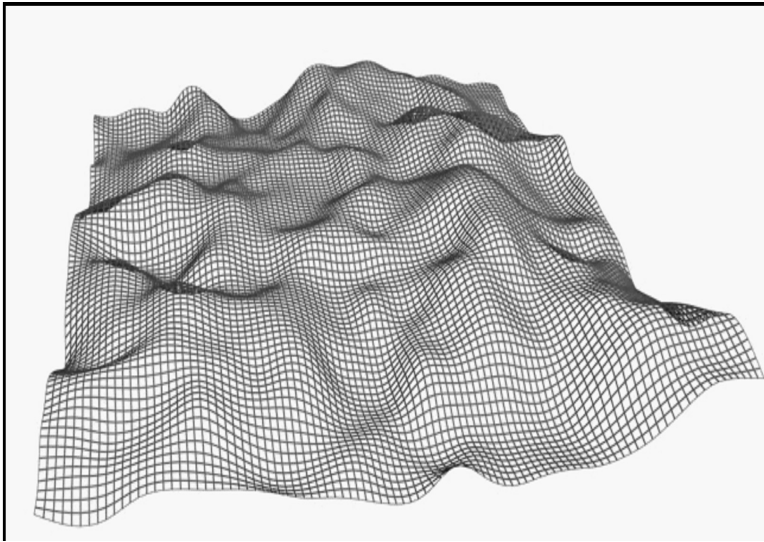


6. ábra. Fotorealisztikus fák²



7. ábra. (a) *Szimulálás*: maximális közelítés véges adathalmazból szimulált domborzaton (GPS). (b) *Szintetizálás* esetén, nincs „maximális” közelítés, ugyanis a domborzatot leíró eljárások mindig generálnak új adatot számunkra

²POV-Ray maketree makró alapján [5]



8. ábra. Alap domborzatmodell

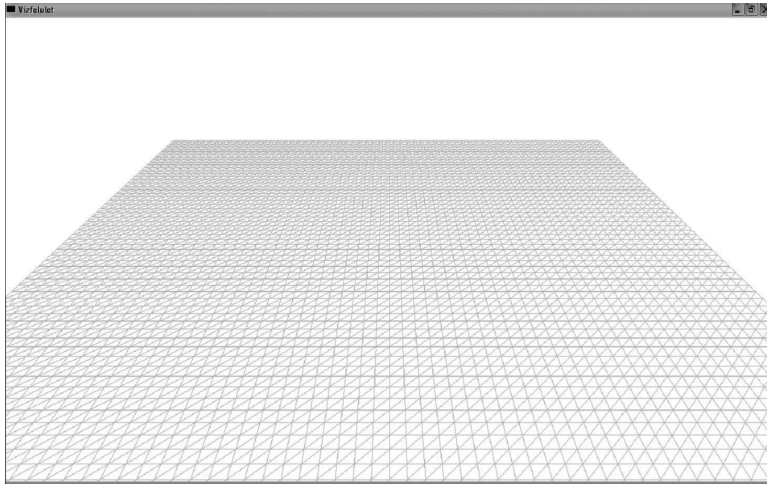
- az oktávok (iterációk) száma,
- a frekvencia módosulási tényezője.

Az algoritmusban a Perlin-zaj első iterációja dönti, hogy az adott pont magasság szerint milyen tájegységhez tartozik, majd az iterációs lépésekben, a tájegységnek megfelelő amplitúdó és frekvencia változás paramétereit alkalmazzuk. Például hegyek esetén az amplitúdó kis változást kell hogy eredményezzen a fraktálösszegben, míg egy fennsík esetén az amplitúdónak egyből redukálnia kell a részletét, hogy ezt egy sima felszínre alakítsa.

Domborzatot kétféleképpen állíthatunk elő: *szimulálás* és *szintetizálás* segítségével.

A szimulálás azt jelenti, hogy létező adatok alapján készül a modell (véges adatmennyiség); a szintetizálás pedig azt, hogy a természetben előforduló szabályosságok alapján állítunk elő virtuális modelleket.

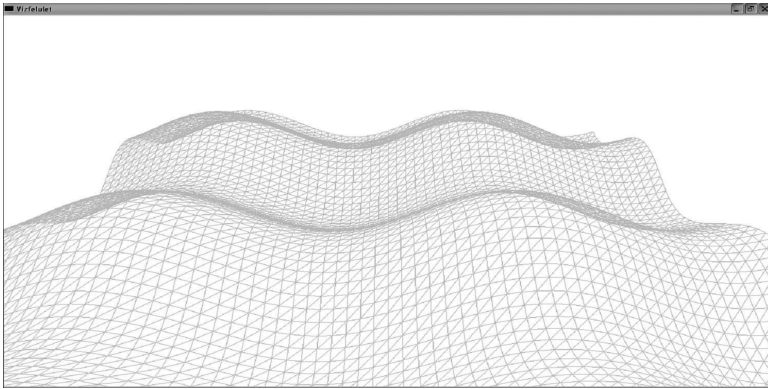
Algoritmus felületgenerálásra:



9. ábra. Alap vízmodell

1. Adott egy bemeneti pont.
2. Minden környező rács-csomópontra választani kell egy pszeudo-random értéket egy előre generált halmazból (mivel a csomópontok koordinátái egész számok, ezeket használjuk az eredmény kiválasztására).
3. Majd interpolálni kell az így megkapott csomópontokhoz rendelt értékek között, valamilyen S görbét használva. (pl. $3t^2 - 2t^3$).
4. Ha ezeket a kifejezéseket fraktál összegben használjuk minden iterációban új adatot vihetünk be a képbe, amik valamilyen módon befolyásolják ezt.
5. Domborzat generálás esetén, az iteráció során a fraktál dimenzióját akarjuk befolyásolni, azaz minden iterációban az amplitúdót osztani fogjuk egy bizonyos értékkel.

Vízfelszín modellezésére is kiválóan alkalmas a Perlin-zaj, itt azonban szem előtt kell tartanunk a különböző fizikai törvényeket is, például a hullámszám megvalósítására. Vízfelszín létrehozására elkerülhetetlen az animáció használata, éppen ezért a gyorsaság és hatékonyság növelése érdekében jobb ezeket az algoritmusokat valamilyen hardver által támogatott árnyaló nyelvben megírni.

10. ábra. Animált vízfelület³

Az animáláshoz használt CG program:

```
1.  struct appdata
2.  {
3.      float4 position: POSITION;
4.      float4 color: COLOR0;
5.      float3 wave: COLOR1;
6.  };
7.
8.  struct vfconn
9.  {
10.     float4 HPos: POSITION;
11.     float4 Col0: COLOR0;
12.  };
13.
14.  vfconn main(appdata IN, uniform float4x4 ModelViewProj)
15.  {
16.     vfconn OUT;
17.     // szinusz hullámok
18.     IN.position.y = (sin(IN.wave.x + (IN.position.x / 5.0) )
19.                     + sin(IN.wave.x + IN.position.z / 4.0) ) * 2.5f;
20.     OUT.HPos = mul(ModelViewProj, IN.position);
21.     OUT.Col0.xyz = IN.color.xyz;
22.     return OUT;
23.  }
```

³Owen Bourne, NeHeGL 47. lecke alapján

Foster és Fedkiw [7] olyan szimulációs módszert dolgozott ki, amelyben egy folyadék térfogatát egy implicit ϕ függvény körvonala határozza meg. A víz felülete: $\phi = 0$, a $\phi \leq 0$ a vizet, a $\phi > 0$ a levegőt jelenti. Az implicit függvény ábrázolása egy ideiglenesen koherens, finom, egyenletes vízfelszín eredményez. Ez az implicit felület időben és térben dinamikusan alakul, a folyadék u sebességének függvényében. Osher és Sethian szerint [8] az egyenlet: $\phi_t + u \cdot \nabla \phi = 0$, ahol ϕ_t a ϕ függvény idő szerinti deriváltja, és ∇ a gradiens operátor: $\nabla = (\partial/\partial x, \partial/\partial y, \partial/\partial z)$.



11. ábra. Fotorealisztikus táj - fraktálok segítségével⁴

Hivatkozások

- [1] Budai A., *A számítógépes grafika*, LSI Oktatóközpont, Budapest, 1999.
- [2] K. Perlin, An image synthesizer, *Computer Graphics*, 19(3), 1985.
- [3] M. Barnsley, *Fractals everywhere*, Academic Press, 1988.

⁴Tilki Csaba nyomán, Debreceni Egyetem

-
- [4] Szirmai-Kalos L., Antal Gy., Csonka F., *Háromdimenziós grafika, animáció és játékfejlesztés*, Computerbooks, Budapest, 2006.
 - [5] G. Tran, : *3D art and graphic experiments*,
<http://www.oyonale.com>
 - [6] D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, S. Worley, *Texturing & Modeling, A Procedural Approach*, AP Professional, 1994.
 - [7] N. Foster, R. Fedkiw, Practical animation of liquids, Proceedings of SIGGRAPH 2001, *Computer Graphics Proceedings, Annual Conference Series, ACM*, pp. 23–30.
 - [8] S. Osher, J. Sethian, Fronts propagating with curvature dependent speed: Algorithms based on hamilton-jacobi formulations. *J. Comp. Phys.* 79 (1988), pp. 12–49.

Webes alkalmazások tesztelésének automatizálása

Kovács Máté*

Eötvös József Collegium**

kovmat86@gmail.com

1. Bevezetés

A cikkben szeretném megosztani a munkahelyi és hobbi fejlesztés során szerzett webes tesztautomatizálással kapcsolatos tapasztalataimat. A munkaerőpiacon növekvő igény van tesztelő illetve tesztautomatizáló szakemberekre, ezért remélem, hogy a cikk egyúttal kedvcsináló is lehet azok számára, akik szimpatizálnak a tesztelői pályával.

Először egy példaalkalmazást és az azzal kapcsolatos követelményeket ismertetem, hogy a bemutatott eszközökhöz példát tudjak nyújtani. A cikkben főként a Selenium tesztautomatizáló eszköz működését fogom tárgyalni.

Egy webes alkalmazás elkülöníthető szerver és kliens oldali részre. Ez a cikk a kliens oldali résszel foglalkozik, ahol a felhasználó egy böngésző segítségével tudja az alkalmazás által szolgáltatott tartalmat megjeleníteni és kezelni.

*EPAM Systems Kft., Budapest

**2005–2010

2. Az automatizálás előnyei a tesztelés fajtái alapján

A különböző tesztelési fajtákon keresztül tekintsük át, hogy miben lehet segítségünkre az automatizáció!

2.1. Funkcionális tesztelés

Funkcionális tesztelés során vizsgáljuk, hogy a rendszer eleget tesz-e a felhasználó által támasztott funkcionális követelményeknek. Például a megadott mezők kitöltése és az „OK” gomb megnyomása után megjelenik „A tranzakciót végrehajtottuk” felirat. A tesztautomatizálás segít abban, hogy funkcionális teszteket kevés energia ráfordításával többször végre tudjuk hajtani.

2.2. Nem funkcionális tesztelés

Hordozhatósági tesztelés: A webes alkalmazás működik különböző operációs rendszereken különböző böngészőkkel megjelenítve.

Teljesítmény tesztelés: Az alkalmazás stabilitását vizsgáljuk különféle jellegű igénybevétel mellett. Például hogyan viselkedik az alkalmazás, ha azt egyidejűleg többen használják. A tesztautomatizáló rendszer segítségével tudjuk ezt a párhuzamos felhasználást szimulálni, ekkor a több böngészőpéldányban indulnak el a tesztek. Tapasztalataim szerint – alkalmazástól függően – egy átlagos számítógépen akár 5 vagy 10 böngészőablakban is futtathatunk tesztek. Figyelembe véve, hogy az automatizált futtatást delegálhatjuk például egy szerver farm vagy egy virtualizációs szerver virtuális gépeire, lehetőségünk nyílik, akár több száz felhasználó egyidejű szimulálására is. Egy másik fajta tesztelési módszer, amikor hosszú időn – több napon vagy héten – keresztül történő felhasználást szimulálunk, mellyel vizsgáljuk a memória illetve tárhely elszívargás esélyét.

2.3. Mérések

Az automatizálás segít különböző mérések végrehajtásában, például az egyes funkciók válaszidejének meghatározásában.

3. Webes alkalmazás példa áttekintése

Tekintsünk át egy esettanulmányt a kimondottan ehhez a cikkhez kitalált Online Piacter webes alkalmazásról! Az Online Piacter weboldalán a felhasználók hirdetményeket tehetnek fel az általuk eladásra kívánt termékről. Más felhasználók kereshetnek ezek között a termékek között különféle keresési feltételek alapján.

Online Piacter





Keresés

Termékkategóriák

Keresési feltételek

Ár(-tól,-ig):

Szobák száma(-tól,-ig):

Település
Állapot

Alapterület
Tájolás

Építési mód
Fűtés

Pince ☐
Erkély ☐
Parkoló ☐
Garázs ☐

1. ábra. Az Online Piacter webes alkalmazás felülete

A webes alkalmazás HTML forrásának csak azt a részét adjuk meg, amelyre a tesztautomatizálás bemutatása során hivatkozni fogunk:

```

...
<h3>Kereses</h3>
<div>
  Termékkategóriák
  <select id="category">
    <option>Ingatlan
    <option>Gepkocsi
    <option>Elektronikai eszkozok
  </select>
  ...

```

```

<p>Keresési feltételek</p>
<table>
  <tr><td>Ár(-tol,-ig): <td><input class="short-number"/>-<input
    class="short-number"/>
  <tr><td>Szobák száma(-tol,-ig): <td><input
    class="short-number"/>-<input class="short-number"/>
  <tr><td>Település <td><input class="normal"/>
  <td>állapot <td><input class="normal"/>
  <tr><td>Alapterület <td><input class="normal"/>
  <td>Tájolás <td><input class="normal"/>
  <tr><td>Építési mód <td><input class="normal"/>
  <td>Fűtés <td><input class="normal"/>
  <tr><td>Pince<input type="checkbox"/><td>Erkély<input
    type="checkbox"/>
  <td>Parkoló<input type="checkbox"/><td>Garázs<input type="checkbox"/>
</table>
<input type="submit" name="start-search" class="submit-button"
  value="Keresés indítása"/>
<input type="submit" name="reset" class="normal-button"
  value="Alaphelyzet"/>
...

```

A továbbiakban csupán a keresési funkciót vizsgáljuk, mellyel szemben támasztott követelmények az alábbiak.

3.1. Funkcionális követelmények

- A felhasználó kiválaszthatja, hogy milyen termék kategóriában szeretne keresni (Ingatlan, Gépkocsi, Elektronikai eszköz).
- A termék kategória kiválasztása után megjelennek a kategóriához tartozó keresési feltételek (Ingatlan esetén Ár, Szobák száma, Település, Alapterület, stb.; Gépkocsi esetén Ár, Település, Évjárat, Hengerűrtartalom, stb.).
- Az Ár keresési feltétel egy értéktartomány megadásával történik.
- A felhasználó a keresést a „Keresés indítása” gombbal indíthatja.
- Érvénytelen keresési feltételek megadása esetén hibaüzenetet ír ki a rendszer. Bizonyos mezőkbe legfeljebb 10 karaktert szabad gépelni, ellenkező esetben a rendszer gépeléskor hibaüzenetet jelenít meg.
- Legalább egy keresési feltételt ki kell tölteni, ellenkező esetben hibaüzenetet ír ki a rendszer.

- Helyesen megadott keresési feltételekkel történő keresés indítása után az oldalon a „Feldolgozás folyamatban ...” felirat jelenik meg, majd némi várakozás után megjelenik a keresési eredmények listája.

3.2. Nem funkcionális követelmények

- A rendszernek képesnek kell lennie legalább 500 felhasználó egyidejű kiszolgálására.
- A rendszernek képesnek kell lennie legfeljebb 10 másodpercen belül keresési eredményt visszaadnia.
- A rendszernek képesnek kell lennie folyamatosan működni leállítással nélkül legalább 2 hétig.
- A rendszernek támogatnia kell az Internet Explorer, Mozilla Firefox és Google Chrome böngészőket (a támogatott böngészők verziószámát is fel szokták tüntetni, de a példában ettől eltekinthetünk).

4. Tesztautomatizáló eszközök

Számos tesztautomatizáló eszközt találhatunk a világhálón, fizetőseket és ingyeneseket egyaránt. Példaként a Selenium Webdriver [2] nyílt forrású ingyenesen használható webes tesztautomatizáló eszközt mutatom be.

A Selenium Webdriver telepítéséhez és használatához útmutatót a <http://docs.seleniumhq.org/> oldalon találunk. A tesztautomatizáció lépéseit Java programozási nyelven rögzítjük és Java alkalmazásként futtatjuk.

Tekintsük át a korábban felvázolt Online Piac tér alkalmazás tesztautomatizálásának lépéseit!

4.1. Navigálás a kívánt weboldalra

A tesztelést Firefox böngészővel végezzük. Első lépésként az alkalmazás tesztkörnyezetének címére navigálunk az alábbi Java kódrészlet alapján:

```
WebDriver driver = new FirefoxDriver();  
driver.get("http://www.piacter.internal.com/test1");
```

4.2. Elemek kiválasztása és szerkesztése weboldalon

Szeretnénk szimulálni, hogy a felhasználó rákattint egy beviteli mezőre és megadja a kívánt értéket. Első lépésben ki kell választanunk a megfelelő mezőt, melyhez a WebDriver osztály `findElement()` metódusát használjuk. A metódus paraméterként egy lokátor típusú objektumot vár, melyet a By osztály segítségével hozhatunk létre.

Az elemek (például lista vagy szövegdoboz) kiválasztására több lehetőségünk is van. A lényeg, hogy valamilyen szempont szerint fogunk az oldal HTML DOM fáájában keresni. Néhány lehetőség:

- egyedi azonosító (id), pl.: `By.id("category")`

```
<select id="category">  
    <option>Ingatlan  
    <option>Gepkocsi  
    <option>Elektronikai eszkozok  
</select>
```

- név (name), pl.: `By.name("start-search")`

```
<input type="submit" name="start-search" value="Kereses"/>
```

- osztály (class), pl.: `By.class("normal-button")`

```
<input type="submit" name="reset" class="normal-button"  
value="Alaphelyzet"/>
```

- XPath vagyis a HTML DOM-fában definiált útvonal segítségével, pl.:

```
By.xpath("/html/body/div[3]/div/div/div/  
table/tbody/tr[1]/td[2]/input[1]")
```

```
<input type="submit" name="start-search" class="submit-button"  
value="Kereses"/>
```

Az egyedi azonosító alapján történő kiválasztás az ajánlott, amennyiben az adott HTML elem rendelkezik egyedi azonosítóval (id attribútummal). Előfordul azonban, hogy a HTML elem nem rendelkezik egyedi azonosítóval, például, mert az alkalmazás fejlesztői nem gondoltak rá, hogy szükség lehet rá vagy pedig valamilyen keretrendszer által generált tartalomról van szó, amiben nincsenek vagy pedig véletlenszerűen generált azonosítók szerepelnek. Az id-vel történő kiválasztás gyorsabb, mint az XPath-szal történő.

Válasszuk ki, hogy milyen kategóriában szeretnénk keresni, legyen az például az Ingatlan! Itt először megkeressük az oldal HTML kódjából a termékkategóriák listát id segítségével. Ez egy **Select** típusú objektum lesz, melyet **categoryList**-nek nevezünk el, és kiválasztjuk belőle az Ingatlan elemet.

```
Select categoryList = new
    Select(driver.findElement(By.Id("category")));
categoryList.selectByVisibleText("Ingatlan");
```

Ezután az oldalon megjelennek az Ingatlanhoz tartozó keresési feltételek megadására szolgáló mezők (lásd ábra), például: Ár, Szobák száma, Aapterület, stb. Töltsünk ki az Ár mezőt! A HTML forrásban látható, hogy az Árhoz tartozó beviteli mezőhöz nem tartozik sem egyedi azonosító, sem név. Ebben az esetben XPath segítségével hivatkozhatunk az adott mezőre, mely így néz ki:

```
/html/body/div[3]/div/div/div/
    table/tbody/tr[1]/td[2]/input[1].
```

A módszer hátránya, hogy amennyiben megváltozik az oldal szerkezete (például az Ár mező elé bekerül egy másik beviteli mező), akkor az XPath kifejezés érvénytelenné válik vagy egy másik, oldalon szereplő mezőre fog mutatni. A korábban használt **findElement()** metódus és a meghatározott XPath segítségével lekérjük a mezőt egy **WebElement** típusú változóba, majd beállítjuk az értékét 5 millióra. Az érték beállítása a **sendKeys()** metódussal történik.

```
WebElement element = driver.findElement(By.xpath(
    "/html/body/div[3]/div/div/div/table/tbody/tr[1]/td[2]/
        input[1]"));
element.sendKeys("5000000");
```

Végül kattintsunk a „Keresés indítása” gombra, mely a **click()**

metódussal történik.

```
driver.findElement(By.name("start-search")).click();
```

4.3. Várakozás elemek betöltésére

Miután a keresést elindítottuk, szeretnénk ellenőrizni, hogy megjelennek a keresési eredmények az oldalon. A keresési eredmények megjelenésére – a követelményeket figyelembe véve – várnunk kell legfeljebb 10 másodpercet. Ezt a tényt közölnünk kell a tesztautomatizáló rendszerrel. Tekintsük át milyen lehetőségeink vannak:

1. Közvetlenül a `click()` metódus meghívása után vizsgáljuk a keresési eredményének megjelenését. Ezzel az a probléma, hogy a Selenium nem várja meg, amíg az oldalon az eredmények megjelennek (ha ezt például egy AJAX hívás frissíti be). Ehelyett az oldal aktuális állapotának megfelelően nem találja meg a keresett elemet, és `NoSuchElementException` kivétel dobásával leállítja a tesztet. Ezt az esetet csupán, azért vázoltam fel, mert a tesztautomatizálás során gyakran váratlanul szembesülünk a ténnyel, hogy a tesztautomatizáló rendszer előbb próbált egy értéket megkeresni az oldalon, minthogy az megjelent volna.
2. Várunk pontosan 10 másodpercet, erre szolgál a `Java Thread.Sleep()` metódusa, ahol ezredmásodpercben adhatjuk meg a várakozás idejét. Ezzel a módszerrel a probléma, hogy mindig tíz másodpercet fogunk várni függetlenül attól, hogy a keresési eredmények 1 másodperc alatt megjelentek. Ha ismételtlen kell kereséseket végrehajtanunk, illetve többször lefuttatni a teszteket, akkor a felesleges várakozásokból tetemes mennyiség adódhat össze. Mindezekről függetlenül többször találkoztam olyan többször is elbukó automatizált tesztesettel, melyet csak ilyen „beégetett” várakozással lehetett megjavítani.
3. Szeretnénk csak addig várni, amíg a keresett elem meg nem jelenik, és utána a teszt futtatását folytatni. Erre nyújt megoldást a Selenium által támogatott Explicit várakozás [3, 4]. Ez azt jelenti, hogy bizonyos időközönként ellenőrizzük, hogy a keresett elem megjelent-e a képernyőn. Tegyük fel, hogy a kere-

sési eredményeket egy `result` id-jű lista tartalmazza. Létrehozunk egy `WebDriverWait` típusú objektumot, aminek paraméterül adjuk a `WebDriver` objektumot, a maximális várakozási időtartamot másodpercben illetve ezredmásodpercben, hogy milyen gyakran kívánjuk ellenőrizni, hogy az elem mikor jelenik meg. Az `until()` metódus meghívásával jelezzük, hogy várakozunk, amíg a paraméterként átadott feltétel nem teljesül. Ezt a feltételt az `ExpectedConditions.presenceOfElementLocated` hívással adjuk, melynek paramétere a keresett elemre mutató lokátor `By.id` formában megadva.

```
WebDriverWait wait = new WebDriverWait(driver, 10, 500));
WebElement result =
    wait.until(ExpectedConditions.presenceOfElementLocated(
        By.id("result")));
```

4. A Selenium egy másik lehetősége az Implicit várakozás, mely annyiban különbözik az Explicit várakozástól, hogy ezt nem csak egy adott elem keresésekor adjuk meg, hanem a `WebDriver` egész élettartamára vonatkozóan (illetve amíg újból felül nem bíráljuk az Implicit várakozást). Első lépésben beállítjuk a `WebDriver`-nek, hogy mennyi ideig várakozzon, a példában 10 másodpercig. Ezután minden egyes `findElement` hívásnál a tesztünk várakozni fog az elem megjelenésére.

```
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
//...
WebElement result = driver.findElement(By.id("result"));
```

Ismétlésképpen tekintsük át mi a különbség az Explicit és az Implicit várakozás között! Explicit várakozás esetén egy adott elem keresésekor mondjuk meg, hogy legfeljebb mennyit kell várni (ezt explicit módon jelezzük a kódban a keresésekor). Implicit várakozás esetén csak a `WebDriver` létrehozásakor jelezzük, hogy várakoznunk kell, amikor egy elemet keresünk. Ez minden egyes `findElement()` híváskor meg fog történni (implicit módon, tehát minden egyes elem keresésekor a várakozás tényét nem kell jeleznünk a kódban).

4.4. Több elem együttes lekérése

Eddig olyan eseteket néztünk, ahol egy adott elem meglétére vagy használatára voltunk kíváncsiak. A Selenium támogatja, hogy egy lekérdezéssel az oldalon található elemek listáját tudjunk lekérni. Ehhez használhatjuk a `findElements()` (s-sel a végén) metódust. A kapott elemek mindegyikére megnézzük, hogyha 10-nél több számjegyetünk be, akkor az „Érvénytelen paraméter!” felirat megjelenik az error id-jű mezőben. Az oldalon 4 olyan mező van (az Ár és a Szobák számának alsó és felső határa), aminek a class attribútuma „short-number”. A `findElement()`-t 4-szer kellett volna meghívunk (például különböző XPath kifejezésekkel), ehelyett egy `findElements()` hívás is elég. A visszakapott elemek listáján végigiterálva megvizsgáljuk, hogy a hibaüzenet megjelenik-e.

```
List<WebElement> numberFields =
    driver.findElements(By.class("shortNumber"));
for (WebElement numberField : numberFields) {
    numberField.sendKeys("12345678901");
    WebElement errorText = findElement(By.id("error"));
    if (!errorText.equals("Ervenytelen parameter!")) {
        //throw exception
    }
    numberField.clear();
}
```

A `findElement()` és a `findElements()` közötti különbség, hogy míg az első kivételt dob, amennyiben a keresett elem nem található addig a második egy üres listát ad vissza és a teszt futása tovább folytatódik.

4.5. Szöveg jelenlétének vagy hiányának vizsgálata az egész oldalon

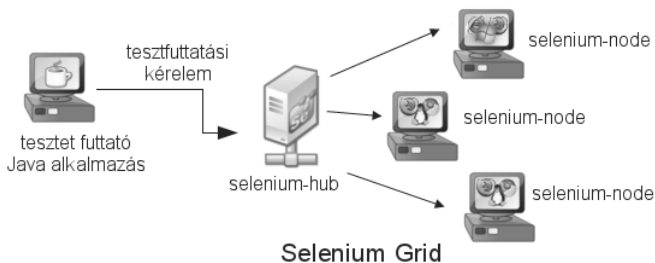
Találkoztam olyan helyzettel, hogy arra voltunk kíváncsiak, hogy egy adott szövegrész – tetszőleges helyen – szerepel-e az oldalon. Teljesen véletlenszerű, hogy a szöveg hol és milyen attribútumokkal jelenik meg, ezért lokátort nem tudunk használni arra, hogy megtaláljuk. Ebben az esetben megoldást nyújt a `getPageSource()` metódus, ami a teljes oldal szöveges reprezentációját adja vissza. A szövegen aztán tudunk különböző műveleteket végezni, hogy például egy szövegrészt tartalmaz-e (`contains()`).

```
if (driver.getPageSource().contains("Hiba!"))  
{  
    //...  
}
```

Az említett módszer nem túl robusztus, hiszen ha az oldalon egy másik helyen „véletlenül” megtalálható a keresett szövegrész, akkor a tesztünk hibás eredményt adhat vissza. Törekedjünk lokátorok használatára, ahol csak lehetséges.

4.6. Nem funkcionális követelmények tesztelésének támogatása automatizálással

Hordozhatósági követelményként szerepelt a különféle böngészők támogatása. A Firefox-hoz tartozó drivert a Selenium könyvtárba beépítve megkapjuk. Internet Explorer és Chrome esetén le kell töltenünk és futtatnunk kell az adott böngészőhöz tartozó drivert, mely a Selenium weboldaláról beszerezhető [5, 6]. Elvileg a Firefox-hoz létrehozott tesztjeink gond nélkül futtathatjuk más böngészőkkel is. A Selenium és más, hasonlóan HTML DOM-fa alapján működő tesztautomatizáló eszközök hátránya, hogy az oldal kinézetét nem tudják ellenőrizni, például, hogy a weboldal egyes blokkjai nem takarják ki egymást. Célszerű manuális teszteléssel ellenőrizni, hogy a weboldal a kívánt módon jelenik meg különböző böngészőkben.



2. ábra. A Selenium Grid felépítése

Részen terhelés teszteléshez nyújt támogatást a Selenium Grid

funkció, amivel több gépen tudunk párhuzamosan teszteket futtatni. Mindez segít annak tesztelésében, hogy a rendszer helyesen működik, ha azt egyidejűleg többen használják.

A Selenium Grid [8] használatához le kell töltenünk és el kell indítanunk a hub-ot. A `selenium-server-standalone-*.jar` letölthető a Selenium oldaláról: <http://code.google.com/p/selenium/downloads/list>. Futtatni az alábbi módon tudjuk parancssorból: `java -jar selenium-server-standalone-2.14.0.jar -role hub`.

A hub egy központi egység, ami a tesztfuttatási kérélmeket kezeli, és amihez a teszteket futtató gépeket – node-okat – csatlakoztatni tudjuk.

A node-ok indítása az egyes gépeken parancssorból az alábbi módon történik (a -hub kapcsolóval az imént elindított hub elérési útvonalát kell megadni): `java -jar selenium-server-standalone-2.14.0.jar -role node -hub http://localhost:4444/grid/register`.

A tesztfuttatási kérélmeket Java kódból tudjuk elküldeni. Ehhez meg kell adnunk egy `DesiredCapabilities` objektumot, amivel többek között beállítható, hogy a tesztek melyik böngészővel fussanak. Ezután létrehozunk egy `RemoteWebDriver` típusú objektumot, amit ugyanúgy tudunk használni, mint az előző példákban, az egyetlen különbség, hogy a driverrel végrehajtott böngészőműveletek nem a saját gépünkön, hanem a hub által kiválasztott node-on fognak végrehajtódni.

```
DesiredCapabilities capability = DesiredCapabilities.firefox();
WebDriver driver = new RemoteWebDriver(new
    URL("http://localhost:4444/wd/hub"), capability);
```

A terhelési teszteléssel kapcsolatban érdemes megemlíteni, hogy erre a célra léteznek a Selenium-on kívül hatékonyabb és pontosabb eszközök. Az egyik ilyen terhelési tesztelést támogató eszköz az Apache JMeter [7].

5. Összefoglalás

A fentiekben ismertettem a Seleniumnal szerzett tapasztalataimat. Az itt felsorolt problémák és a rájuk adott megoldások megjelennek más tesztautomatizáló eszközöknél is. Például más eszközöknél is megoldást kell találnunk a weboldal elemeinek lokalizálására illetve a keresett elem késleltetett megjelenésének kivárására. Remélem, hogy cikkemmel megfelelő iránymutatást tudtam adni azoknak, akik a későbbiekben webes tesztautomatizálással szeretnének foglalkozni.

Hivatkozások

- [1] D. Graham, E. van Veenendaal, I. Evans, R. Black, *Foundations of Software Testing: ISTQB Certification*, Cengage Learning, 2012.
- [2] Selenium Webdriver használata:
http://docs.seleniumhq.org/docs/03_webdriver.jsp
- [3] Selenium Webdriver használata haladóknak:
http://docs.seleniumhq.org/docs/04_webdriver_advanced.jsp
- [4] Implicit és Explicit várakozás:
<http://santoshsarmajv.blogspot.ch/2012/10/implicit-wait-vs-explicit-wait.html>
- [5] Webdriver Internet Explorer-hez:
<https://code.google.com/p/selenium/wiki/InternetExplorerDriver>
- [6] Webdriver Chrome-hoz:
<https://code.google.com/p/selenium/wiki/ChromeDriver>
- [7] A JMeter weboldala: <http://jmeter.apache.org/>
- [8] Selenium Grid használata:
<https://code.google.com/p/selenium/wiki/Grid2>

A Collatz-sejtés matematikai és informatikai megközelítéseinek elemzése

Kovács Péter*

Eötvös József Collegium**

pjotr129@inf.elte.hu

Témavezető: Burcsi Péter, ELTE IK

1. Történeti áttekintés

Lothar Collatz német matematikus 1937-ben fogalmazta meg híres sejtését, melyre a mai napig nem ismert bizonyítás. A probléma eleinte nem keltette fel a matematikusok érdeklődését, csak az 1950-es években hívta fel rá a figyelmet Stanislaw Ulam professzor a Syracuse egyetemen. A 70-es években már többen foglalkoztak a sejtéssel. Erdős Pál azt állította, a matematika még nem áll készen ilyen problémákra, és 500 dollárt kínált a megoldásért. J. H. Conway [1], Jeffrey C. Lagarias [3] és Helmut Hasse is jelentetett meg cikkeket a témában. Természetesen az informatika rohamos fejlődésével a számítógépeket is bevonták a kutatásba, de csak részeredményeket értek el. 1996-ban Sir Bryan Thwaites 1000 dollárra emelte a tétet – ennyit ajánlott a sejtés bizonyításáért. 2007-ben Kurtz és Simon János (Conway korábbi munkájára alapozva) igazolták [2], hogy a probléma természetes általánosítása algoritmikusan eldönthetetlen.

A számítógépes verifikáció területén Tomás Olivera e Silva jelentős eredményeket ért el [4, 5]. 1996 augusztusától 2000 áprilisáig futtatta C nyelven írt programját két 133 MHz-es és két 266 MHz-es DEC

*Astron Informatikai Kft., Budapest

**2007–2013

Alpha processzoros gépen, amely $100 \cdot 2^{50}$ -ig ellenőrizte a sejtést. Egy újabb hosszú futtatás vette kezdetét 2004 júniusában az előzőnél kb. háromszor gyorsabb algoritmussal, amely 2009 januárjában ért véget, és $20 \cdot 2^{58} \approx 5.765 \cdot 10^{18}$ -nal bezárólag minden pozitív egészre igazolta az állítást. Eric Roosendaal 2011 májusában tovább javította a rekordot: 2^{60} -ig jutott.

2. Alapfogalmak

Legyen f a következő, pozitív egész számokon értelmezett függvény:

$$f(n) = \begin{cases} 3n + 1 & \text{ha } n \text{ páratlan;} \\ \frac{n}{2} & \text{ha } n \text{ páros.} \end{cases}$$

Definiáljuk az (a_i) rekurzív sorozatot az alábbi módon (n tetszőleges pozitív egész):

$$a_i = \begin{cases} n & \text{ha } i = 1; \\ f(a_{i-1}) & \text{ha } i > 1. \end{cases}$$

A Collatz-sejtés (szokás $3n + 1$ vagy $3x + 1$ sejtésnek is nevezni) azt állítja, hogy az (a_i) sorozat tartalmazza az 1-et, függetlenül az első elemtől. Más szavakkal: tetszőleges pozitív egészről kiindulva, az f függvény kellően sokszori iterációja után 1-et kapunk. Nyilvánvaló, hogy ha az 1 előfordul a sorozatban, akkor onnantól kezdve az $(1, 4, 2)$ ciklus ismétlődik a végtelenségig.

Néhány példa különböző kiindulási értékekre [7]:

- $n = 6$: 6, 3, 10, 5, **16**, 8, 4, 2, 1;
- $n = 11$: 11, 34, 17, **52**, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1;
- $n = 27$: 27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, **9232**, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488, 244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1.

Láthatjuk, hogy viszonylag kis n -re (27) is elég sok (111) lépést vehet igénybe, hogy eljussunk az 1-hez. A sorozatban előforduló legnagyobb szám a 9232.

Definiáljuk a következő (f -hez nagyon hasonló) T függvényt a pozitív egész számok halmazán:

$$T(n) = \begin{cases} \frac{3n+1}{2} & \text{ha } n \text{ páratlan;} \\ \frac{n}{2} & \text{ha } n \text{ páros.} \end{cases}$$

Könnyen látható, hogy ha a Collatz-sejtésben az f függvény helyett T -t írunk, ekvivalens állítást kapunk. Ugyanis ha n páratlan, akkor a $3n+1$ szabály alkalmazása után páros lesz, így a következő lépésben felezni kell.

Az n pozitív egész *trajektóriája* a következő sorozat: $(T^{(k)}(n))_{k=0}^{\infty}$, ahol $T^{(k)}(n)$ a T függvény k -adik iterációja n -re alkalmazva. n *megállási ideje*, $\sigma(n)$ a legkisebb pozitív k , amelyre $T^{(k)}(n) < n$, ha létezik; különben végtelen. A trajektória *legnagyobb kitérése* – jelölés: $t(n)$ – a maximális értéke $T^{(k)}(n)$ -nek $k \geq 0$ esetén, ha létezik; különben végtelen.

Az $n > 1$ egészt σ -rekordtartónak nevezzük, ha $\sigma(m) < \sigma(n)$ minden $1 < m < n$ esetén, vagyis az n -nél kisebb (és 1-nél nagyobb) egészek megállási ideje is kisebb. Hasonlóan, n -t t -rekordtartónak nevezzük, ha $t(m) < t(n)$ minden $1 < m < n$ esetén.

3. Számítási eredmények

3.1. Algoritmus rekordtartók keresésére

A következőkben összefoglalom Silva eredményeit [4].

Legyen $n_0 = 2^k n_k + m_k$, ahol $n_k = \lfloor \frac{n_0}{2^k} \rfloor$ és $m_k = n_0 \bmod 2^k$. Ekkor a következő állítás teljesül:

3.1. Állítás.

$$T^{(k)}(n_0) = 3^{y(k; m_k)} n_k + T^{(k)}(m_k) \quad (k \geq 0),$$

ahol $y(k; m_k)$ a páratlan elemek száma a $(T^{(i)}(m_k))_{i=0}^{k-1}$ trajektóriában.

(a)		(b)	
n	$t(n)$	n	$\sigma(n)$
2	2	2	1
3	8	3	4
7	26	7	7
15	80	27	59
27	4 616	703	81
255	6 560	10 087	105
447	19 682	35 655	135
639	20 762	270 271	164
703	125 252	362 343	165
1 819	638 468	381 727	173

1. táblázat. (a) Az első 10 t -rekordtartó, (b) Az első 10 σ -rekordtartó

Könnyen belátható, hogy az állítás igaz. Minden lépésben osztunk 2-vel, így $2^k n_k$ -ból n_k lesz. Mivel n_0 és m_k kongruensek modulo 2^k , ezért a trajektóriájukban az első k elem paritása megegyezik. Így pontosan annyszor kell 3-mal szorozni n_k -t, ahány páratlan elem van a $(T^{(i)}(m_k))_{i=0}^{k-1}$ trajektóriában (a $+1$ -ek benne vannak $T^{(k)}(m_k)$ -ban).

Vegyük a 7, 39 és 103 számokat, melyek modulo 32 kongruensek, és nézzük meg, hogyan viselkednek az iteráció első 5 lépésében (2. táblázat). Tehát most $k = 5$, $m_k = 7$ és $n_k = 0, 1, 3$.

Észrevehetjük egyrészt, hogy a három szám paritássorozatának első 5 eleme megegyezik, a hatodik viszont már nem (7-nél páros, 39-nél és 103-nál páratlan). Másrészt

$$T^{(5)}(39) = 3^4 \cdot 1 + T^{(5)}(7) = 81 + 20 = 101$$

és

$$T^{(5)}(103) = 3^4 \cdot 3 + T^{(5)}(7) = 243 + 20 = 263,$$

vagyis a fenti képlet valóban igaz erre a konkrét esetre ($y(k; m_k) = 4$, ugyanis a 7 trajektóriájának első 5 eleme közül 4 páratlan).

Nézzünk egy másik példát (3. táblázat). Most az 573, 3645 és 9789 számokból indulunk ki, melyek modulo 1024 kongruensek. Így a paritássorozatuk első 10 eleme megegyezik, míg a trajektória 11. eleme az 573-nál páratlan, a 3645-nél és 9789-nél viszont páros.

	1	2	3	4	5
7	11	17	26	13	20
39	59	89	134	67	101
103	155	233	350	175	263

2. táblázat. Modulo 32 kongruens számok iterációja

	1	2	3	4	5	6	7	8	9	10
573	860	430	215	323	485	728	364	182	91	137
3645	5468	2734	1367	2051	3077	4616	2308	1154	577	866
9789	14684	7342	3671	5507	8261	12392	6196	3098	1549	2324

3. táblázat. Modulo 1024 kongruens számok iterációja

A képlet most is igaz:

$$T^{(10)}(3645) = 3^5 \cdot 3 + T^{(10)}(573) = 729 + 137 = 866,$$

továbbá

$$T^{(10)}(9789) = 3^5 \cdot 9 + T^{(10)}(573) = 2187 + 137 = 2324.$$

3.2. Állítás. $k \geq 0$ esetén

$$T^{(k)}(m_k) < 3^{y(k;m_k)}.$$

Teljes indukcióval bizonyítható az állítás.

Összefoglalva, az előző két állítás azt mondja ki, hogy $T^{(k)}(n_0)$ reprezentációja 3-as alapú számrendszerben nagyon egyszerű: az utolsó $y(k; m_k)$ jegyet $T^{(k)}(m_k)$ határozza meg, a többit pedig n_k . Ennek az észrevételnek az általánosításáról szól a következő állítás.

3.3. Állítás. $0 \leq p \leq k$ esetén

$$T^{(p)}(n_0) = 2^{k-p} \cdot 3^{y(p;m_k)} n_k + T^{(p)}(m_k)$$

és

$$T^{(p)}(m_k) < 2^{k-p} \cdot 3^{y(p;m_k)} n_k.$$

Az iteráltak nagyságát tekintve Silva a következő észrevételt tette:

3.4. Állítás. $1 \leq k \leq 40$ és $1 < m_k < 2^k$ esetén ha a

$$2^{k-p} \cdot 3^{y(p;m_k)}, \quad 0 \leq p \leq \min\{k, \sigma(m_k)\}$$

számokat növekvő sorrendbe rendezzük, akkor a megfelelő

$$T^{(p)}(m_k), \quad 0 \leq p \leq \min\{k, \sigma(m_k)\}$$

számok is növekvően lesznek rendezve.

Ennek egy következménye:

3.5. Állítás. Ha $1 \leq k \leq 40$, $1 < m_k < 2^k$ és $n_k \geq 0$, akkor a $T^{(p)}(m_k)$ számok relatív sorrendje megegyezik a $T^{(p)}(2^k \cdot n_k + m_k)$ számok relatív sorrendjével $0 \leq p \leq \min\{k, \sigma(m_k)\}$ -ra.

Az állítást ellenőrizhetjük is az előbbi példákön. A 2. táblázatban az iteráltak növekvő sorrendje a következő: 0, 1, 4, 2, 5, 3; míg az 3. táblázatban: 9, 10, 8, 3, 4, 7, 2, 5, 0, 6, 1. Ugyan az állítás csak a megállási időnél nem nagyobb p -re van bizonyítva (vagyis az 3. táblázat esetén $p \leq 2$ -re), de Silva sejtése szerint $p > \sigma(m_k)$ -ra is igaz, ha $T^{(p)}(m_k) > 1$. Ebben a konkrét esetben láthatjuk, hogy a megállási időnél nagyobb p -re is teljesül az állítás.

A $2^k \cdot n_k + m_k$ szám p -edik és 0-adik iteráltjának összehasonlításából adódik a következő állítás:

3.6. Állítás. Legyen $\sigma(m_k) = p \leq k \leq 40$ és $1 < m_k < 2^k$. Ekkor a $2^k \cdot n_k + m_k$ szám megállási ideje minden $n_k \geq 0$ -ra pontosan p .

Ebből következik, hogy ha σ -rekordtartókat keresünk, akkor a $2n$, illetve $4n + 1$ ($n > 0$) alakú számokkal nem kell foglalkoznunk. Ugyanis $T(2n) = n$, vagyis a megállási idő 1, míg $T(4n + 1) = 6n + 2$, $T(6n + 2) = 3n + 1$, tehát $\sigma(4n + 1) = 2$. Ezzel a negyedére csökkentettük a jelöltek számát, hiszen csak a $4n + 3$ alakúakat kell vizsgálnunk.

Készítsünk egy 40 mélységű bináris fát, melynek k mélységű csúcsai a modulo 2^k maradékosztályokat reprezentálják (így a levelek a modulo 2^{40} maradékosztályoknak felelnek meg). Egy k mélységű csúcsot zártnak nevezünk, ha az általa reprezentált maradékosztály elemeinek megállási ideje legfeljebb k . Az eddigiek alapján zárt csúcs pl. a $2n$ és a $4n + 1$. Hasonlóan, a $16n + 3$ is zárt csúcs, hiszen 4 mélységű, és a megállási ideje is 4: $T(16n + 3) = 24n + 5$, $T(24n + 5) = 36n + 8$, $T(36n + 8) = 18n + 4$,

$T(18n+4) = 9n+2$. A nem zárt csúcsokat nyíltak nevezzük. A bináris fát úgy építjük fel, hogy csak a nyílt csúcsokból ágazunk tovább.

Silva számításai szerint a 40. szinten 6 402 835 000 nyílt csúcs van. Ennyi tehát azoknak a modulo 2^{40} maradékosztályoknak a száma, melyek megállási ideje 40-nél nagyobb. Ez az összes maradékosztály 0.5823%-a.

A nyílt csúcsok száma olyan nagy, hogy nem érdemes őket eltárolni és az általuk reprezentált maradékosztályok szerint növekvő sorba rendezni. Helyette a generálásuk időrendje szerint vizsgáljuk őket – így lényegesen kevesebb memóriára van szükség.

A program két listát tart nyilván: egyet a t -rekordtartójelölteknek, egyet pedig a σ -rekordtartójelölteknek. Minden egyes nyílt csúcsához tartozó maradékosztálynak 2^{10} egymást követő elemét teszteljük növekvő sorrendben. Felírjuk m_{40} első 40 iteráltját, majd az 3.1. állítás felhasználásával kiszámoljuk $T^{(40)}(n_{40} \cdot 2^{40} + m_{40})$ -t ($n_{40} = 0, 1, \dots, 2^{10} - 1$). Mivel n_{40} értékei egyesével növekednek, ezért a $T^{(40)}(n_{40} \cdot 2^{40} + m_{40})$ elemek számtani sorozatot alkotnak $3^{y(40; m_{40})}$ differenciával. Így az első esetet leszámítva csak egy-egy összeadásra van szükség 2^{10} elem 40. iteráltjának meghatározásához.

Két probléma van ezzel a keresési stratégiával. Egyrészt a 41-nél kisebb megállási idejű σ -rekordtartókat nem jegyezzük fel. Ez könnyen orvosolható: mivel $\sigma(27) = 59$, ezért elég tesztelni a 2 és 27 közötti számokat kiindulási értéként, hogy megtaláljuk a hiányzó rekordtartókat. A másik probléma a t -rekordtartókkal kapcsolatos: nagyon költséges kiszámolni az első 40 iterált maximumát minden nyílt csúcsra. Ez az információ csak kis kiindulási értékek esetén fontos, ezért a program nem is foglalkozik vele. Így viszont ha a legnagyobb kitérés még a 40. iterált előtt bekövetkezik, a t -rekordtartók listája pontatlan lehet.

Legyen c_k a k -adik t -rekordtartójelölt. Mivel

$$T^{(k)}(n) \leq \frac{3^k - 1}{2^k - 1} n,$$

ezért ha

$$\frac{3^{40} - 1}{2^{40} - 1} c_{k+1} < t(c_k),$$

akkor nem lehet t -rekordtartó c_k és c_{k+1} között, melynek legnagyobb kitérése az első 40 iterációban bekövetkezik. Ez alapján megkeressük az

utolsó jelöltet a t -rekordtartójelöltek listájában, amely nem elégíti ki a fenti egyenlőtlenséget, és megvizsgáljuk az ezt követő jelöltnél kisebb kiindulási értékeket. Így módon a 319 804 831-nél kisebb kiindulási értékeket kell vizsgálni. (Mint kiderült, ez egy nagyon óvatos becslés. A legnagyobb t -rekordtartó, mely a 40. iterált előtt eléri a maximális kitérést, a 704 511.)

Van még egy optimalizálási lehetőség: azokat a kiindulási értékeket figyelmen kívül hagyhatjuk, amelyek rajta vannak egy kisebb kiindulási érték trajektóriáján (ez egy speciális esete az ún. trajektóriaegyesítésnek). Pl. $T(2n+1) = 3n+2$, ezért $3n+2$ nem lehet σ - vagy t -rekordtartó semmilyen n -re. Hasonlóan, mivel $T^{(3)}(8n+3) = 9n+4$, ezért a $9n+4$ alakú számokat is kizárhatjuk. További eseteket úgy kaphatunk, hogy a $T^{-1}(n)$ inverz függvényt iteráljuk valamelyik modulo 3^k kongruenciaosztályhoz tartozó kiindulási értékre, amíg nála kisebbet nem kapunk. Zártnak nevezzük azokat a kongruenciaosztályokat, melyekre ez bekövetkezik. $k=2$ esetén például a kongruenciaosztályok 4/9-e zárt.

3.1.1. Hatékonysági elemzés

Most azt vizsgáljuk, Silva programja mennyivel gyorsabb a naiv módszernél, mikor is egyesével teszteljük az összes kiindulási értéket, hogy t - vagy σ -rekordtartók-e. Minden n -re addig iterálunk, amíg $T^{(k)}(n) < n$ nem teljesül – így az iterációk száma egyenlő a megállási idővel. Legyen $n_c(k)$ a k mélységű zárt csúcsok száma, $n_o(k)$ pedig a k mélységű nyílt csúcsok száma. Ekkor az átlagos megállási időt a következő képlet adja meg:

$$\sigma_{naiv} = \sum_{k=1}^{\infty} k \frac{n_c(k)}{2^k}.$$

Ennek egy becslült értéke:

$$\sigma_{naiv} \approx 3.493.$$

Másrészt a 40 mélységű nyílt csúcsokat reprezentáló kongruenciaosztályok esetén szükséges további iterációk száma:

$$\sigma_{40} = \frac{2^{40}}{n_o(40)} \sum_{k=41}^{\infty} (k-40) \frac{n_c(k)}{2^k},$$

becsülve:

$$\sigma_{40} \approx 18.674.$$

Így a gyorsítás a naiv módszerhez képest:

$$\frac{2^{40}}{n_o(40)} \frac{\sigma_{naiv}}{\sigma_{40}} \approx 32.118.$$

Még a trajektóriaegyesítés gyorsító hatásának elemzése van hátra. Mivel 2^k és 3^p relatív prímek minden nemnegatív k, p -re, ezért a modulo 2^k (3^p) kongruenciaosztályokhoz tartozó egészek egyenletesen oszlanak el a modulo 3^p (2^k) kongruenciaosztályok között. Speciálisan, a 2^{10} kiindulási érték $(n_{40} \cdot 2^{40} + m_{40})$, amelyet minden nyílt csúcsnál tesztlünk, (majdnem) egyenletesen oszlik el a modulo 9 maradékosztályok között. Azt állítjuk, hogy az egyes maradékosztályokhoz tartozó átlagos iterációs szám mind a 9 maradékosztály esetén megegyezik. Ez abból következik, hogy az n_{40} számok (melyek arról döntenek, hogy n_0 melyik modulo 9 maradékosztályba tartozzon) egyenletesen oszlanak el a modulo 2-hatvány maradékosztályok között. Ezért a trajektóriaegyesítés független az eddigi optimalizációktól, vagyis egy extra 9/5-ös gyorsító faktort ad (a 9 maradékosztályból csak 5-öt kell vizsgálni). Így a teljes gyorsulási tényező kb. 57.813.

3.1.2. További észrevételek

A t -rekordtartók legnagyobb kitérésével kapcsolatban Silva az alábbi sejtést fogalmazta meg:

3.7. Sejtés. *A $t(n)$ legnagyobb kitérésre a következő egyenlőtlenség teljesül:*

$$t(n) < n^2 f(n),$$

ahol $f(n)$ konstans, vagy egy nagyon lassan növvő függvénye n -nek.

$t(n) > n^2$ csak hét t -rekordtartóra teljesül $n < 3 \cdot 2^{53}$ esetén, de a $t(n) < 8n^2$ egyenlőtlenséget ezek is kielégítik. Így $t(n)$ jó közelítéssel négyzetes függvénye n -nek, ahol az értelmezési tartomány a t -rekordtartók halmaza.

A teljességhez hozzátartozik annak vizsgálata is, hogyan növekszik a σ -rekordtartók megállási ideje. Itt azonban nem figyelhető meg olyan szabályosság, mint a t -rekordtartók legnagyobb kitérésénél.

Fontos eredménye Silva cikkének a Collatz-sejtés igazolása $n = 3 \cdot 2^{53} \approx 2.702 \cdot 10^{16}$ -ig.

3.2. Egy hatékonyabb módszer

A fenti algoritmust Silva továbbfejlesztette, így még nagyobb számokra igazolta a Collatz-sejtést [5].

Az új algoritmus 46 mélységű bináris fát használ. Ez nagyobb gyorsulási tényezőt eredményez, mint a 40 mélységű fa (32.118 helyett 44.866). Silva táblázatba foglalta, hogy d mélységű fa esetén átlagosan hány iteráció szükséges a d mélységű nyílt csúcsok lezárásához. Egy másik oszlopban csak azokat a d mélységű nyílt csúcsokat vizsgálta, amelyek trajektóriájában minimális számú ($k = \left\lceil d \frac{\log 2}{\log 3} \right\rceil$) páratlan ág van. $d = 46$ és $k = 30$ esetén az átlagos iterációs szám 9.721, ami relatíve magas. Többek között ezen szempont alapján választotta Silva a fa mélységének 46-ot.

3.8. Megjegyzés. *Némi magyarázatra szorul, hogy miért $k = \left\lceil d \frac{\log 2}{\log 3} \right\rceil$ a páratlan ágak minimális száma a d mélységű nyílt csúcsokat tekintve. Nagy páratlan számok esetén a T függvény alkalmazása jó közelítéssel egy $\frac{3}{2}$ -del való szorzást jelent, míg páros számoknál 2-vel való osztást. Nyílt csúcsról lévén szó, a d -edik iteráció után kapott érték nagyobb, mint a kiindulási. Így ha k -val jelöljük a trajektória páratlan elemeinek számát, akkor a következő egyenlőtlenségnek kell teljesülnie:*

$$\left(\frac{3}{2}\right)^k \cdot \left(\frac{1}{2}\right)^{d-k} > 1$$

$$\frac{3^k}{2^d} > 1$$

$$3^k > 2^d$$

$$k \log 3 > d \log 2$$

$$k > d \frac{\log 2}{\log 3}.$$

Az előző algoritmusnál már volt szó a trajektóriaegyesítés egy speciális esetéről. Egy hasonló optimalizálási lehetőség, amely Eric Roosendaal nevéhez fűződik, annak észrevétele, hogy n és $n - 1$ trajektóriája is gyakran egyesül. Például:

$$\begin{aligned}
&64n + 14 \rightarrow 32n + 7 \rightarrow 48n + 11 \rightarrow 72n + 17 \rightarrow 108n + 26 \rightarrow \\
&54n + 13 \rightarrow 81n + 20, \\
&64n + 15 \rightarrow 96n + 23 \rightarrow 144n + 35 \rightarrow 216n + 53 \rightarrow 324n + 80 \rightarrow \\
&162n + 40 \rightarrow 81n + 20.
\end{aligned}$$

Így a levelek kb. 17.5%-ának vizsgálata elhagyható.

A fő különbség a mostani és az előző algoritmus között, hogy az előzőnél a fa kiértékelése után egyesével iteráltunk. Most viszont egyszerre több iterációt végzünk, így az algoritmus még gyorsabb lesz. Az egyszerre elvégzendő iterációk maximális száma legyen Δ . Ha az i -edik iteráltnál tartunk, és a következő lépésben δ iterációt akarunk végezni, akkor a következő feltételeknek teljesülniük kell:

- $T^{(i+j)}(n) \leq \tau \max(T^{(i)}(n), T^{(i+\delta)}(n))$ minden $0 < j < \delta$ esetén, ahol $\tau \geq 1$ a toleranciatényező;
- ha $T^{(i+j)}(n) < T^{(i)}(n)$ néhány $j \leq \delta$ esetén, akkor minden $0 \leq j < \delta$ -ra $T^{(i+\delta)}(n) < T^{(i+j)}(n)$.

Az első feltétel azt jelenti, hogy egyetlen átugrott iterált sem lehet nagyobb, mint egy vizsgált iterált τ -szorososa. A második feltétel pedig azt biztosítja, hogy ha az iteráció során a kiindulási érték alá megyünk, akkor az utolsó iterált kisebb, mint az összes többi.

Az 3.1. állítás alapján minden modulo 2^Δ maradékosztályra készítünk egy táblázatot δ , $3y^{(\delta; m_\delta)}$ és $T^{(\delta)}(m_\delta)$ értékeivel. Így gyorsan meg tudjuk határozni $T^{(i+\delta)}(n)$ -t. Silva programjában a $\Delta = 14$ és $\tau = 10$ értékeket használta.

A futtatás előtt meghatároztak egy legnagyobb kitérés küszöböt és egy megállási idő küszöböt. Mivel a rekordtartók kivételesen nagy $t(n)$, illetve $\sigma(n)$ értékkel rendelkeznek, ezért minden n kiindulási értékre elég annyit biztosan tudni, hogy $t(n)$ és $\sigma(n)$ az adott küszöb felett van-e. Amennyiben igen, akkor egy lassabb, de pontos algoritmussal kiszámíthatjuk a kérdéses értékeket. Ha pedig nem, akkor nem kell az adott n -nel foglalkoznunk, mert biztosan nem rekordtartó. Természetesen az utóbbi eset jóval gyakoribb.

A legnagyobb kitérés küszöböt az addigi legnagyobb kitérés ezredének (mivel $\tau = 10$, a tizede is elég lett volna), míg a megállási idő küszöböt az addigi legnagyobb megállási időnél 200-zal kisebbnek választották.

Az algoritmus segítségével minden modulo 2^{46} maradékosztálynak 2^{12} egymást követő elemét tesztelték – ez összesen $2^{46+12} = 2^{58}$

egész. Közülük a korábban említett optimalizációs lehetőségek alapján elhagyták a redundáns elemeket. A megmaradt kiindulási értékeket addig iterálták, amíg a következő három feltétel valamelyike be nem következett:

- az iterációk száma átlépte a megállási idő küszöböt;
- az aktuális iterált meghaladta a legnagyobb kitérés küszöböt;
- az aktuális iterált a kiindulási érték alá csökkent.

Az első két esetben további vizsgálatra van szükség a pontos megállási idő, ill. legnagyobb kitérés meghatározásához. Mivel ezek a kivételesen nagy értékek ritkán fordulnak elő, ezért a tesztelésükhöz szükséges idő nem jelentős.

A programot 20, egyenként 2^{58} egészből álló intervallumra futtatták négy és fél éven keresztül. Ennek eredményeképp bizonyítást nyert a következő tétel:

3.9. Tétel. *A $3x + 1$ sejtés minden $x \leq 20 \cdot 2^{58}$ -ra igaz.*

Hivatkozások

- [1] J. H. Conway, Unpredictable Iterations, *Proceedings of the 1972 Number Theory Conference: University of Colorado*, 1972, pp. 49–52.
- [2] S. A. Kurtz, J. Simon, The Undecidability of the Generalized Collatz Problem, *Proceedings of the 4th International Conference on Theory and Applications of Models of Computation, TAMC 2007*, pp. 542–553.
- [3] J. C. Lagarias, *The $3x + 1$ Problem: An Annotated Bibliography (1963–1999)*, <http://arxiv.org/pdf/math/0309224v13.pdf>, 2012. december.
- [4] T. O. e Silva, Maximum Excursion and Stopping Time Record-Holders for the $3x + 1$ Problem: Computational Results, *Mathematics of Computation*, 68(225) (1999), pp. 371–384.

-
- [5] T. O. e Silva, Empirical Verification of the $3x + 1$ and Related Conjectures, *The Ultimate Challenge: The $3x + 1$ Problem*, American Mathematical Society, 2010, pp. 189–207.
 - [6] J. Simons, B. de Weger, Theoretical and computational bounds for m -cycles of the $3n + 1$ -problem, *Acta Arithmetica*, 117(1) (2005).
 - [7] http://en.wikipedia.org/wiki/Collatz_conjecture, 2012. december.
 - [8] <http://mathworld.wolfram.com/CollatzProblem.html>, 2012. december.
 - [9] <http://www.ieeta.pt/~tos/3x+1.html>, 2012. december.

A programok helyességéről

Kozma László

Eötvös Loránd Tudományegyetem, Informatikai Kar

`kozma@ludens.elte.hu`

1. Bevezetés

A programok helyességének vizsgálata a tárolt programú Neumann-elvű számítógépek megjelenése óta fontos és teljes körűen máig meg nem oldott probléma. A programok helyességét vizsgálhatjuk informális módon és használhatunk a verifikáció során formalizált eszközöket. Megvizsgáljuk azt a kérdéskört röviden, hogy milyen eszközök, módszerek kellenek programok helyességének eldöntéséhez.

Első és legfontosabb, hogy a programozási nyelvek pontos szintaxissal [42] és szemantikával rendelkezzenek [38–40]. Informális helyességvizsgálathoz ez már elég is lehet, ha megelégszünk az úgynevezett viselkedéselemzés használatával a programunk vizsgálatakor [29]. Formalizált módszerek esetében továbbá szükség van egy logikai kalkulusra [8, 15, 16, 33, 35], amelyet a módszertől függően speciális elemekkel kell kiegészítenünk.

A kész program helyességét vagy egy módszer segítségével bizonyítjuk [11, 14, 19, 21, 25–27], vagy a helyes programot egy módszerrel származtatjuk, ezek a szintézis módszerek [1, 4, 6, 13, 17, 22–24, 32, 41]. Helyes párluzamos programok előállítását szolgáló szintézis módszerek általában a tabló módszeren alapulnak [8, 15].

A helyességvizsgálat szempontjából új megközelítést jelentettek a modellellenőrzők, amelyeket eredetileg a hardvertervező mérnökök számára fejlesztettek ki [34, 36, 37]. Tanulmányunkban röviden ezeket az elemeket, módszereket vesszük sorra abból a szempontból, hogy milyen

szerepük van a helyes programok előállításában. Az elmondottakat a szakirodalomból jól ismert, a párhuzamos programozás terén felmerült alapproblémával, két vagy több folyamat kölcsönös kizárását megoldó folyamatok helyességének vizsgálatával illusztráljuk [30, 31].

A következő fejezetekben röviden tárgyaljuk a programozási nyelvek szintaxisát és szemantikáját, majd a logikai kalkulusok szerepét a helyességbizonyításban és a helyes programok származtatásában. Röviden összefoglaljuk a különböző módszerek lényegét és a már említett példákon keresztül mutatjuk be az egyes módszerek előnyeit és hátrányait.

2. Programozási nyelvek szintaxisa és szemantikája

A továbbiakban a számítógépek történetében először kifejlődött úgynevezett imperatív vagy más néven procedurális nyelvekkel és az ilyen nyelveken készült programok vizsgálatával foglalkozunk, és nem vizsgáljuk például a logikai és funkcionális nyelveket, sem az újabb keletű alkalmazások számára kidolgozott speciális nyelveket.

A procedurális programozási nyelvek a Chomsky által definiált formális nyelvosztályok rendszerében a környezetfüggő nyelvek közé tartoznak [42]. Környezetfüggő nyelvtanokkal egy nyelv szintaktikus és szemantikus tulajdonságait is ki tudjuk fejezni. A programozási nyelvek absztrakciós szintje azonban jóval magasabb, mint a számítógép anyanyelvének, a bináris gépi kódnak a szintje. Ezért szükség van fordítóprogramokra, amelyek segítségével a magas szintű programnyelven megírt programokat transzformáljuk a számítógép anyanyelvére.

Hamar kiderült, hogy fordítóprogramokat a környezetfüggetlen nyelvekre építve érdemes elkészíteni, mert ezekhez lehet a szöveg szintaktikus elemzését megkönnyítő levezetési fát generálni. Az első magas szintű programozási nyelvek (Algol 60, Fortran) [38] fordítóprogramjai környezetfüggetlen nyelvtani leírások alapján készült elemzőket használtak. Ez azt jelentette, hogy számos nyelvi tulajdonság nem volt leírható környezetfüggetlen szabályok segítségével. Ezeket a tulajdonságokat a nyelv definiálói a szemantika körébe utalták. Ilyenek voltak például a láthatóságot biztosító nyelvi elemek.

Kezdetben a szemantikát informális módon definiálták: például az

Algol 60 esetében természetes nyelvi mondatokkal fejezték ki, írták le az egyes nyelvi elemek jelentését [44]. Ez nagyon hamar oda vezetett, hogy a nyelvek különböző nyelvjárásai alakultak ki, mivel a természetes nyelvű leírások esetenként többféle módon is értelmezhetők voltak. A nyelvjárások lehetetlenné tették a később nagyon fontossá váló célt, a programok hordozhatóságát.

További problémát jelentett a hordozhatóság szempontjából, az hogy például az Algol 60 esetében az input-output utasításokat szintaxis szintjén sem definiálták. Ez később azt eredményezte, hogy az Algol 60 nyelv háttérbe szorult a Fortran nyelvvel szemben [45] a tudományos alkalmazásokban, és megmaradt algoritmusok publikálását szolgáló nyelvnek. Az Algol 68 nyelv volt az első olyan magas szintű programozási nyelv, amelyet a nyelv implementációja előtt már pontosan formalizáltan definiáltak [43].

A fordítóprogramok segítségével szintaktikusan helyes programokat tudunk előállítani [62], de még messze nem biztos, hogy ezek szemantikusan is helyesek. A szemantikai vizsgálatok céljaira az idők folyamán több formalizált módszer alakult ki [39, 40, 49]. A nyelv definiálói számára fontos volt, hogy a definiálandó nyelv legáltalánosabb tulajdonságait például az egyes elemek konzisztenciáját a teljes nyelv szempontjából vizsgálni tudják. Az ilyen szakemberek számára hozták létre a denotációs, vagy matematikai szemantikát [39, 40].

Ennek a leírásnak a legfontosabb tulajdonsága az úgynevezett kompozicionalitás elve, amely azt jelenti, hogy a szemantika definiálásakor megadjuk az egyszerű szintaktikus nyelvi elemek jelentését általában matematikai függvények, objektumok segítségével, és az összetett szintaktikus elemek jelentését valamilyen függvény kompozícióval definiáljuk. Ebben az esetben feltétel, hogy az összetett nyelvi elemek egyértelműen dekomponálhatók legyenek az őket felépítő egyszerű nyelvi elemekre. A denotációs szemantika esetében a programok jelentését tehát matematikai objektumokkal definiáljuk, amelyek a programok végrehajtásának jelentését reprezentálják. A denotációs szemantika nagyon jó eszköz programok statikus elemzésére (konstans propagálás, előjel és függőségi analízisek, stb.).

A denotációs szemantika egyik nagy hátránya az, hogy új nyelvi elemek beépítése esetén gyakran a szemantikai leírást is újra kell szervezni. Erre jó példa a kivételek kezelése. Egy kivételek kezelését nem támogató egyszerű nyelv szemantikája leírható direkt denotációs

szemantikával, de a kivételek és a feltétlen ugró utasítások jelentése már nem fejezhető ki benne. Erre a célra egy új denotációs szemantikát kellett kifejleszteni a maradék program fogalmán alapuló standard szemantikát [39]. Ugyancsak hátránya ennek a szemantikának az, hogy a konkurencia és a párhuzamosság nem, vagy csak nagyon bonyolultan fejezhető ki benne. Ennek alapvető oka, hogy a programok jelentését a denotációs szemantika az input-output relációval fejezi ki, ami azt jelenti, hogy a program jelentését azzal definiáljuk, hogy adott bemeneti adatokhoz a program milyen eredményt rendel. A konkurens és a párhuzamos programok alapvetően nem determinisztikusak, így az input mellett az eredményt a hozzá vezető végrehajtási útvonal is befolyásolja.

Az operációs, vagy műveleti szemantikák a fordítóprogramok készítői számára fontosak és absztrakt matematikai gépeken alapulnak [39]. Az első ilyen módszer a VDL nyelven alapult [53], majd ezt követték a természetes és strukturált szemantikák [39]. Előnyük, hogy ezeket kifejezetten a fordítóprogramok készítői számára hozták létre, nagyon megkönnyíti a munkájukat. Hátrányuk viszont, hogy nem támogatják a kompozicionalitás elvét.

Az absztrakt adattípusok az objektum elvű programok tervezésben hoztak lényegesen újat, elsősorban az absztrakció és az öröklődés fogalmának köszönhetően [28, 52]. Új problémaként jelentkezett az absztrakt adattípusok megvalósítása és azok helyességének témaköre szekvenciális [20, 21, 60] és párhuzamos programozási környezetben [19, 59, 61].

A cselekvés (action) alapú szemantikák a denotációs, a műveleti és az algebrai szemantikai leírások előnyeit egyesítik. Alkalmasak a modellvezérelt szoftverfejlesztést támogató nyelvek szemantikai problémáinak kifejezésére a dinamikus tulajdonságokat is beleértve [49–51].

A programozók számára hasznos az axiomatikus szemantika, amelyeken a programhelyesség-bizonyító módszerek alapulnak. Ezeket a későbbi fejezetekben bővebben tárgyaljuk.

3. Logikák és logikai kalkulusok

A klasszikus logika predikátumkalkulusa központi szerepet tölt be a programok helyességének vizsgálatában [15, 16]. Programok esetében a helyességet ugyanis nem önmagában vizsgáljuk, hanem valamilyen

specifikációra nézve bizonyítjuk, hogy az adott program helyes, vagy egy adott specifikációból kiindulva vezetjük le, származtatjuk, szintetizáljuk a helyes programot.

Programok helyességének bizonyításához a klasszikus logika predikátumkalkulusa mellett a temporális logikák különböző változatait használhatjuk [14]. Ez utóbbiak segítségével általában konkurens és párhuzamos programok tulajdonságait írjuk le, és/vagy helyes programokat generálunk [2, 3, 6–9, 17, 18, 22–24].

Egy helyességbizonyítási módszer három fő részből áll:

- egy általános részből, amely egy következtetési rendszert tartalmaz;
- egy speciális részből, amely a program értelmezési tartományának axiomatikus leírását tartalmazza, egész számok esetében például a Peano-axiómákat;
- egy logikai részből, amely a programozási nyelvhez rendelt axiómasémákat tartalmazza, ezen sémákból a program szövege alapján egy sor axiómát, következtetési szabályt származtathatunk.

Különböző kalkulusok és axiómarendszerek megválasztásával különböző helyességbizonyítási rendszerekhez jutunk [11, 14, 16, 25–27]. A predikátumkalkulus nagyon jól alkalmazható szekvenciális programok helyességének bizonyítására [25–27]. A predikátumkalkulus alkalmas eszköz párhuzamos programok helyességének bizonyítására is [11], de Owicki és Gries módszere elsősorban elméleti szempontból jelentős, mivel annak kimutatása, hogy a párhuzamos folyamatok bizonyításai egymással nem interferálnak, igen munkaigényes feladat. A továbbiakban röviden bemutatjuk a CTL* és az MPCTL* temporális logikák szintaxisát és informális szemantikáját, amelyek felhasználásával helyes párhuzamos programok származtathatók [23, 24].

3.1. A CTL*

A CTL* (Computational Tree Logic*) egy ítéletalapú elágazó idejű temporális logika (propositional branching-time temporal logic), amelynek formulái induktív módon definiálhatók [22–24, 35]:

- (S1) Minden atomi ítélet egy állapotformula;
- (S2) ha f, g állapot formulák, akkor $f \wedge g, \neg f$ is állapotformulák;

- (S3) ha f egy útformula, akkor Ef és Af állapotformulák.
(P1) Minden állapotformula egyúttal útformula is;
(P2) ha f, g útformulák, akkor $f \wedge g, \neg f$ is útformulák;
(P3) ha f, g útformulák, akkor $X_j f, Y_j f, fUg$ is útformulák.

A CTL* *szemantikája* röviden a következő:

Legyen $M = (S, R_{i1}, \dots, R_{ik})$, ahol

- S az állapotok egy megszámlálható halmaza. Minden állapot egy olyan leképezés, amely az atomi ítéletek halmazát a $\{true, false\}$ halmazra képezi le.
- $R_{ij} \subseteq S \times S$, egy bináris reláció, amely az ij szekvenciális folyamat átmeneteit adja meg.

Legyen $R = R_{i1} \cup \dots \cup R_{ik}$.

Egy *út* állapotok egy $(s_1, s_2, \dots)$ szekvenciája, amelyekre $\forall i, (s_i, s_{i+1}) \in R$.

Egy *teljes út* (*fullpath*) egy maximális út, azaz egy $(s_1, s_2, \dots)$ végtelen sorozat, kivéve, ha valamely s_k állapothoz nincs olyan s_{k+1} állapot amelyre $(s_k, s_{k+1}) \in R$.

Ef intuitív jelentése: létezik olyan maximális út, amelyre az f teljesül.

Af intuitív jelentése: f teljesül minden maximális útra.

$X_j f$ (*strong nexttime*) intuitív jelentése: az $X_j f$ formula teljesül egy s állapotban egy maximális úton, ha a P_j folyamat következő atomi tevékenységének végrehajtásával az s állapotból közvetlenül elérhető állapotban az f teljesül.

$Y_j f$ (*weak nexttime*) jelentése: gyenge rákövetkezés.

fUg formula teljesül egy maximális úton egy állapotban, ha létezik olyan s állapot a maximális úton, ahol g teljesül és f az s állapotig minden állapotban teljesül az úton, de az s állapotban már nem feltétlenül teljesül.

A CTL* formalizált szemantikája megtalálható például [23]-ban.

3.2. Az MPCTL*

Az MPCTL* (Many-Process CTL*) specifikációs nyelv a CTL* temporális logika egy kiterjesztése [35].

Az MPCTL* temporális logikát speciálisan sok hasonló folyamatból álló rendszer specifikálására dolgozták ki, ezért a specifikáció részét

képezi a folyamatok közötti kapcsolatokat kifejező ún. I *interconnection* reláció is.

$I \subseteq \{i_1, \dots, i_k\} \times \{i_1, \dots, i_k\}$, ahol $\{i_1, \dots, i_k\}$ a K folyamatból álló rendszer azon folyamatainak indexét tartalmazza, amelyeket specifikálni akarunk. Az $i I j$ reláció akkor és csak akkor áll fenn az i és a j folyamatok között, ha i és j között kapcsolat van. Az I reláció reflexív, szimmetrikus és totális, azaz minden folyamat legalább egy másik folyamattal összeköttetésben áll.

K folyamatból álló rendszerre több összekapcsolási séma definiálható, így a K folyamatból álló rendszer helyett a továbbiakban I rendszerről beszélünk (I -system).

Legyen $AP = \{AP_{i_1}, \dots, AP_{i_k}\}$ atomi formulák egy indexelt családja. Az AP_i halmaz formulái csak indexeikben különböznek az AP_j halmaz formuláitól. Például, $AP_i = \{Q_i, R_i\}$ és $AP_j = \{Q_j, R_j\}$.

Egy térbeli (spatial) modalitást $\wedge_i$ vagy $\wedge_{ij}$ jelöl. $\wedge_i$ modalitás az i folyamatindexre vonatkozik, míg a $\wedge_{ij}$ modalitás azon i és j folyamatindexekre vonatkoznak, amelyeket az I reláció kapcsol össze. Ha a spatial (térbeli) modalitás $\wedge_i$, akkor a CTL* formulában csak az AP_i halmazból való atomi formulák megengedettek. Ha a spatial modalitás $\wedge_{ij}$, akkor a CTL* formulában csak az $AP_i \cup AP_j$ halmazból való atomi formulák megengedettek.

Ezen temporális logikák alkalmazására a további fejezetekben adunk példát.

4. Programok helyességének bizonyítására vonatkozó módszerek

Először a szekvenciális programok helyességének bizonyítására vonatkozó módszerek jelentek meg az 1960-as évek végén. R. W. Floyd 1967-ben publikálta módszerét folyamatábra programokra [25]. C. A. R. Hoare 1969-ben közölte módszerét, amely már a strukturált programok helyességének bizonyítására is alkalmas volt [26]. Ezek a módszerek elméleti szempontból úttörőek voltak és alapozó jellegük miatt az oktatás szempontjából napjainkban is fontosak. Floyd és Hoare módszereinek közös vonása, hogy a program bizonyos pontjaihoz invariáns állításokat rendelünk, amelyeknek minden olyan esetben igaznak kell lenniük, amikor a vezérlés a program adott pontjához érkezik. Számos további

módszert dolgoztak ki a szakemberek a hetvenes években a szekvenciális programok helyességének bizonyítására. Ezek egyike R. M. Burstall módszere [27], amely abban különbözik Floyd és Hoare módszerétől, hogy a program vágási pontjaihoz nem invariáns állításokat rendel, hanem olyan állításokat, amelyek csak bizonyos időpontokban kell, hogy teljesüljenek. Burstall módszerének hiányossága, hogy a program helyességét biztosító tételek bizonyítására nem tudott egzakt módszert adni. Megjegyezzük, hogy Burstall módszere teljesen formalizálható, ha a program vágási pontjaihoz rendelt állításainkat temporális logikai formulákkal adjuk meg.

A strukturált programozás bevezetése a gyakorlati programozásban egy kísérlet volt az úgynevezett szoftverkrízis legyőzésére. Ebben a folyamatban fontos szerepe volt E W. Dijkstrának [10]. A szoftverkrízis makacsnak bizonyult és a legyőzésére tett kísérletek vezettek el az objektum elvű programozáshoz [28]. Más kérdés, hogy a szoftverkrízist abban az értelemben még napjainkban sem tudtuk legyőzni, hogy a minden szempontból teljesen megbízható és helyes programot garantáló módszerek még a mai napig sem születtek meg.

S. Owicki és D. Gries az 1970-es évek közepén terjesztették ki Hoare módszerét párhuzamos programok helyességének bizonyítására [11]. A különböző temporális logikák megjelenésével számos helyességbizonyítási módszer született párhuzamos programok helyességének igazolására. Sajnos, ezen módszerek alkalmazásával is csak laboratóriumi méretű programok helyességét tudjuk kimutatni.

A gyakorlatban bizonyos korlátok között jól használható módszer a konkurens és párhuzamos programok helyességének vizsgálata viselkedéselemzéssel. Ennek lényege, hogy a programunk végrehajtási útjait végigkövetjük és megvizsgáljuk, hogy a helyesség szempontjából fontos kritériumaink teljesülnek-e.

Konkurens és párhuzamos programok esetében a programokat a biztonságosság és az elevenség szempontjából vizsgáljuk. Legáltalánosabban fogalmazva egy program *biztonságos*, ha semmi rossz nem történik és *eleven* akkor, ha valami jó előbb utóbb bekövetkezik.

A komponens alapú programozási paradigmán belül a kontraktumok használata új lendületet adott a helyességbizonyítási módszerek alkalmazásának. A komponens alapú rendszerek helyességét két szintre bonthatjuk, mivel vizsgálnunk kell a komponensek helyességét és külön lépésben bizonyíthatjuk a helyes komponensekből felépített rendszer

helyességét. Ez már lényeges egyszerűsítést jelent a helyességvizsgálat szempontjából [58, 63, 64].

További egyszerűsítést hozott a kontraktusok fogalmának bevezetése a szoftvertechnológiában és alkalmazása a komponens alapú szoftverfejlesztésben. A kontraktusok használata ugyanis jelentősen csökkenti a szoftver komplexitását azzal, hogy a szolgáltatónak be kell tartania a szerződésekben (kontraktusokban) lefektetett megállapodásokat és ezeket a fogadó oldalon már nem kell ellenőrizni [54–58].

4.1. Programok helyességének vizsgálata viselkedéselemzéssel

Vizsgáljuk meg a viselkedéselemzés módszerével helyesség szempontjából Peterson példáját, amelyet a kölcsönös kizárásra adott két folyamat esetében [30].

A két egymással konkuráló folyamat struktúrája legyen a következő:

- **process P;**
- **begin**
- **repeat**
- **entry protocol**
- **critical section**
- **exit protocol**
- **non-critical section**
- **forever**
- **end;**

A folyamatokról a következő feltevésekkel élünk. A kritikus szakaszok (critical section) végrehajtása mindig véges időn belül befejeződik, a nem kritikus szakaszok (non-critical section) végrehajtása végtelen ideig tarthat, de akár mindegyik lehet üres is. A **repeat ... forever** utasítás szemantikája alapján a folyamatok ciklikusak és nem használhatnak párhuzamos nyelvi utasításokat kölcsönös kizárásra és szinkronizációs célokra. Feltesszük továbbá, hogy a megoldásban használt utasítások informális szemantikája az olvasó előtt jól ismert.

Peterson megoldása (1981):

Használjuk a **flag1**, **flag2** - mindegyik kezdeti értéke false és a **turn** - kezdeti értéke 1 vagy 2 lehet - osztott változókat, amelyeknek értékeit a folyamatok csak a belépési és kilépési protokolljainak időszakában változtathatják meg. Programozási nyelvünk tartalmazza a párhuzamos végrehajtás utasítását, de a párhuzamosság vezérlése szempontjából

fontos szinkronizációs és/vagy kölcsönös kizárást biztosító utasítások hiányoznak a nyelvből. A közös, osztott memóriával rendelkező párhuzamos számítógépekben a memóriarekeszek tartalma az úgynevezett „hardware lock” felügyelete mellett érhető el. Ez azt jelenti, hogy egy adott időpillanatban egy adott memóriarekeszt csak egy folyamat érhet el egyetlen olvasás vagy írás céljából. A szinkronizálásra megoldásában Peterson a „busy waiting” technikát alkalmazza, e technika segítségével oldja meg két folyamat kölcsönös kizárásának problémáját a következő módon [30]:

```

Mut_ex program
shared boolean variable flag1 := false; flag2 := false;
shared integer variable turn := 1 vagy 2;
process P1;
begin
  repeat
    flag1 := true; (* announce intent to enter *)
    turn := 2;      (* give priority to other process *)
    while flag2 and (turn = 2) do
      null;
    CriticalSection;
    flag1 := false; (* exit protocol*)
    NonCriticalSection
  forever
end;

•

process P2;
begin
  repeat
    flag2 := true; (* announce intent to enter *)
    turn := 1;      (* give priority to other process *)
    while flag1 and (turn = 1) do
      null;
    CriticalSection;
    flag2 := false; (* exit protocol*)
    NonCriticalSection
  forever
end;

parbegin P1 || P2 parend
end;

```

Viselkedéselemzéssel mutassuk ki, hogy a Mut_ex program biztonságos és eleven. Ehhez a következő feltevésekkel élünk. Az egymással ekvivalensnek tekinthető fizikai processzorok számáról az egyszerűség kedvéért tegyük fel, hogy legalább kettő és egymáshoz viszonyított se-

bességük tetszőleges. Továbbá a folyamatok kritikus szakaszai (Critical Section) végesek a nem kritikus szakaszok (NonCriticalSection) mérete viszont tetszőleges, lehetnek végtelenek, de üresek is egymástól függetlenül. Első lépésként konkretizáljuk, hogy mit értünk biztonságosságnak és elevenység alatt a **Mut_ex** program esetében. A **Mut_ex** program biztonságos, ha a két folyamat, P1 és P2, nem tartózkodhatnak egyszerre a kritikus szakaszaikban. A **Mut_ex** program eleven, ha a két folyamat közül bármelyik bármikor be akar lépni a kritikus szakaszába, akkor ezt véges időn belül meg tudja tenni.

1. Állítás. A **Mut_ex** program biztonságos.

Bizonyítás. Viselkedéselemzéssel vizsgáljuk meg a program egyes lehetséges végrehajtási útvonalait és mutassuk ki, hogy nem fordulhat elő olyan eset, amikor két folyamat egyszerre van a kritikus szakaszában.

- Tegyük fel, hogy a két folyamat egyszerre elindul. P1 beállítja a **flag1** értékét **true** értékre, a P2 pedig a **flag2** értékét szintén **true** értékre. E két tevékenység történhet teljesen azonos időben, abban az esetben, ha a két folyamat két különböző fizikai processzoron fut, mivel a **flag1** és **flag2** osztott változók két különböző memóriarekeszt reprezentálnak. Ezután mindkét folyamat rátér a következő utasítás végrehajtására, amely során a **turn** osztott változó értékét akarja módosítani mindkét folyamat. Ha egyszerre akarnak a **turn** által reprezentált memóriarekeszhez fordulni, akkor a hardware lock miatt ezt csak kölcsönös kizárásos alapon tehetik meg. Tegyük fel, hogy a P1 folyamat hajtja végre először az utasítást, azaz beállítja a **turn** változó értékét 2 értékre és ezzel átadja a kritikus szakaszba belépés jogát a P2 folyamatnak. Ezután P1 elkezd végrehajtani a **while** utasítást, ellenőrzi a ciklus feltételt, ami ebben az esetben **true** értékkel bír, ha P2 közben nem haladt előre. Ekkor P1 végrehajtja a ciklusának törzsét, a **null** utasítást és visszatér azonnal a ciklusfeltétel újabb ellenőrzésére. Ha közben a P2 folyamat végrehajtotta a **turn** változóra vonatkozó utasítását, akkor a **turn** változó értéke 1 lesz, ami azt jelenti, hogy a P2 folyamat is lemondott a kritikus szakaszba elsőként történő belépésének jogáról, ezáltal a P1 folyamat ciklus feltétele hamissá válik, aminek hatására befejezi a ciklus-utasításának végrehajtását és belép a kritikus szakaszába. A P1

folyamat a processzorok sebességétől függően a ciklusutasításának első végrehajtását már abban a fázisban hajthatja végre, amikor a **turn** osztott változó értékét P2 már átállította P1 számára kedvező értékre. Ekkor a P1 a ciklusfeltételt hamisnak találja és a ciklusból kilépve azonnal belép a kritikus szakaszába. Ebben az időpillanatban a helyzet a következő P1 belépett a kritikus szakaszába, P2 a belépési protokolljának ciklus utasítását hajtja végre. Kérdés: beléphet-e P2 is a kritikus szakaszába, miközben P1 már a saját kritikus szakaszában tartózkodik? A válasz: egyértelműen nem léphet be, amíg P1 ki nem lép a saját kritikus szakaszából, mivel P2 ciklusutasításának feltétele mindaddig igaz lesz, amíg P1 végre nem hajtja a kilépési protokollját. A program tehát ezen végrehajtási utak mentén biztonságosan működik.

- Megvizsgálva a többi lehetséges végrehajtási útvonalat hasonlóképpen beláthatjuk, hogy a programunk biztonságosan működik.
- Megjegyezzük, hogy a programunk akkor is biztonságosan működik, ha az egyik program el sem indul, de a másik szeretne a kritikus szakaszába belépni, majd dolga végeztével kilépni és a nem kritikus szakaszban történő véges idejű tartózkodás után ismét be akar lépni a kritikus szakaszába a **repeat ... forever** ciklusutasítás szemantikájának megfelelően. Ezen speciális eset átgondolását az olvasóra bízom.

2. Állítás. A **Mut_ex** program eleven.

Bizonyítás. Ezt a tételt is viselkedéselemzéssel igazoljuk. Azt kell bizonyítanunk, hogy ha bármelyik folyamat be akar lépni a kritikus szakaszába, akkor véges időn belül ezt megteheti. Állításunk igazolásához vizsgáljuk meg a lehetséges végrehajtási utak mentén a programunk viselkedését.

- Tegyük fel most, hogy a P2 folyamat elindult, de a P1 folyamat még nem. Kérdés most az, hogy a P2 folyamat beléphet-e a kritikus szakaszába? A válasz igen, mivel a **flag1** változó kezdeti értéke hamis, ami azt jelzi, hogy a P1 még nem kíván belépni a kritikus szakaszába. P2 végrehajtva a belépési protokollját udvariassan lemond ugyan a kritikus szakaszába elsőként történő belépés jogáról, de a **while** utasításának feltétele a **flag1** változó hamis értéke miatt hamis lesz és a ciklus törzsének végrehajtása nélkül azonnal kilép a ciklusból és belép a kritikus szakaszába.

- Az elevenség szempontjából érdekes lehet az az eset, amikor mondjuk P1 miután belépett a kritikus szakaszába és véges időn belül végrehajtotta a kilépési protokollját, a nem kritikus szakaszából is azonnal kilépve újra végrehajtja a belépési protokollját. Kérdés, hogy ebben az esetben nem monopolizálhatja-e az „erőforrást” a P1 folyamat akkor, ha a P2 folyamat be akar lépni a kritikus szakaszába? A válasz az, hogy P1 nem tudja megakadályozni, hogy a P2 folyamat be tudjon lépni a kritikus szakaszába. Tegyük fel ugyanis, hogy miközben a P1 folyamat a kritikus szakaszában tartózkodott, a P2 végrehajtotta a belépési protokollját. Ekkor a **flag2** értéke **true**, a **turn** változó értéke 1. Tegyük fel, hogy P2 lassan halad előre és a P1 által végrehajtott újabb belépési protokoll végrehajtásáig nem lépett be a kritikus szakaszába. Ekkor a rendszer osztott változói a következő értékekkel bírnak: a **flag1** és a **flag2** változók igaz értékkel, a **turn** változó 2 értékkel bír, mivel a P1 később hajtott végre a belépési protokollját, mint P2. Ez azt jelenti, hogy P2 léphet be a kritikus szakaszába és nem P1. Tehát ebben az esetben is biztosított a folyamatok előrehaladása.
- A többi esetben is belátható, hogy egyik folyamat sem akadályozhatja meg a másik véges időn belüli belépését a saját kritikus szakaszába. Ez még abban az esetben is igaz, ha egyik folyamat egy idő után végtelen ideig tartózkodik a kritikus szakaszában.

Belátható az is, hogy Petersonnak a „busy waiting” technikán alapuló fenti megoldása nem működik kettőnél több folyamat esetében.

Lamport háromnál több folyamat esetén kölcsönös kizárásra a „bakery” algoritmust javasolja [31], amelyben szintén csak szekvenciális utasításokat használhatunk kölcsönös kizárásra és szinkronizációs célokra.

A fenti egyszerű példa is mutatja, hogy a viselkedéselemzés is nagyon munkaigényes módszer, a gyakorlatban használható, de a megbízhatósága is megkérdőjelezhető.

A viselkedéselemzés módszerét formalizálta A. U. Shankar lineáris idejű temporális logika segítségével [29].

4.2. Helyes programok generálására vonatkozó módszerek

A *programszintézis* lényege, hogy megadjuk egy adott probléma specifikációját és ebből generáljuk a helyes programot egy előre megadott célnyelven. Szintézismódszereket dolgoztak ki klasszikus predikátumkalkulus formuláival megadott specifikációból kiindulva [1, 4, 13, 32] és különböző temporális logikai formulákat használva [22–24]. A temporális logikai formulákon alapuló programszintézisek általában a tabló módszer valamelyik változatát használják a célprogram modellgráfjának előállítására [6, 8, 23].

Példaként a kölcsönös kizárás problémáját megoldó program szintézisét mutatjuk be, amely kettőnél több folyamat esetén is használható. A tabló módszeren alapuló programszintézisek komoly problémája az állapotrobbanás. Ennek kiküszöbölésére számos módszert dolgoztak ki. Az állapotok számának növekedését kétféle módon akadályozhatjuk meg. Az egyik lényege, hogy újabb és újabb algoritmusokkal redukáljuk az állapotok számát [6], a másik lehetőség, hogy speciális feladatosztályokra dolgozunk ki szintézismódszereket és ezzel tudjuk csökkenteni a program állapotainak számát [23].

4.2.1. A párhuzamos kiszámítás modellje

Egy $P = P_1 \parallel \dots \parallel P_k$ konkurens program véges számú, párhuzamosan végrehajtott $P_1, \dots, P_k$ szekvenciális folyamatból áll. A folyamatok szinkronizációja általában szétválasztható az alkalmazásorientált kiszámítástól. Ezáltal lehetővé válik a konkurens programok szinkronizációs vázának önálló vizsgálata, származtatása.

Egy P_i folyamat szinkronizációs váza egy automata (state machine), amelyet egy irányított gráf reprezentál. A gráf csomópontjai egyedi neveket kapnak. Az s_i csomópont a P_i folyamat egy lokális állapotát jelöli. A gráf minden éle egy $B \rightarrow A$ szinkronizációs utasítással van címkézett, ahol B egy őrfeltétel (guard), A pedig egy tevékenységet jelöl. A hurokélek és az egyetlen kimenő él nélküli csomópontok (dead ends) kizártak.

A rendszer egy *globális állapotát* $(s_1, \dots, s_k, v_1, \dots, v_m)$ írja le, ahol s_i a P_i folyamat aktuális állapotát, a $v_1, \dots, v_m$ értékek pedig az $x_1, \dots, x_m$ közös (shared) változók aktuális értékeit jelölik. Egy B őrfeltétel egy predikátum az állapotok felett, egy A tevékenység

(action) egy párhuzamos értékadó utasítás az osztott, közös változókra vonatkozóan.

A *párhuzamosság* modellezhető az egyes folyamatok atomi tranzakcióinak nem-determinisztikus átlapolásával. Így a kiszámítás minden lépésében valamely folyamat egy „megengedett” átmenete kerül véletlenszerűen kiválasztásra és végrehajtásra.

A feladat specifikációját az MPCTL* formuláival adjuk meg.

4.2.2. A kölcsönös kizárás specifikálása

A sok hasonló folyamatból álló programra jó példa a $k \geq 3$ folyamatból álló rendszer, ahol egy P_i folyamat az N_i, T_i, C_i állapotok közül minden időpillanatban pontosan egy állapotban lehet. Az N_i állapot a folyamat nem kritikus szakaszának kódterületét, a T_i állapot a kritikus szakaszba való belépési próbálkozás kódterületét, a C_i állapot a folyamat kritikus szakaszának kódterületét jelöli. Egy folyamat a nem kritikus szakaszában nem használ közös, kritikus erőforrásokat, olyan számításokat végez, amelyek párhuzamosan végrehajthatók más folyamatok számításaival. A P_i folyamat a T_i állapotába kerül, ha egy kritikus erőforrást akar megszerezni további működéséhez. Ebből az állapotból belép a kritikus szakaszába a lehető legrövidebb időn belül, feltéve, hogy a szükséges erőforrások rendelkezésére állnak.

A rendszer specifikációja MPCTL* formulákkal:

- (1) A kezdeti állapotban minden folyamat a nem kritikus szakaszában van:

$$\bigwedge_i N_i$$

- (2) Egy folyamat a nem kritikus szakaszból mindig a próbálkozás szakaszába megy át és az ilyen átmenet mindig lehetséges:

$$\bigwedge_i AG(N_i \Rightarrow (AY_i T_i \wedge EX_i T_i))$$

- (3) Egy folyamat a próbálkozás szakaszából mindig a kritikus szakaszába megy át:

$$\bigwedge_i AG(T_i \Rightarrow (AY_i C_i))$$

- (4) Egy folyamat a kritikus szakaszból mindig a nem kritikus szakaszába megy át és az ilyen átmenet mindig lehetséges:

$$\bigwedge_i AG(C_i \Rightarrow (AY_i N_i \wedge EX_i N_i))$$

- (5) A P_i folyamat az N_i, T_i, C_i állapotok közül minden időpillanatban pontosan egy állapotban lehet:

$$\begin{aligned}\wedge_i AG(N_i &\equiv \neg(T_i \vee C_i)) \\ \wedge_i AG(T_i &\equiv \neg(T_i \vee C_i)) \\ \wedge_i AG(C_i &\equiv \neg(T_i \vee C_i))\end{aligned}$$

- (6) A P_i folyamat nem koplal:

$$\wedge_i AG(T_i \Rightarrow (AFC_i))$$

- (7) Az egyik folyamat egy átmenete nem eredményezi egy másik folyamat valamely átmenetének bekövetkeztét:

$$\begin{aligned}\wedge_{ij} AG((N_i \Rightarrow AY_j N_i) \wedge (N_j \Rightarrow AY_i N_j)) \\ \wedge_{ij} AG((T_i \Rightarrow AY_j T_i) \wedge (T_j \Rightarrow AY_i T_j)) \\ \wedge_{ij} AG((C_i \Rightarrow AY_j C_i) \wedge (C_j \Rightarrow AY_i C_j))\end{aligned}$$

- (8) Két folyamat nem lehet egyszerre a kritikus szakaszában:

$$\wedge_{ij} AG(\neg(C_i \wedge C_j))$$

- (9) A rendszer specifikációjának része az I összeköttetési reláció, ebben az esetben:

$$I = \text{dom}(I) \times \text{dom}(I) - (i, i) | i \in \text{dom}(I)$$

Megjegyzések:

- Ha $I = \text{dom}(I) \times \text{dom}(I) - (i, i) | i \in \text{dom}(I)$, azaz a folyamatok minden két különböző párja I -kapcsolatban van, akkor a fenti specifikáció a k folyamat kölcsönös kizárását írja le.
- Ha $I = \{\{1, 2\}, \{2, 3\}, \{3, 4\}, \{4, 5\}, \{5, 1\}\}$, akkor öt étkező filozófust szerveztünk egy gyűrűbe.
- Ha $I = \{\{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}\}$ és P_1, P_2 az olvasó folyamatokat, P_3, P_4 pedig az író folyamatokat jelenti, akkor az író-olvasó probléma egy specifikációját adtuk meg, ahol az íróknak nincs persze prioritásuk.

4.2.3. A szintézis lényege

Jelölések. A párhuzamos kiszámítás átlapolásos modelljét használjuk. Egy $P_i^j || P_j^i$ páros rendszer globális állapota ebben az esetben a következő módon definiálható:

$$(s_i, s_j, v_{ij}^1, \dots, v_{ij}^m),$$

ahol s_i, s_j az i -ik illetve a j -ik folyamat aktuális állapotait, $v_{ij}^1, \dots, v_{ij}^m$ pedig a közös, osztott változók értékeit jelölik.

Egy $(S_I^0 || P_{i1}^I || \dots || P_{ik}^I)$ I -rendszer k folyamatból áll, amelyeken egy, a folyamatok közötti kapcsolatokat kifejező I (*Interconnection*) reláció definiált. Az S_I^0 jelöli a folyamatok kezdőállapotainak halmazát.

Emerson és Clarke módszerének lényege.

- A probléma specifikációját megadjuk az MPCTL* nyelven, ebből a specifikációból meghatározzuk a két folyamatból álló rendszer specifikációját. Ezután alkalmazzuk például Emerson és Clarke szintézis módszerét a két folyamat szinkronizációs vázának (automaták) generálására [22]. Egy páros rendszernek $O(n^2)$ állapota van, ahol n jelöli az egyes szekvenciális programok méretét. Így ezt sokkal könnyebb generálni, mint az I -rendszert.
- A következő lépésben a páros rendszert I -rendszerré általánosítjuk [23].

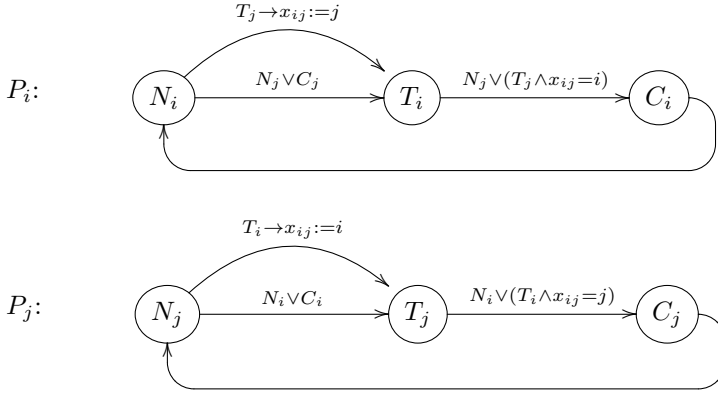
Az algoritmusok részletesen megtalálhatók [22, 23]-ban. Itt csak az algoritmusok eredményeként kapott állapotgépeket mutatjuk be.

Korábban megadtuk a kölcsönös kizárás specifikációját MPCTL* nyelven k folyamatból álló rendszerre. Vegyük a specifikáció „projekcióját” $k = 2$ esetre. Vezessük le ebben az esetben a megoldást Emerson és Clarke módszerével [22].

Az alábbi eredményt kapjuk a következő (i, j) párokra, amelyeket példányosítjuk a $(1, 2), (1, 3), (2, 3)$ párokra.

Két folyamat kölcsönös kizárását megadó automaták:

$$P_i || P_j$$

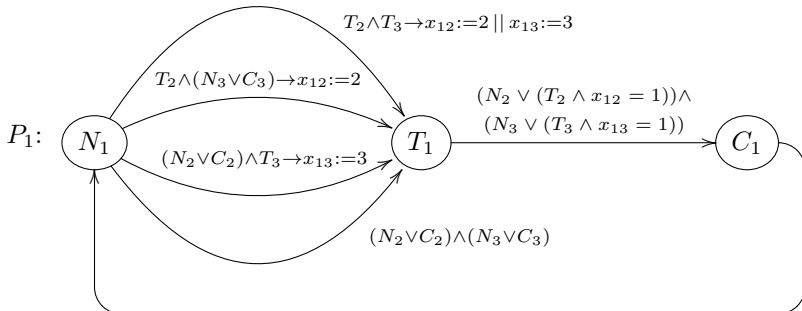


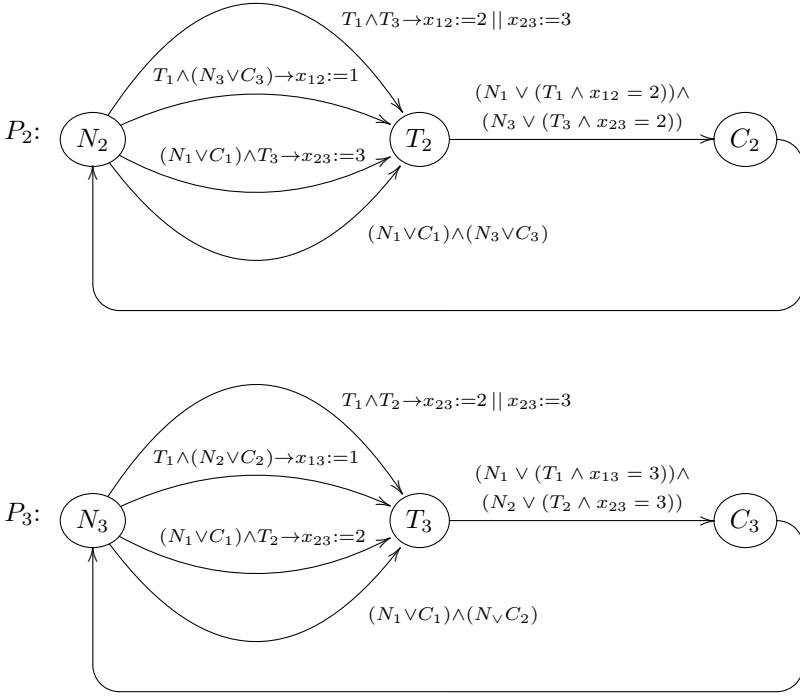
Ha összevetjük a kapott állapotgépeket Peterson megoldásával, azt találjuk, hogy két folyamatra a megoldások szemantikusan ekvivalensek. Így az I -rendszerre történő általánosítással Peterson megoldásának egy újabb általánosításához jutunk $n \geq 3$ számú folyamat esetére.

A több folyamatból álló I -rendszer átmeneteit az összes releváns páros rendszer átmeneteiből komponáljuk meg. Speciálisan a három folyamatból álló rendszer átmeneteit a felsorolt páros rendszerekből generáltuk a következő módon [23].

Három folyamat kölcsönös kizárását megadó automaták:

$$P_1 || P_2 || P_3$$





4.3. Modellellenőrzés

A modellellenőrzés egy teljesen automatikus módszer véges állapotú, konkurens rendszerek ellenőrzéséhez [5, 34]. Előnye, hogy a felhasználónak elegendő csupán a tervezett rendszer egy magas szintű modelljét, valamint az elvárt tulajdonságokat rögzítő specifikációt megadnia temporális logikai formulák formájában. A modell és a specifikáció ismeretében a módszer megvizsgálja az összes lehetséges állapotátmenetet, és igaz válasszal tér vissza, ha a rendszer modellje teljesíti a specifikációban megadott tulajdonságokat. Amennyiben hamis válasszal tér vissza, akkor megadja azt a végrehajtási szekvenciát, ami a tulajdonság megsértéséhez vezetett, ami nagy segítség a hiba javításához. A modellellenőrzés viszonylag gyors, még a közepes teljesítményű (akár asztali) számítógépeken is használható. Némely esetben nem véges állapotú rendszerek is ellenőrizhetők, ehhez különböző absztrakciós és indukciós feltevések szükségesek.

A specifikáció megadásához használt temporális logika biztosítja, hogy a konkurens rendszerek időbeli viselkedését vizsgálni tudjuk. Hasonlóan a tábló alapú szintézis módszerekhez, az állapottér robbanása okozza itt is a legnagyobb gondot. Az utóbbi időben komoly eredmények születtek ennek leküzdésére: a bináris döntési fák (BDD), a szimbolikus modellellenőrzés, a részleges sorrendiségen alapuló redukció, vagy a különösen ígéretes - a rendszerek bővítésével csak a megváltozott részek újraellenőrzését megkívánó - nyílt inkrementális modellellenőrzés (OIMC) [5, 34, 36, 37].

4.3.1. Példa

Peterson által megoldott feladat – kölcsönös kizárást megvalósító két folyamatból álló program – helyességének bizonyítása.

Viselkedéselemzéssel megvizsgálva a kölcsönös kizárást biztosító, két folyamatból álló, párhuzamos programot, azt találtuk, hogy mind a biztonságossági, mind az elevenségi tulajdonságoknak eleget tesz. Ezt most bebizonyítjuk az SMV modellellenőrző felhasználásával is [5]. A Symbolic Model Verifier (SMV) olyan eszköz, amely alkalmas annak eldöntésére, hogy egy véges állapotú rendszer kielégíti-e a CTL (Computational Tree Logic) vagy az LTL (Linear Temporal Logic) nyelvű specifikációját [5].

Egy véges állapotú rendszer leírására létrehozott nyelv elemei:

- Modulok
- Szinkron és átlapolt kompozíciók
- Nem determinisztikus átmenetek
 - Átmenet-relációk

Először megoldjuk a feladatot egyetlen modul segítségével, ekkor a kölcsönös kizárás SMV kódja a következő:

```
MODULE main
VAR
  state1: n1, t1, c1;
ASSIGN
  init(state1) := n1;
  next(state1) :=
  case
    (state1 = n1) & (state2 = t2): t1;
```

```

        (state1 = n1) & (state2 = n2): t1;
        (state1 = n1) & (state2 = c2): t1;
        (state1 = t1) & (state2 = n2): c1;
        (state1 = t1) & (state2 = t2) & (turn = 1): c1;
        (state1 = c1): n1;
        1: state1;
    esac;
VAR
    state2: n2, t2, c2;
    ASSIGN
        init(state2) := n2;
        next(state2) :=
            case
                (state2 = n2) & (state1 = t1): t2;
                (state2 = n2) & (state1 = n1): t2;
                (state2 = n2) & (state1 = c1): t2;
                (state2 = t2) & (state1 = n1): c2;
                (state2 = t2) & (state1 = t1) & (turn = 2): c2;
                (state2 = c2): n2;
                1: state2;
            esac;
VAR
    turn: 1, 2;
    ASSIGN
        init(turn) := 1;
        next(turn) :=
            case
                (state1 = n1) & (state2 = t2): 2;
                (state2 = n2) & (state1 = t1): 1;
                1: turn;
            esac;
SPEC AG (!((state1 = c1) & (state2 = c2))
SPEC AG ((state1 = t1) -> AF (state1 = c1))
SPEC AG ((state2 = t2) -> AF (state2 = c2))

```

Végrehajtva a modellellenőrzést az 1. ábrán látható eredményt kapjuk.

Kölcsönös kizárás SMV kódját modulokra bontással is megadhatjuk.

- MODULE main „két folyamat kölcsönös kizárását végző komponens"
- VAR
- s0: noncritical, trying, critical;
- s1: noncritical, trying, critical;
- turn: boolean;
- pr0: process prc(s0, s1, turn, 0);
- pr1: process prc(s1, s0, turn, 1);

```

SMV
Microsoft Windows XP [verziószám: 5.1.26001]
(C) Copyright 1985-2001 Microsoft Corp.

C:\WINDOWS>d:
D:\>cd smv2.5
D:\smv2.5>smv examples\mutex_mod.smv
-- specification AG (!<state1 = c1 & state2 = c2>) is true
-- specification AG (state1 = t1 -> AF state1 = c1) is true
-- specification AG (state2 = t2 -> AF state2 = c2) is true

resources used:
processor time: 0.016 s,
BDD nodes allocated: 251
Bytes allocated: 1045256
BDD nodes representing transition relation: 31 + 1
D:\smv2.5>_

```

1. ábra. A modellellenőrzés eredménye

- ASSIGN init(turn) := 0;
- FAIRNESS !(s0 = critical)
- FAIRNESS !(s1 = critical)

Globális specifikáció

```

SPEC AG !(s0 = critical & s1 = critical)
SPEC AG ((s0 = trying) -> AF ((s0 = critical))
SPEC AG ((s1 = trying) -> AF ((s1 = critical))
MODULE prc(state0, state1, turn, turn0)
ASSIGN
    init(state0) := noncritical;
    next(state0) :=
    case
        (state0 = noncritical) : trying, noncritical;
        (state0 = trying) & (state1= noncritical) : critical;
        (state0 = trying) &
            (state1= trying) & (turn = turn0) : critical;
        (state0 = critical) : critical, noncritical;
    1: state0;
    esac;
    next(turn) :=
    case
        turn = turn0 & state0 = critical : !turn;
    1: turn;
    esac;
FAIRNESS running

```

Végrehajtva a **main** és a **prc** modulokra az SMV-t eredményül azt kapjuk, hogy a kölcsönös kizárás teljesül:

```
SPEC AG !(s0 = critical & s1 = critical) is true
```

és a rendszer garantáltan koplalás mentes:

```
SPEC AG ((s0 = trying) -> AF ((s0 = critical))) is true
```

```
SPEC AG ((s1 = trying) -> AF ((s1 = critical))) is true
```

A megoldás elemzése:

Ha azt is ellenőrizni akarjuk, hogy az egyes folyamatok nem szigorúan alternálva kell, hogy belépjenek a kritikus szakaszukba, akkor kiegészíthetjük a komponens globális specifikációját a következőkkel.

Nincs szigorú alternáló belépési kényszer

```
SPEC AG (( s0 = critical) -> A[(s0 = critical)
    U (!(s0 = critical) &
      !E[!(s1 = critical) U (s0 = critical)])])
SPEC AG (( s1 = critical) -> A[(s1 = critical)
    U (!(s1 = critical) &
      !E[!(s0 = critical) U (s1 = critical)])])
```

Lefuttatva az SMV-t azt kapjuk, hogy a két utóbbi formula értéke hamis. Ez azt jelenti, hogy egy folyamat elhagyva a kritikus szakaszát, újra beléphet abba anélkül, hogy meg kellene várnia a másik folyamatot, hogy belépjen a kritikus szakaszába, majd elhagyja azt.

4.4. Tesztelés

Programok hibamentességének kimutatására leggyakrabban használt módszer a tesztelés. A tesztelés támogatására az idők folyamán számos módszert dolgoztak ki, amelyek kiterjednek a tesztadatok generálásától az eszköztámogatáson keresztül a különböző módszerek kidolgozásáig. A tesztelés részletes tárgyalása meghaladja jelen tanulmány kereteit. Azt azonban megjegyezzük, hogy teszteléssel mindig csak azt tudjuk kimutatni, hogy a tesztadatokra nézve a program helyes-e. A tesztadatok számossága végtelen is lehet, így a teljes hibátlanság kimutatása teszteléssel lehetetlen. A modellvezérelt programozási paradigma megjelenésével a tesztelés módszerei is fejlődnek, változnak. Megjelent a modell alapú tesztelés fogalma és egyre jelentősebb a tesztelés modellezésének szerepe [58].

5. Záró gondolatok

A programok helyességének kérdése a számítógépek megjelenésével gyakorlatilag azonnal felvetődött. Az idők folyamán a programok helyességének, megbízhatóságának vizsgálata több irányban fejlődött. Tanulmányunkban az imperatív (procedurális) programozási nyelveken alapuló eszközökbe, módszerekbe igyekeztünk betekintést nyújtani. A helyes programok automatikus előállításától még messze vagyunk, de napjainkban már ígéretes eredmények vannak a gyakorlatban is hatékonyan alkalmazható módszerek tekintetében például a kontraktusok alkalmazása [54–57], a tesztelés [58] és a modellellenőrzők használata a témakörében [34, 36, 37]. Az Eötvös Loránd Tudományegyetem Informatikai Karán az informatika oktatásában nagy hangsúlyt helyezünk a létrehozandó rendszerek helyességének igazolására informális, vagy formalizált módszerek felhasználásával.

Hivatkozások

- [1] G. R. Andrews, A Method for Solving Synchronization Problems, *Science of Computer Programming*, 13 (1989/90), pp. 1–21.
- [2] Z. Chaochen, Specifying Communicating Systems with Temporal Logic, *LNCS* 389 (1987), pp. 304–323.
- [3] Z. Horváth, The Weakest Precondition and the Specification of Parallel Programs, *Proc. of the Third Symposium on Programming Languages and Software tools*, Kaariku, Estonia (1993), pp. 24–34.
- [4] A. van Lamsweerde, M. Sintzoff, Formal Derivation of Strongly Correct Concurrent Programs, *Acta Informatica*, 12(1) (1979), pp. 1–31.
- [5] K. L. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem*, Kluwer Academic, 1993.
- [6] Z. Manna, P. Wolper, Synthesis of Communicating Processes from Temporal Logic Specifications, *ACM TOPLAS*, 6 (1984), pp. 68–93.
- [7] É. Rácz, Specifying a Transaction Manager Using Temporal Logic, *Proc. of the Third Symposium on Programming Languages and Software Tools*, Kaariku, Estonia (1993), pp. 109–119.

-
- [8] P. Wolper, The Tableau Method for Temporal Logic: an Overview, *Logique et Anal.* 28 (1985), pp. 119–136.
 - [9] G. Gopalakrishnan, R. Fujimoto, Design and Verification of the Rollback, Chip Using HOP: A Case Study of Formal Methods Applied to Hardware Design, *ACM Trans. on Comp. Syst.*, 11(2) (1993), pp. 109–145.
 - [10] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliff, 1976.
 - [11] S. Owicki, D. Gries, An Axiomatic Proof Technique for Parallel Programs, *Acta Informatica*, 6 (1976), pp. 319–340.
 - [12] C. A. R. Hoare, Communicating Sequential Processes, *Comm. ACM*, 21 (1978), pp. 666–677.
 - [13] L. Kozma, A Transformation of Strongly Correct Concurrent Programs, *Proc. of the Third Hungarian Computer Science Conference* (1981), pp. 157–170.
 - [14] F. Kröger, *Temporal Logic of Programs*, Springer-Verlag, 1987.
 - [15] R. M. Smullyan, *First Order Logic*, Springer-Verlag, 1971.
 - [16] Pásztorné Varga Katalin, *Logikai alapozás alkalmazásokhoz (matematikai logika - számítástudomány)*, egyetemi jegyzet, ELTE, 1992.
 - [17] K. M. Chandy, J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1989.
 - [18] J. Misra, Specifying Concurrent Objects as Communicating Processes, *Science of Computer Programming*, 14 (1990), pp. 159–184.
 - [19] L. Kozma, Proving the Correctness of Implementations of Shared Data Abstractions, *LNCS* 137 (1982), pp. 227–241.
 - [20] L. Varga, Implementation of Abstract Data Types with Correctness Proof, *Annales Univ. Sci. Budapest, Sect Comp.* 8 (1987), pp. 109–118.

-
- [21] J. V. Guttag, E. Horowitz, D. R. Musser, Abstract Data Types and Software Validation, *Comm. ACM*, 21(12) (1978), pp. 1048–1064.
 - [22] E. A. Emerson, E. M. Clarke Using branching time temporal logic to synthesize synchronization skeletons, *Science of Computer Programming*, 2, pp. 241–266.
 - [23] P. C. Attie, E. A. Emerson Synthesis of Concurrent Systems with Many Similar Processes, *ACM TOPLAS*, 20(1) (1998), pp. 51–115.
 - [24] P. C. Attie, E. A. Emerson Synthesis of Concurrent Programs for an Atomic Read/Write Model of Computation *ACM TOPLAS*, 23(2) (2001), pp. 187–242.
 - [25] R. W. Floyd, Assigning meaning to programs, *Proc. Sym. in Applied Math.*, 19, Mathematical Aspects of Computer Science (1967), pp. 19–32.
 - [26] C. A. R. Hoare, An axiomatic basis of computer programming, *Comm. ACM*, 12 (1969), pp. 576–583.
 - [27] R. M. Burstall, Program proving as hand simulation with a little induction, *Information Processing* (1974), pp. 308–312.
 - [28] P. Wegner, Classification in Object-Oriented Systems, *SIGPLAN Notices*, 21(10) (1986), pp. 173–186.
 - [29] A. Udaya Shankar, An Introduction to Assertion Reasoning for Concurrent Systems, *ACM Computing Surveys*, 25(3) (1993), pp. 225–262.
 - [30] G. L. Peterson, Myths about the mutual exclusion problem, *Inf. Process. Lett.*, 12(3) (1981), pp. 1133–1145.
 - [31] L. Lamport, A fast mutual exclusion algorithm, *ACM Trans. Comput. Syst.* 5(1) (1987), pp. 1–11.
 - [32] N. Wirth, Program development by stepwise refinement, *Comm. of the ACM*, 14(4) (1971).
 - [33] Z. Manna, Properties of programs and the first order predicate calculus, *Journal of ACM*, 16 (1969).

-
- [34] E. Clarke, O. Grumberg, D. Peled, *Model Checking*, MIT Press, 2000.
 - [35] E. A. Emerson, Temporal and modal logic, *Handbook of Theoretical Computer Science*, Vol. B, Formal Models and Semantics. MIT Press, 1990.
 - [36] D. Dams, R. Gerth, O. Grumberg, Generation of reduced models for checking fragments of CTL, *Proceedings of the 5th Conference on Computer-Aided Verification, LNCS 697* (1993), pp. 479–490.
 - [37] S. Krishnamurthi, K. Fisler, Foundations of incremental aspect model-checking, *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(2) (2007), pp. 1–36.
 - [38] Kozics S., *Az Algol 60, a Fortran, a Cobol és a PL/1 programozási nyelvek*, ELTE, 1992.
 - [39] H. R. Nielson, F. Nielson, *Semantics with Applications, A Formal Introduction*, John Wiley & Sons, 1992.
 - [40] J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, 1977.
 - [41] N. Wirth, *Systematic programming, An Introduction*, Prentice-Hall, 1973.
 - [42] N. Chomsky, *Aspects of the Theory of Syntax*, MIT Press, 1965.
 - [43] *Revised Report on the Algorithmic Language Algol 68*, Ed. by A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, R. G. Fisher, Springer-Verlag, 1973.
 - [44] Lőcs Gy., *Algol 60 programozási nyelv*, Műszaki Könyvkiadó, Budapest, 1973.
 - [45] Lőcs Gy., Vigassy J., *A Fortran programozási nyelv*, Műszaki Könyvkiadó, Budapest, 1973.
 - [46] R. F. Paige, D. S. Kolosov, F. A. C. Polack, An Action Semantics for MOF 2.0, SAC’06, *Proc. of the 2006 ACM Symposium on Applied Computing* (2006), pp. 1304–1305.

-
- [47] B. K. Aichering, H. Brandl, E. Jöbstl, W. Krenn, UML in Action: A Two-Layered Interpretation for testing, *ACM SIGSOFT Software Engineering Notes*, 36(1) (2011), pp. 1–8.
 - [48] D. Arora, B. Hazela, V. Saxena, Semantics for UML Model Transformation and Generation of Regular Grammar, *ACM SIGSOFT Software Engineering Notes*, 37(3) (2012).
 - [49] P. D. Mosses, *A Tutorial on Action Semantics*, FME'94.
 - [50] P. D. Mosses, Component-Based Semantics, SAVCBS'09, *Proc. of the 8th International Workshop on Specification and Verification of Component-Based Systems*, (2009).
 - [51] G. Stuurman, I. Kurtev, Action Semantics for Defining Dynamic Semantics of Modelling Languages, *BM-FA'11: Proc. of the Third Workshop on Behavioural Modelling*, (2011), pp. 64–71.
 - [52] F. Parisi-Precise, A. Pierantonio, An Algebraic Theory of Class Specification, *ACM Trans, On Soft., Eng., and Meth.*, 3(2) (1994), pp. 166–199.
 - [53] P. Wegner, The Vienna Definition Language, *ACM Comp. Surveys*, 4(1) (1972), pp. 5–63.
 - [54] B. Meyer, Applying 'design by contract', *Computer*, 25(10) (1992), pp. 40–51.
 - [55] CADP <http://www.inrialpes.fr/vasy/cadp/>, 2011.
 - [56] Contracts for Java (cofoja). <http://code.google.com/p/cofoja/>, 2011.
 - [57] L. Kozma, Gy. Orbán, Defining Contracts with Different Tools in Software Development, *Annales Univ. Sci. Budapest., Sect. Comp.*, 36 (2012).
 - [58] H. G. Gross, *Component-Based Software Testing with UML*, Springer-Verlag, 2005.
 - [59] M. S. Laventhal, Synthesis of Synchronization Code for Data Abstraction, *MIT/LCS/TR-203* (1978).

- [60] B. Liskov, A. Snyder, R. Atkinson, C. Schaffert, Abstraction Mechanisms in CLU, *Comm. ACM*, 20(8), pp. 564–576.
- [61] S. Owicki, Axiomatic Proof Technique for Parallel Programs, *TR75-251*, *Cornell University*, 1975.
- [62] Csörnyei Z., *Fordítási algoritmusok*, Erdélyi Tankönyvtanács, Kolozsvár, 2000.
- [63] C. Szyperski, *Component Software Beyond Object-Oriented Programming*, (Second Edition), Addison-Wesley, 2002.
- [64] Á. Dávid, T. Pozsgai, L. Kozma, Extended systems with verified components, *Periodica Polytechnica Electrical Engineering*, 51(3-4) (2007), pp. 133–139.

A programkód és a konfigurációs kód közötti határ

Köllő Hanna*

Eötvös József Collegium**

hanna.kollo@gmail.com

Az informatika világa nem csak akadémiai körökben, hanem az iparban is lehetőséget ad szakmai kiteljesedésre. Ebben a dolgozatban egy példát mutatok erre, ipari környezetből, egy személyesen kigondolt és megvalósított ötletet, amely új szemléletével megoldást adott egy hosszú távú szoftverfejlesztési problémánkra. A feladat: komplex, heterogén, folyamatosan változó, egy ember által át nem látható bonyolultságú algoritmusokat írni és karbantartani egy specifikus üzleti környezetben. Kidolgoztam egy paradigmát amivel biztonságosan és gyorsan építhetünk ilyen algoritmusokat már meglévő kisebb algoritmusokból. Az algoritmust részekre bontjuk, amelyek egymás között áramoltatják az adatokat. Így az egész egy irányított gráfként fogható fel. A részek külön-külön implementálhatók, és egy későbbi fázisban összekombinálhatók (... ezt egy programozásban járatlan szakember is elvégezheti). Az ötlet a *flow based programming* elméleten alapszik [2, 3].

1. Bevezetés

Szoftverfejlesztőként dolgozom egy tőzsdei kereskedéssel foglalkozó cégnél. Napjainkban a kereskedés szinte teljes mértékben a kibertérben zajlik, a kereskedők nem találkoznak egymással szemtől-szemben a parkettán, hanem csak egy számítógépes alkalmazás segítségével, hálózaton

*Optiver Holding BV, Amszterdam, Hollandia

**2004–2010

küldik el megbízásaikat egy központi szervernek ... valahová, egy távoli adatközpontba, amely az eladási és a vételi megbízások igazságos összepárosítását valós időben elvégzi. A tőzsde impozáns épülete már csak szimbolikus szerepet tölt be. A kereskedés digitalizálása új lehetőséget nyitott a gazdaságban, és pedig azt, hogy a kereskedési logikát is többé-kevésbé automatizáljuk, azaz egy számítógép hozza meg a döntést, hogy eladjuk-e vagy megvegyük a részvényeket, és ezt pontosan mikor tesszük. Ebben a körben, ahol a döntéshozás sebessége kulcsfontosságú, a számítógéppel támogatott vagy extrém esetben teljesen automatizált döntéshozás hihetetlen piaci előnyöket nyújthat. Nem csoda, hogy az iparág igen jelentős mértékben támaszkodik az informatikára. Szoftverfejlesztőként az a feladatunk, hogy a pénzügyi világban jártas kereskedők mindennapi tevékenységét a lehető leghatékányabban automatizáljuk, olyan eszközöket adjunk a kezükbe, amelyekkel gyorsabban, biztonságosabban és sokkal nagyobb tételben tudják a pénzügyi termékeket adni-venni.

Két fontos kihívással állunk szemben: az egyik a program sebessége (a tőzsde működése szerint aki először adja ki a megbízást, azé lesz az ügylet, és a konkurencia nagy, emiatt a programjaink sebességét a fizika elméleti határáig optimalizálni kell), *a másik a flexibilitás* (a piaci körülmények folyamatosan változnak, tehát a stratégiáink hamar elavulttá válnak, ha nem adaptáljuk őket).

2. Általános problémák

A programunk amely a kereskedési logikát implementálja, többé-kevésbé monolitikus rendszer. Maga az algoritmus izolálva van a rendszer infrastrukturális elemeitől, azonban a program tervezésekor senki nem gondolt arra, hogy a kereskedési algoritmus, amit folyamatosan továbbfejlesztünk, egyszer majd olyan bonyolulttá és emiatt kezelhetetlenné válik, hogy a piachoz való alkalmazkodási képességünk, tehát a flexibilitásunk fő visszatartó ereje lesz.

A programot száznál is több kereskedő használja, különböző tőzsdéken és különböző pénzügyi termékek kereskedésére. Több fő logikai egységet (ú.n. stratégiát) tartalmaz, ezek általában egy-egy specifikus termék kereskedésére lettek kifejlesztve, azonban a stratégiáknak vannak közös elemei is. Összességében néhány száz paraméter kontrollálja a teljes logikát, és a felhasználónak ezek jelentős részét pontosan ismernie kell.

Egy hasonlattal élve, a program olyan, mint egy versenyautó, aminek segítségével a paraméterek folyamatos módosítása révén a felhasználó végighalad a versenypályán azzal a reménnyel, hogy elsőként ér célba.

Amikor egy új felhasználó megpróbálja megtanulni a rendszer használatát, semmi mást nem lát, mint egy sereg paramétert egy-egy rövid leírással. A betanítás során kialakul benne egy kép, hogy milyen logikai egységekből áll össze a logika, és melyik paraméterek melyik részekhez tartoznak, de az korántsem biztos, hogy maga a programkód is ilyen struktúra mentén szerveződik. Másrészt a programot folyamatosan fejlesztjük, így a paraméterek funkciója, ha csak kicsit is, de változik, továbbá minden új verzióban új paramétereket vezetünk be a létezők mellé, és a felhasználónak tudnia kell, hogy az új paraméterek lényegesek-e vagy nem az általa használt konfigurációban, amit végső soron a meglévő paraméterek értéke határoz meg.

A programozónak sincs könnyű dolga. Amikor a felhasználó egy új funkcionalitást kér, egyáltalán nem triviális, hogy pontosan melyik logikai egységek megváltoztatásával lehet azt megvalósítani. Mindig több járható út van, és minden választott megoldás jár kellemetlen mellékhatásokkal is (azaz az új funkció bevezetésekor a már létező funciók is változnak). Ráadásul a modularizáció, amely a program megtervezésekor kristálytiszta volt, az évek során a sok változtatás miatt elmosódott, és így a logikai egységek inkább csak koncepció szintjén léteznek, mint programkód szintjén. Mondhatnánk persze azt, hogy készítsünk új, különálló programot minden funkció leimplementálására, de ezzel elveszítenénk azt az előnyt, hogy egy logikai (rész)egységet csak egyszer implementálunk, és emiatt sokkal nagyobb programozói kapacításra lenne szükség.

Összefoglalva tehát a túl nagyra nőtt monolitikus rendszerünk problémái:

- fekete dobozként viselkedik,
- nehéz megérteni,
- nehéz megváltoztatni,
- nehéz használni.

Mindez azt eredményezte, hogy az olyan fontosnak kinevezett flexibilitás, azaz a piac változásaihoz való alkalmazás képessége vérszenen

lecsökkent. Viszonylag kis változtatások is sok programozói munkát igényeltek, és a rendszer váratlan helyeken okozott hibákat egy új funkcionalitás bevezetése után. Próbálkoztunk orvosolni a problémát újrastrukturálással, szétdarabolással, a komplexitás csökkentésével, és bár ezek a lépések hoztak eredményt, egyik megoldás sem kecsegtetett hosszútávú nyugalommal.

3. Egy új megközelítés: *Hive*

Amikor egy rendszer túl bonyolulttá válik, érdemes megvizsgálni, hogy a megfelelő absztrakciós szinten kezeljük-e a problémánkat.

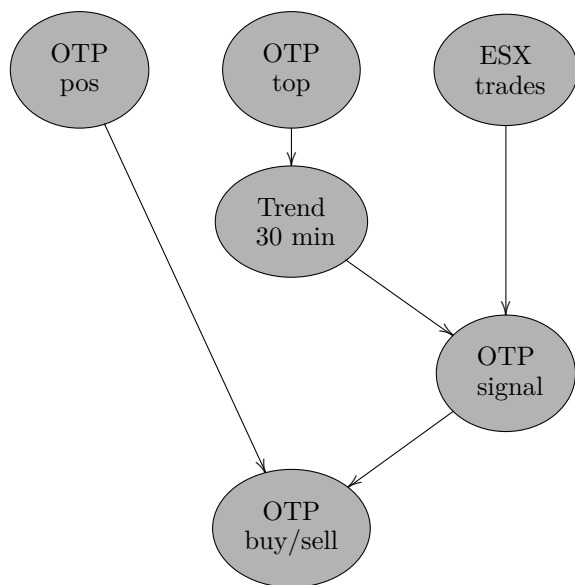
Az ötlet a következő: nyilvánvaló, hogy a program újratervezésekor a jobb modularizálásra kell tenni a hangsúlyt, de lehetne-e ezeket a modulokat a felhasználó számára is láthatóvá, sőt, konfigurálhatóvá tenni? Bontsuk szét az algoritmust logikai egységekre, amelyek egymáshoz lazán csatolódnak ... de lépünk eggyel tovább: *az egymáshoz csatolás fázisát halasszuk el későbbre, akár a programozási munkán túlra.* Programozzuk le az algoritmus logikai egységeit, de hagyjuk a felhasználóra, hogy melyik egységeket akarja használni, és ezeket milyen sorrendben. Sőt, ne korlátozzuk a lehetőségeket egy lineáris sorrendre, hanem engedjünk meg tetszőleges irányított körmentes gráfot a komponensek összekombinálásában.

Ezzel az ötlettel álltam elő, és paradigmának a *Hive* (*Méhkas*) nevet adtam.

3.1. Egy példán keresztül ...

Egy kereskedési stratégia – nagyon leegyszerűsítve – a következőképpen működik. Figyelünk egy vagy több adatfolyamot a számunkra érdekes részvények árfolyamának alakulásáról. Egy modell alapján előrejelezük, hogy egy kiválasztott részvény árfolyama milyen irányba fog elmozdulni a következő néhány másodpercben. Abban a pillanatban, amint ez az előrejelzés szignifikánssá válik, kiadjuk a megbízást egy bizonyos mennyiség eladására vagy vételére. A megbízás hatása azonnal tükröződni fog a bejövő adatfolyamokban, ezek hatását kiszűrjük. A saját ügyleteink által generált pozíciónk kockázatát határok között tartjuk (ellentétes hatású megbízások kiadásával).

A *Hive* rendszerben a kereskedő egy grafikus felületen összeválogathatja a komponenseket, amelyek ezt a logikát megvalósítják. Bizonyára szüksége lesz olyan komponensekre, amelyek az adatfolyamokra feliratkoznak és ezeket a *Hive*-ba áramoltatják – minden adatfolyamhoz egy komponens. Szüksége lehet olyan komponensekre, amelyek az adatfolyamokat normalizálják, trendet elemeznek vagy szokatlan mintázatokat keresnek. Biztosan lesz egy olyan komponens, amely a sok bejövő adatfolyamot aggregálja egyetlen jellé. Jó ha tudjuk a saját pozícióinkat, így egy komponens felelős lehet azért, hogy ezt egy külső adatforrásból behozza, és folyamatosan naprakészen tartsa. Végül pedig a legfontosabb komponens az, amely az aggregált jelből és a pozícióból eldönti, érdemes-e megbízást kiadni, vagy nem. Az egész rendszer egy irányított gráfként fogható fel amelyben a csúcsok a komponensek, az élek pedig az áramló adatok (1. ábra).



1. ábra. Egy egyszerű példa: Hat komponensből álló *Hive*-stratégia

A komponensek mögött viszonylag rövid, pár száz soros programkód van, és minden komponens csak egyetlen, a kereskedők által közismert funkcionalitást implementál. Például: árfolyamot átlagoló, szűrő, simító,

két adatfolyamot összefésülő komponensek. A komponenshez tartozhatnak paraméterek, és a felhasználó számára egyértelmű, hogy melyik paraméter melyik komponenshez tartozik. Ha a felhasználó megváltoztat egy paramétert egy komponensben, a teljes algoritmus irányított gráfjára rápillantva gyorsan felmérheti, hogy melyik más komponensek befolyásolódnak a változtatástól, nevezetesen az összes olyan komponens, amely egy irányított úttal elérhető a megváltoztatott komponensből.

Minden komponens a következő egyszerű interfész alapján működik: üzeneteket lehet elhelyezni a bejövő üzenetsorába, új üzeneteket képes létrehozni, valamint van egy „Futtat()” eljárása, aminek a meghívásakor elvégezheti a teendőit. Azt, hogy mikor melyik komponens kapja meg a vezérlést, illetve hogy melyik üzenetek melyik üzenetsorokba kerülnek be (mert egy üzenetet több komponens is megkaphat), egy központi vezérlő modul dönti el. A komponens, amikor létrehoz egy új üzenetet, nem kontrollálja, hogy azt ki kapja meg, sőt, egyáltalán megkapja-e valaki. Az üzenetküldés aszinkron, vagyis a komponens nem várja meg, hogy az általa létrehozott üzenet feldolgozása befejeződjék, mielőtt folytatná a tevékenységét. Úgy tapasztaltuk, hogy *az aszinkron kommunikáció egy helyes tervezési döntés volt*, szemben a szinkron kommunikációval, amikor több mindenre kell figyelni az implementáláskor, és emiatt az nehézkes lehet.

Röviden összefoglalva: ennyiből áll a *Hive* paradigma, amely az algoritmikus kereskedés üzleti problémájára alkalmazva ismereteim szerint az első megvalósítás. (Más problémákra alkalmazva a kulcsötlet megtalálható, mint például a matematikai modellezések [1] vagy adattárházak lekérdező nyelve [4] területén.)

3.2. Előnyök és hátrányok

Melyek az előnyei ennek a paradigmának?

Az egyik legnyilvánvalóbb előny, hogy a kereskedési logika nem fekete doboz többé, a felhasznált komponensek és a köztük levő függőségek nyilvánvalóak, tehát a paradigma öndokumentáló jellegű.

További nyereség, hogy a konfigurációt a felhasználó végzi, nem pedig a programozó, emiatt a programozóra kevesebb munka hárul, igaz, a felhasználó felelőssége megnő. Úgy tapasztaltuk hogy a felhasználók örömmel használják ki új „hatalmukat”, és megbirkóznak az extra felelősséggel.

A rendszer sokkal könnyebben unit-tesztelhető, hiszen egy komponens vagy néhány összefüggő komponens kiragadható a rendszerből tesztelés céljából.

A legnagyobb előny viszont az, hogy a rendszer leíró ereje ugrásszerűen megnő. Prototípusaink során azt tapasztaltuk, hogy 10–15 rövid komponens implementálásával és különböző módon való összekombinálásával a cég által használt kereskedési logika 80%-a leírható. Ezzel kb. tízszeres kódbasis-csökkenést értünk el.

A másik nagy nyereség a visszakapott flexibilitás: amikor áttérünk egy *Hive*-jellegű paradigmára, először sok komponenst kell implementálni, de egy felfutási idő után már egyre kevesebbet, mert az új funkcionálisok legtöbbször kifejezhetők a már létező komponensek másként való összekombinálásával.

Ha egy gondolatként akarjunk megfogalmazni a paradigma előnyeit, akkor azt mondhatjuk, hogy a rendszer ereje elsősorban nem a komponensek erejéből fakad, hanem komponensek és a köztük levő kapcsolatok alkotta gráfból.

Persze vannak hátrányai is a paradigmának.

Az első és nyilvánvaló hátrány, hogy a programozó nem tudja többé olyan szinten garantálni a logika helyességét, mint eddig, hiszen csak a komponensek helyességét tudja garantálni. Emiatt új üzleti folyamatokra van szükség, hiszen a konfigurálás után is be kell iktatni egy tesztelési fázist.

A másik előrelátható probléma a kombinatorikus robbanás, azaz hogy a komponensek összekombinálásának lehetőségei végtelenek, és minden kombinációt, amit élesben használunk, minőségileg garantálni kell. Ehhez kapcsolódik az az előrelátható veszély is, hogy amíg eddig túl hosszú ideig kellett várni egy új funkcionalitás megvalósítására, mostantól (ha nincs szükség programozásra, csak konfigurálásra) épp hogy túl rövid lesz ez az idő, és akár átgondolatlan, elhamarkodott algoritmusok is éles használatba kerülhetnek.

Mindezen hátrányok ellenére a *Hive* paradigma felkeltette a kereskedők érdeklődését, és nagyon pozitív fogadtatásban részesült.

4. Értékelés

A *Hive* paradigma nagyon pozitív fogadtatásban részesült mind a leendő felhasználók, mind a fejlesztőtársak, mind a vezetőség részéről.

A kereskedők szinte percekben belül új üzleti ötletekkel álltak elő, amiket a meglévő rendszerbe nehéz lenne beépíteni, de a *Hive*-ban ennek nem látták akadályát. A vezetőség részéről több hét extra fejlesztési időt kaptam a prototípus részletes kidolgozására és sebesség-optimalizálásra. Fejlesztőtársaim részéről is támogatást és további ötleteket kaptam, többek között egy grafikus *Hive editor*, egy teszt-rendszer, és egy domain specifikus leíró nyelv kidolgozására (ezek közül a grafikus editor meg is valósult.)

A prototípus sikeressége után a teljes körű integrálás jóval rögzesebb útja következett.

5. Következtetés

Az ilyen tapasztalatok, mint amilyen a *Hive* kigondolása és implementálása volt – újra és újra meggyőznek a szoftverfejlesztői szakma szépségeiről. A legfontosabb tanulság annak az elméletben már ismert ténynek a gyakorlati megtapasztalása volt, hogy *a megfelelő absztrakciós szinten való gondolkodás mennyire megkönnyíti egy probléma szoftveres kezelését*. Minden probléma és minden megoldás kis elemekből épül fel. Amikor olyan eszközöket alkalmazunk a megoldás megvalósítására, ahol *a megoldásban használt konstrukciók minták hasonlítanak a problémánk konstrukciós mintáihoz*, a szoftverfejlesztés könnyű és jutalmazó tevékenység.

Hivatkozások

- [1] MathWorks: *Simulink*,
<http://www.mathworks.nl/products/simulink/index.html>,
2014. január 12.
- [2] Wikipedia: *Flow-based programming*,
http://en.wikipedia.org/wiki/Flow-based_programming,
2014. január 12.
- [3] Dr Dobbs: J. Falgout, *Dataflow Programming: Handling Huge Data Loads Without Adding Complexity*,
<http://www.drdobbs.com/database/dataflow-programming-handling-huge-data/231400148>, 2014. január 12.

-
- [4] OneMarketData: *OneTick - a times database and complex event processing (CEP)*, <http://www.onetick.com/web1>, 2014. január 12.
 - [5] J. P. Morrison, *Flow-Based Programming*, 2nd edition, CreateSpace Independent Publishing Platform, 2010.
 - [6] W. W. Wadge, *Lucid, the Dataflow Programming Language (Apic Studies in Data Processing)*, Academic Press, 1985.
 - [7] R. Jennings, *LabVIEW Graphical Programming*, McGraw-Hill, Bk&Disk edition, 1994.

Racionális függvényrendszerek és alkalmazásuk a jelfeldolgozás területén

Lócsi Levente*

Eötvös József Collegium**

locsi@inf.elte.hu

Témavezető: Schipp Ferenc, ELTE IK

1. Bevezetés

Elsőéves programtervező matematikus hallgatóként jómagam is ott voltam az Informatikai Műhely 2004. februári alapító gyűlésén, majd sokáig aktív tagja voltam ennek a műhelynek, aminek rengeteg kellemes élményt és szakmai segítséget köszönhetek. Így örömmel fogadtam Csörnyei Zoltán tanár úr, műhelyvezetőnk felkérését, hogy írjak egy cikket a műhely tízéves jubileumi kiadványába.

Egyúttal ez egy remek lehetőség arra is, hogy 2008-tól doktoranduszként, majd 2011-től az ELTE Informatikai Kar Numerikus Analízis Tanszékének tanársegédjeként végzett kutatómunkámat röviden bemutassam, ráhangolódva PhD disszertációm megírására is, amelyet 2014. május végéig kell benyújtanom.

Kutatási témám komplex racionális függvényrendszerek tanulmányozása, illetve ezek alkalmazási lehetőségeinek vizsgálata a jelfeldolgozás területén, különös tekintettel EKG görbék feldolgozására.

Komplex racionális függvények alatt egyszerű tört alakjában felírható, komplex változós, komplex értékű függvényeket értünk. Jelfeldolgozás szempontjából ezeknek a komplex egységkörön felvett értékeit

*ELTE Informatikai Kar

**2003–2011



1. ábra. Egy valódi EKG görbe egy szegmense (balra) és egy azt közelítő függvény (jobbra).

vizsgáljuk, majd vetjük össze a feldolgozni kívánt jellel. Matematikai szempontból pedig érdekes kérdés többek között az ezekből képzett ortogonális rendszerek, a gyors feldolgozást lehetővé tevő FFT-szerű algoritmusok vizsgálata, kidolgozása.

Az EKG görbék, avagy ElektroKardioGramok az emberi szív működése során bekövetkező elektromos változásokat rögzítik. Ezek vizsgálata igen elterjedt eszköz az orvosok kezében különféle betegségek diagnosztizálása céljából. Manapság már a hagyományos papír alapú megoldáson túl, vagy ahelyett sokszor digitálisan tárolják és továbbítják az EKG felvételeket. Ezek hatékony és megbízható feldolgozása: tömörítése, zajszűrése, szegmentálása, elemzése egyre gyakoribb és igen összetett feladat.

Komplex racionális függvények, illetve függvényrendszerek EKG görbék esetében való felhasználási lehetőségeinek alaposabb vizsgálatát – az utóbbi évtizedek racionális approximációval kapcsolatos matematikai eredményei, és a jelfeldolgozásban történő egyre intenzívebb alkalmazása mellett – az a tény is motiválja, hogy már egyszerű racionális függvények képe is nagyon hasonlít az EKG jellegzetes részeihez. Néhány paraméter megfelelő megválasztásával pedig egy-egy EKG-szegmenst nagyon jól közelítő függvényt kaphatunk. Például az 1. ábrán balra egy valódi EKG jel egy szegmensét láthatjuk (a PhysioNet interneten elérhető Apnea-ECG adatbázisából), jobbra pedig egy ehhez nagyon hasonló függvényt, melyet racionális függvények segítségével állítottunk elő.

A következő oldalakon részletesebben bemutatjuk a szóban forgó racionális függvényeket, azok tulajdonságait. Bemutatunk néhány ezekhez kapcsolódó matematikai eredményt, további érdekességet is. Ismertetjük azokat az algoritmusokat, amelyek segítségével megtaláltuk

egy-egy EKG-szegmenshez a függvényrendszerek megfelelő paramétereit, valamint az ezekhez kapcsolódó újításainkat, fejlesztéseinket.

Munkánk során mindvégig figyelmet fordítottunk az adott téma megfelelő szemléltetésére, minél szebb illusztrációk elkészítésére is. Ebben az írásban – annak összefoglaló jellegére való tekintettel – több esetben egy-egy kérdés részletes formális megválaszolása helyett hagyjuk inkább a kapcsolódó ábrákat beszélni.

Hasonló okokból a szakirodalmi hivatkozásokat indirekt módon bocsátjuk az érdeklődő Olvasó rendelkezésére: méghozzá saját publikációinkra (azok irodalomjegyzékére) hivatkozva.

2. Racionális függvényrendszerek

Ebben a fejezetben bevezetjük az alkalmazott jelöléseket, valamint a használt komplex racionális függvényeket, függvényrendszereket, bemutatjuk ezek néhány tulajdonságát, valamint formalizáljuk EKG görbék (illetve tetszőleges feldolgozandó jelek) approximációjának feladatát.

Jelölje $\mathbb{C}$ a komplex számok halmazát, $\mathbb{D} := \{z \in \mathbb{C} : |z| < 1\}$ a „diszket”, azaz a komplex egységkör belsejét, $\mathbb{T} := \{z \in \mathbb{C} : |z| = 1\}$ a „tóruszt”, azaz a komplex egységkört, valamint $\mathbb{D}^* := \mathbb{C} \setminus (\mathbb{D} \cup \mathbb{T})$. Természetes számok alatt az $\mathbb{N} := \{1, 2, 3, \dots\}$ halmazt fogjuk érteni. A $\mathbb{D}$ -n analitikus és $\mathbb{D} \cup \mathbb{T}$ -n folytonos függvények halmazát, azaz a „diszk algebrát” jelölje $\mathcal{A}$. Továbbá vezessük be a

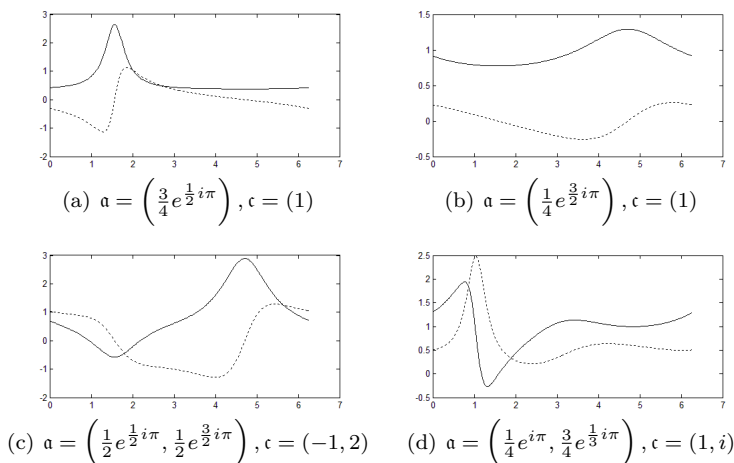
$$\langle \cdot, \cdot \rangle : \mathcal{A} \times \mathcal{A} \rightarrow \mathbb{C}, \quad \langle f, g \rangle = \frac{1}{2\pi} \int_0^{2\pi} f(e^{it}) \cdot \overline{g}(e^{it}) dt = \int_{\mathbb{T}} f(z) \cdot \overline{g}(z) dz$$

skaláris szorzatot, melyben tehát a $\mathbb{T}$ -n vett szorzatintegrált számítjuk.

Induljunk ki a következő racionális függvényekből:

$$\varphi_n(z) := \frac{1}{(1 - \overline{a_n}z)^{m_n}} \quad (z \in \mathbb{C} \setminus \{1/\overline{a_n}\}; m \in \mathbb{N}; n = 1, \dots, m),$$

ahol $a_n \in \mathbb{D}$ ($n = 1, \dots, m$) paraméterek és $m_n = \sum_{i \leq n} \chi(a_i = a_n)$ pedig az a_n paraméter multiplicitása. Megfigyelhetjük, hogy φ_n -nek m_n -edrendű pólusa van az $1/\overline{a_n} \in \mathbb{D}^*$ helyen, továbbá $\Phi := (\varphi_n : n =$



2. ábra. Példák komplex racionális függvényekre.

$1, \dots, m) \subset \mathcal{A}$. Például az $a, a, b, b \in \mathbb{D}$ paraméterekkel a következő függvényeket nyerjük:

$$\varphi_1 = \frac{1}{1 - \bar{a}z}, \quad \varphi_2 = \frac{1}{(1 - \bar{a}z)^2}, \quad \varphi_3 = \frac{1}{1 - \bar{b}z}, \quad \varphi_4 = \frac{1}{(1 - \bar{b}z)^2}.$$

Ezeket a függvényeket jelfeldolgozás céljából az egységkörön vizsgáljuk, tehát egy f függvény esetén az $f(e^{it})$ ($t \in [0, 2\pi]$) értékeket, ezek valós illetve képzetes részét. A 2. ábrán néhány példát láthatunk ilyen függvényekből képzett egy- és kéttagú lineáris kombinációkra, azaz $f = \sum_{n=1}^m c_n \varphi_n$ alakú függvényekre. A paraméterek megadásához a tömörebb $\alpha = (a_1, \dots, a_m)$, $\varsigma = (c_1, \dots, c_m)$ jelölést használtuk. (A folytonos vonal a valós részt, a szaggatott vonal a képzetes részt mutatja.)

Kiderül, hogy 3–4 paraméter, illetve hozzájuk tartozó együtthatók megfelelő megválasztásával már igen jó közelítést kaphatjuk EKG görbék szegmenseinek. Kérdés azonban, hogy hogyan keressük meg a megfelelő paramétereket és hogyan határozzuk meg az ismeretlen együtthatókat.

Az együtthatók kiszámítására kínálkozik egyszerű módszer, ha ortogonális vagy ortonormált rendszerből indulunk ki. A Φ lineárisan

független rendszerre a Gram–Schmidt-féle ortogonalizációs eljárást alkalmazva egy $\Psi := (\psi_n : n = 1, \dots, m)$ ortonormált rendszert, úgynevezett Malmquist–Takenaka-rendszert kapunk a fenti skaláris szorzatra nézve. Ennek elemei kifejezhetők a

$$B_a(z) := \frac{z - a}{1 - \bar{a}z} \quad (a \in \mathbb{D}; z \in \mathbb{C} \setminus \{1/\bar{a}\})$$

Blaschke-függvények segítségével, méghozzá

$$\psi_n(z) = \frac{\sqrt{1 - |a_n|^2}}{1 - \bar{a}_n z} \prod_{k=1}^{n-1} B_{a_k}(z) \quad (z \in \mathbb{C}; n = 1, \dots, m),$$

amely alak egyúttal ψ_n értékeinek egy kényelmesebb kiszámítási módját is sugallja az ortogonalizációs eljárás tényleges végrehajtása helyett. Megállapíthatjuk, hogy a Φ és Ψ rendszerek által kifeszített m -dimenziós alterek megegyeznek, azaz $\text{span } \Phi = \text{span } \Psi$.

A Malmquist–Takenaka-rendszer ortonormáltságát kihasználva valamely adott $f \in \mathcal{A}$ függvény $\mathcal{P}_\Psi f = \mathcal{P}_{a_1, \dots, a_m} f$ ortogonális projekcióját a $\text{span } \Psi$ altérre kiszámíthatjuk a következő formula alapján:

$$\mathcal{P}_\Psi f = \sum_{n=1}^m \langle f, \psi_n \rangle \psi_n,$$

tehát az ismeretlen együtthatókat az $\langle f, \psi_n \rangle$ skalárszorzatok adják.

A megfelelő paraméterek keresése egy újabb feladat. A problémát a következőképpen formalizálhatjuk. Jelölje $\mathcal{E}_\Psi f = \mathcal{E}_{a_1, \dots, a_m} f$ az f függvény legjobb közelítését $\|\cdot\|_2$ -ban a $\text{span } \Psi$ altéren, azaz

$$\mathcal{E}_\Psi f := \|f - \mathcal{P}_\Psi f\|_2 = \sqrt{\langle f - \mathcal{P}_\Psi f, f - \mathcal{P}_\Psi f \rangle}.$$

A célunk egy (a $\mathbb{T}$ -n felvett értékei által) adott $f \in \mathcal{A}$ függvény – felfoghatjuk így az EKG görbe egy szegmensét – és $m \in \mathbb{N}$ rögzített dimenziószám esetén az $\mathcal{E}_\Psi f$ érték minimalizálása a rendszer $a_1, a_2, \dots, a_m$ paramétereinek „jó” megválasztása által. A paraméterek jó megválasztásának problémájára és annak megoldási lehetőségeire visszatérünk a 4. fejezetben.

Természetesen gépi számításaink során a függvények és a skaláris szorzat $\mathbb{T}$ egyenletes felosztása mentén diszkrétizált változataival dolgozunk. Például egy EKG szegmens általában 200–300 mintavételezett értékből áll össze.

A nemrég bevezetett Blaschke-függvényekről külön is érdemes néhány szót ejteni, ezek sok érdekes tulajdonsággal rendelkeznek, a matematika több ágában és számos alkalmazási területen felbukkannak.

A Blaschke-függvények egyaránt $\mathbb{D} \rightarrow \mathbb{D}$, illetve $\mathbb{T} \rightarrow \mathbb{T}$ bijekciók. A B_a függvénynek egy darab egyszeres zérushelye van a $z = a$ pontban, valamint egy darab elsőrendű pólusa van a $z = 1/\bar{a}$ pontban – ami voltaképpen az a pont egységkörre vett tükörképe (inverzió által létesített képe). Invertálható függvényekről van szó, méghozzá B_a inverze B_{-a} . Speciális esetként, az $a = 0$ választással $B_0(z) = z$ adódik, ehhez kapcsolódóan – az egységkörön felvett értékeket nézve – a Blaschke-függvények a trigonometrikus függvények általánosításaként is felfoghatók. Továbbá a hiperbolikus geometria Poincaré-féle körmodelljét tekintve ezek a függvények egybevágósági transzformációkat definiálnak.

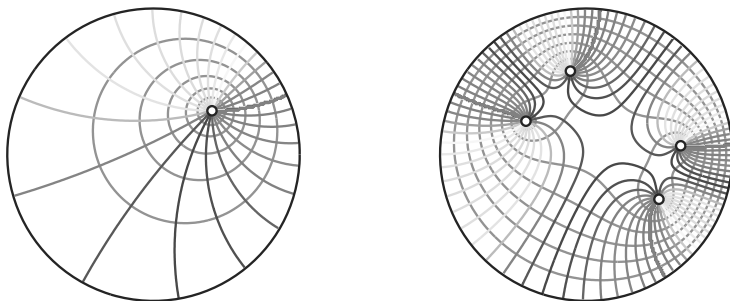
Bizonyos esetekben érdemes bevezetni egy második paramétert, egy $\varepsilon \in \mathbb{T}$ szorzótényezőt a fenti $B_a(z)$ értékhez. Az így kapott kétparaméteres függvénycsaládnak talán még szebb tulajdonságai is vannak a fentieken kívül. Egyrészt csoportot alkotnak függvények kompozíciójára nézve – ez az egyparaméteres Blaschke-függvényekre nem igaz. Másrészt pedig általuk felírható a diszk minden iránytartó egybevágósági transzformációja.

Definiálhatjuk a Blaschke-szorzatok fogalmát is. Egy Blaschke-szorzat nem más, mint néhány (véges számú) Blaschke-függvény szorzata, azaz adott $n \in \mathbb{N}$ és $a_1, \dots, a_n \in \mathbb{D}$ paraméterek esetén

$$A_{a_1, \dots, a_n}(z) := \prod_{k=1}^n B_{a_k}(z) \quad (z \in \mathbb{C}).$$

A Blaschke-szorzatok is fontos szerepet játszanak különféle approximációs és interpolációs eljárások konstruálásában, például – ahogy láthattuk – a Malmquist–Takenaka-rendszerek felírásában is. Egy érdekes tulajdonságuk, hogy egy n -tényezős Blaschke-szorzat és $w \in \mathbb{T}$ esetén az $A(z) = w$ egyenletnek mindig n különböző gyöke van.

A 3. ábrán egy Blaschke-függvényt, valamint egy négytényezős Blaschke-szorzatot ábrázoltunk $\mathbb{D}$ -n. A komplex értékek abszolút értékét kvázi-koncentrikus szintvonalak, argumentumát pedig különféle árnyalatú, a zérushelyekből induló vonalak mutatják.



3. ábra. Balra: példa Blaschke-függvényre. Jobbra: egy négytényezős Blaschke-szorzat. A zérushelyeket is jelöltük (mindkét ábrán).

3. Néhány matematikai eredmény

A kutatás fő csapásáról egy kicsit letérve, egy rövid kitérőben tömören ismertetünk néhány matematikai jellegű eredményt, melyek motivációját azonban az alkalmazott jelfeldolgozás területén találhatjuk. Az eredmények a Malmquist–Takenaka-rendszerek Dirichlet-féle magfüggvényének „szeleteiként” megadható újabb ortogonális rendszerekkel [4], valamint Blaschke-szorzatok kompozícióival kapcsolatosak [11].

A Fourier-analízis témakörében jól ismert a bizonyos sorfejtések n -edik kezdőszeletének integráltranszformáció segítségével történő megadását lehetővé tévő Dirichlet-féle magfüggvény. Ennek általános alakjának megadásához induljunk ki egy tetszőleges $(\phi_k : k \in \mathbb{N})$ ortogonális rendszerből, a korábban bevezetett skaláris szorzatra nézve, amely szerint majd egy $f \in \mathcal{A}$ függvényt Fourier-sorba fejtünk.

Az f függvény Fourier-sorának n -edik részletösszege ($n \in \mathbb{N}$) így írható fel a $z \in \mathbb{T}$ pontban:

$$(S_n f)(z) = \int_{\mathbb{T}} f(\zeta) \overline{\phi_k}(\zeta) d\zeta \Bigg|_{k=1}^n \phi_k(z) = \int_{\mathbb{T}} f(\zeta) \underbrace{\left(\sum_{k=1}^n \phi_k(z) \overline{\phi_k}(\zeta) \right)}_{D_n(z, \zeta)} d\zeta.$$

Ezzel összhangban a (kétváltozós) Dirichlet-magfüggvény általánosan tehát a következő alakot ölti:

$$D_n: \mathbb{T} \times \mathbb{T} \rightarrow \mathbb{C}, \quad D_n(z, \zeta) = \sum_{k=1}^n \phi_k(z) \overline{\phi_k(\zeta)}.$$

Beláttuk a következőt.

3.1. Tétel. Az $n \in \mathbb{N}$ és $a_k \in \mathbb{D}$ ($k = 1, \dots, n$) paraméterek által adott ψ_k ($k = 1, \dots, n$) Malmquist–Takenaka-rendszer Dirichlet-magfüggvényére igazak az alábbi összefüggések:

$$D_n(z, \zeta) = \sum_{k=1}^n \psi_k(z) \overline{\psi_k(\zeta)} = \frac{\prod_{k=1}^n B_{a_k}(z) \overline{B_{a_k}(\zeta)} - 1}{z \bar{\zeta} - 1} \quad (z \neq \zeta),$$

$$D_n(z, z) = \sum_{k=1}^n \psi_k(z) \overline{\psi_k(z)} = \sum_{k=1}^n \frac{1 - |a_k|^2}{|1 - \overline{a_k} z|^2} \quad (z = \zeta).$$

Az iménti paraméterek mellett tetszőleges $w \in \mathbb{T}$ értéket választva, az

$$A_{a_1, \dots, a_n}(z) = w$$

egyenlet $\zeta_1, \zeta_2, \dots, \zeta_n \in \mathbb{T}$ különböző gyökeire

$$D_n(\zeta_k, \zeta_l) = \delta_{k,l} \cdot D_n(\zeta_k, \zeta_k) \quad (k, l = 1, 2, \dots, n)$$

teljesül, ahol $\delta_{k,l}$ a Kronecker-féle szimbólum. Ezt az állítást kihasználva, a magfüggvény megfelelő „szeleteit” vizsgálva azonban máris előállt az új ortogonális rendszer. Méghozzá:

3.2. Tétel. A fenti jelölésekkel a

$$H_k: \mathbb{T} \rightarrow \mathbb{C}, \quad H_k(z) := D_n(z, \zeta_k) \quad (k = 1, 2, \dots, n)$$

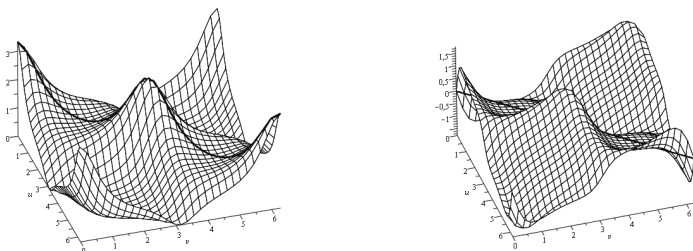
függvények ortogonális rendszert alkotnak,

$$\langle H_k, H_l \rangle = \delta_{k,l} \cdot D_n(\zeta_k, \zeta_k) \quad (k, l = 1, 2, \dots, n)$$

teljesül.

Diszkrét ortogonalitási tulajdonságok is beláthatók.

A 4. ábrán $n = 2$ esetén láthatunk egy példát Dirichlet-magfüggvényre. A vízszintes tengelyeken z és ζ argumentuma szerepel.



4. ábra. Egy Malmquist–Takenaka-rendszer Dirichlet-magfüggvénye. (Balra: valós rész, jobbra: képzetes rész.)

A koordináta-rendszer síkjaival párhuzamos vonalak a korábban említett „szeletek”. (Az átló mentén pedig ebben a speciális esetben a Poisson-féle magfüggvényt láthatjuk.)

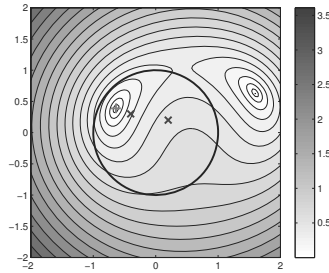
Gyors (FFT-szerű) algoritmusok konstruálására alkalmas rendszerek vizsgálata közben merült fel a következő probléma: Hogyan választhatók meg Blaschke-szorzatok paraméterei úgy, hogy azok kompozíciójának zérushelyei valamely előírt helyekre essenek? (Esetleg EKG görbék közelítéséhez optimális helyekre...) Munkánkban a legegyszerűbb nemtriviális esetet tisztáztuk, amikor két darab kéttényezős Blaschke-szorzatról van szó. Ezek kompozíciója egy négytényezős Blaschke-szorzat, négy különböző zérushellyel.

Az elemzés során bevezettük a reciprok Blaschke-függvény fogalmát, és beláttuk annak egyértelmű létezését.

3.3. Definíció. Rögzített $b_1, b_2, a_2 \in \mathbb{D}$ értékek esetén az $a_1 \in \mathbb{D}$ paraméterű Blaschke-függvényt a B_{a_2} függvény reciprok Blaschke-függvényének nevezzük a b_1 és b_2 pontokra nézve, ha a következő összefüggés teljesül:

$$\frac{B_{a_2}(b_1)}{B_{a_2}(b_2)} = \frac{B_{a_1}(b_2)}{B_{a_1}(b_1)} = \left(\frac{B_{a_1}(b_1)}{B_{a_1}(b_2)} \right)^{-1}.$$

Reciprok Blaschke-függvények létezésének numerikus vizsgálata egy függvény $\mathbb{D}$ -beli zérushelyének lokalizálására vezet. Ezt csak egyszerűen szintvonalas képével mutatjuk be az 5. ábrán – amely egy, az Olvasó



5. ábra. Reciprok Blaschke-függvények numerikus elemzése.

felé tekintő teknősbékára emlékeztet –, a b_1, b_2 paramétereket iksz jelöli, a keresett zérushely numerikus közelítését pedig pluszjel.

Különös módon a reciprok Blaschke-függvények létezését és egyértelműségét kimondó tételben meglepő szerep jut újfent a Blaschke-függvényeknek.

3.4. Tétel. Rögzített $b_1, b_2, a_2 \in \mathbb{D}$ értékek esetén egyértelműen létezik olyan $a_1 \in \mathbb{D}$, hogy B_{a_1} reciprok Blaschke-függvénye a B_{a_2} függvénynek a b_1 és b_2 pontokra nézve. Méghozzá

$$a_1 = -B_{p(b_1, b_2)}(a_2),$$

ahol

$$p: \mathbb{D} \times \mathbb{D} \rightarrow \mathbb{D}, \quad p(b_1, b_2) = \frac{(b_1 + b_2) - \overline{(b_1 + b_2)}b_1b_2}{1 - |b_1b_2|^2}.$$

Kiderül azonban, hogy nem adható meg tetszőleges $\mathbb{D}$ -beli pontnégyeshez két olyan kéttényezős Blaschke-szorzat, hogy az előírt pontok legyenek a kompozíció zérushelyei. Viszont igaz a következő – itt a pontos konstrukció megadása nélkül közölt – tétel.

3.5. Tétel. Bármely $b_1, b_2, b_3 \in \mathbb{D}$ értékekhez (sorrendjüktől függően) egyértelműen létezik $b_4 \in \mathbb{D}$ úgy, hogy egy $A_{a_3, a_4} \circ A_{a_1, a_2}$ alakú kompozíció zérushelye pontosan ez a négy pont. Végtelen sok ilyen kompozíció létezik: $a_1 \in \mathbb{D}$ tetszőlegesen megválasztható, $a_2, a_3, a_4 \in \mathbb{D}$ pedig a_1 segítségével egyértelműen kifejezhető.

A 3. ábra négytényezős Blaschke-szorzata egyébként egyben egy ilyen alakú kompozíció is. Megfigyelhető a zérushelyek jellegzetes (hiperbolikus) paralelogrammaszerű elhelyezkedése.

4. Közelítő algoritmusok, fejlesztések

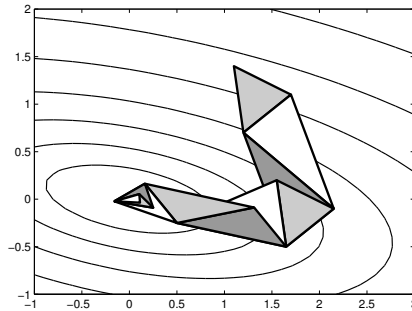
Láthattuk, hogy ortogonális rendszereket, például a Malmquist–Takenaka-rendszert alkalmazva egy jel (négyzetes) közelítéséhez szükséges együtthatókat könnyen megkaphatjuk, ezek skaláris szorzat alakjában kifejezhetők. A szóban forgó racionális rendszerek $a_1, a_2, \dots, a_m \in \mathbb{D}$ paraméterei azonban nem rögzítettek, ezek szabadon megválaszthatók, így maga a rendszer is úgymond adaptálható a feldolgozandó jelhez. Kérdés azonban, hogy hogyan keressük meg egy adott jelhez legjobban illeszkedő paramétereket? Ebben a szakaszban az erre használt algoritmusokat, illetve kapcsolódó újításainkat, a kutatások támogatására elkészített MATLAB programcsomagjainkat mutatjuk be tömören.

Felmerültek Monte-Carlo-jellegű módszerek, szimulált lehűtési technikák, raj-optimalizációs eljárások is, azonban a Nelder–Mead-féle szimplex algoritmus volt az első módszer, amellyel sikerült az EKG görbék esetében a paraméterek (avagy pólusok) apriori becslése nélkül jó közelítést kapnunk azok optimális értékére [1, 7].

A Nelder–Mead-féle szimplex módszer – nem összekeverendő a lineáris programozásban ismert szimplex módszerrel – egy általános nemlineáris numerikus optimalizációs eljárás tetszőleges $\mathbb{R}^n \rightarrow \mathbb{R}$ függvény lokális minimumhelyeinek meghatározására. Igen elterjedten alkalmazzák gyorsasága, egyszerűsége és könnyű számolhatósága miatt a természettudományos és mérnöki kutatásokban annak ellenére, hogy matematikai értelemben a módszer konvergenciájával kapcsolatban rendkívül kevés állítást tudunk bizonyítani.

Az algoritmus működése vázlatosan a következőképpen írható le:

- Induljunk ki egy tetszőleges (általában kicsit átmérőjű, nem degenerált) szimplexből, azaz $\mathbb{R}^n$ esetén $n + 1$ pontból (síkban egy háromszögből, térben egy tetraéderből); valamint határozzuk meg a csúcspontokban a függvényértékeket. Azonosítsuk a legnagyobb („legrosszabb”) és legkisebb („legjobb”) függvényértékű csúcsokat.



6. ábra. A Nelder–Mead-algoritmus működése.

- Tükrözzük a legrosszabb csúcsot a többi csúcs súlypontjára, és menjünk tovább az így kapott szimplexszel. A *tükrözés* lépésén kívül a függvényértékek alapján sor kerülhet a *nyújtás*, *összehúzás*, valamint a *zsugorítás* lépéseire is; amelyek alkalmazása által nem egyszerűen egy rács mentén vizsgáljuk csak a függvényértékeket, hanem a szimplex úgymond „alkalmazkodik a terephez”, megnyúlik a lankás vidékeken, végül pedig a végső minimum köré zsugorodik.
- Állítsuk meg az eljárást ha a csúcsokban tapasztalt függvényértékek szórása már megfelelően kicsi, vagy ha már elég sok lépést megtettünk.

A tapasztalatok azt mutatják, hogy ez az algoritmus elég megbízhatóan megadja az indítási helyéhez legközelebbi lokális minimumhelyet. A legtöbb gyakorlati probléma esetében ez már elegendő is, mindazonáltal jelenleg is kutatások tárgya pontosabb konvergencia-tételek bizonyítása. Az algoritmus elterjedtségét az is jól mutatja, hogy a MATLAB programcsomag beépített `fminsearch` utasítása is ezt a módszert használja.

A 6. ábrán megfigyelhető a Nelder–Mead-módszer működése, miközben egy kétváltozós, másodfokú függvény minimumát keressük. Az ábra az említett lépéseket – tükrözés, nyújtás, összehúzás – is szemlélteti.

A Malmquist–Takenaka-rendszer paramétereit $\mathbb{D}$ -n keressük, viszont a Nelder–Mead-algoritmus $\mathbb{R}^n$ -en dolgozik. A problémát egy megfelelő transzformációval orvosolhatjuk, $\mathbb{R}^{2m}$ és $\mathbb{D}^m$ között páronként a

következő formulával adhatunk meg bijektív leképezést:

$$\mathbb{R}^2 \ni (u, v) \longleftrightarrow z = \frac{u}{\sqrt{1+u^2+v^2}} + \frac{v}{\sqrt{1+u^2+v^2}}i \in \mathbb{D}.$$

Ez a hagyományos leképezés egy képzeletben $\mathbb{D}$ -re helyezett félgömb és az arra fektetett sík (mint $\mathbb{R}^2$) segítségével származtatható.

Említettük, hogy a Blaschke-függvények megfeleltethetők az egybevágósági transzformációknak a Bolyai–Lobacsevszki-féle hiperbolikus geometria Poincaré-féle körmodelljén, ahol a síkot az egységkör, $\mathbb{D}$ adja. A Nelder–Mead-algoritmus lépései pedig pusztán geometriai fogalmakkal megadhatók (súlypont, tükrözés). Ennek folytán felmerült ezen algoritmus hiperbolikus változata elkészítésének ötlete is [7]. Így a keresést közvetlenül az egységkőrön végezhetjük.

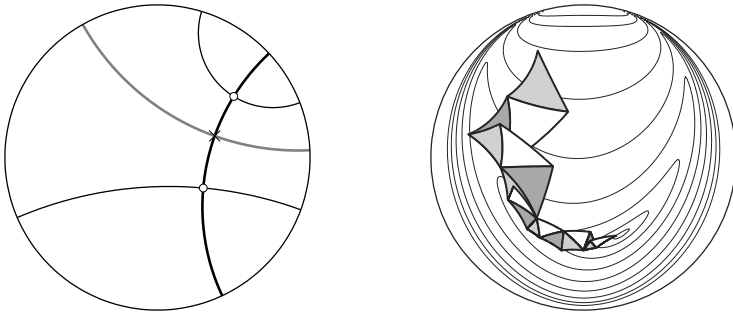
Az algoritmus adaptációja elkészült, 2 és 3 dimenzióban rendelkezésre áll a MATLAB rendszerben készített implementációnk [12].

A Poincaré-féle körmodellben a síkot az egységkör (belső) pontjai adják, az egyenesek pedig az egységkört merőlegesen metsző átmérők, illetve körvonalak. Ez valóban a hiperbolikus geometria egy modellje, hiszen itt egy egyeneshez egy rajta kívül fekvő ponton keresztül több, az eredeti egyenest nem metsző („párhuzamos”) egyenes is húzható. A hiperbolikus geometria konstrukcióit nem ismertetjük részletesen, sem azok analitikus leírását racionális komplex függvények segítségével.

A 7. ábrán bemutatjuk a Poincaré-féle modell néhány alapvető szerkesztését: két pontra illesztett egyenes, erre állított merőleges egyenesek, szakaszfelező merőleges. Valamint egy $\mathbb{D}$ -n értelmezett függvény minimalizálása közben láthatjuk a hiperbolikus síkra átültetett Nelder–Mead-algoritmus működését.

Munkánk során végig a MATLAB numerikus matematikai programcsomagot használtuk erős matematikai képességeire, a jelfeldolgozás könnyedségére, jó programozhatóságára és kiváló grafikai lehetőségeire támaszkodva. A kutatás során megvalósított programjaink is ebben a rendszerben íródtak. Tömören megemlítjük még további fejlesztéseinket, eredményeinket.

- Blaschke-függvények és Blaschke-szorzatok segítségével elegánsan megadhatók nem egyenletes felosztások is. Egy nem egyenletes diszkretizáció hasznos lehet, ha egy jel esetében vannak lassan



7. ábra. Balra: a hiperbolikus geometria néhány egyszerű eleme. Jobbra: a hiperbolikus Nelder–Mead-módszer.

változó, viszonylag egyenes részek (itt kevesebb pont elég), illetve gyorsan változó részek (itt több pont kell) – egy EKG görbe pont ilyen [2]. Viszont a felosztás osztópontjainak meghatározása szintén egy külön feladat, amelynek lehetséges a természetes úton adódó megoldásnál gyorsabb kivitelezése [3].

- Racionális rendszereken is alkalmazható FFT-szerű konstrukció. Itt Blaschke-szorzatok kompozícióival, *szorzatrendszerekkel* van dolgunk, de szintén nem egyenletes felosztást kapunk – szemben a hagyományos trigonometrikus FFT egyenletes felosztásával [6].
- Természetesen összegyűjtöttük a 2. fejezetben bemutatott racionális függvényrendszerek könnyed előállítását és használatát, valamint a közelítő algoritmusok alkalmazását lehetővé tevő programjainkat is, kiegészítve különféle diszkrét változatokkal és szemléltető eszközzel. Ez a gyűjtemény a RAIT – *Rational Approximation and Interpolation Toolbox* – nevet kapta [8, 9].

5. Alkalmazás: EKG szegmensek közelítése

Miután meggyőződünk róla, hogy az ismertetett elméleti konstrukciók (ortogonális rendszerek, közelítő algoritmusok) alkalmasak mester-

ségesen előállított racionális függvények $\mathbb{T}$ -n felvett értékeiből a paramétereket megfelelő pontosságú visszafejtésére [1], megvizsgáltuk alkalmazásukat valós EKG szegmensek közelítése esetében is.

Példaként ismételten felhívnanék a figyelmet az 1. ábrára, ahol egy valódi EKG görbe egy szegmensét és annak egy közelítését mutatjuk be, amit komplex racionális rendszerek és a Nelder–Mead-algoritmus segítségével számítottunk.

Munkánkat több folyóiratban és nemzetközi konferencián bemutattuk, figyelmet fordítva a javasolt feldolgozási módszerek elméleti hátterére, azok újdonságára [5, 7], valamint a racionális rendszerek segítségével elért közelítések mérnöki szemmel történő elemzésére, a közelítés hibájának különböző metrikákban mért jellemzésére is [7, 10]. A közelítés persze mást is jelent: egyúttal a görbék simítását, szűrését is nyerjük. Sőt, mivel csupán néhány komplex paraméter elegendő egy szegmens jól közelítő leírásához, egyben algoritmust nyertünk ezen jelek tömörítésére is, ami nagy mennyiségű EKG jel tárolása esetén lehet előnyös. Ekkor vizsgálható a tömörítés mértéke is.

A tömörítés mértékének elemzésénél két kézenfekvő mennyiséget is szokás definiálni. Méghozzá, ha az eredeti jel méretét valamely mértékegységben (tárolt értékek száma, kilobyte stb.) E jelöli, a tömörített jel méretét pedig ugyanabban a mértékegységben T , akkor vizsgálható a tömörítési eljárás *tömörítési aránya* (*Compression Ratio*, rövid. CR), valamint a *tárhely-megtakarítás* (*Space Saving*, rövid. SS), méghozzá

$$\text{CR} = \frac{E}{T}, \quad \text{SS} = 1 - \frac{T}{E}.$$

A tárhely-megtakarítást szokás százalékban is megadni.

A közelítés hibájának jellemzésére pedig a hagyományos normák (integrál-, négyzetes-, maximum-norma) mellett orvosi jelekre alkalmazott algoritmusok vizsgálatánál elterjedt a PRD (*Percentage Root-mean-square Difference*) rövidítésű mennyiség, egyfajta százalékban megadott négyzetes eltérés megadása is. Méghozzá ha egy N hosszúságú jel esetén az eredeti értékeket $x(k)$, a közelítő értékeket $\tilde{x}(k)$ jelöli, akkor

$$\text{PRD} = \sqrt{\frac{\sum_{k=1}^N (x(k) - \tilde{x}(k))^2}{\sum_{k=1}^N x(k)^2}} \times 100.$$

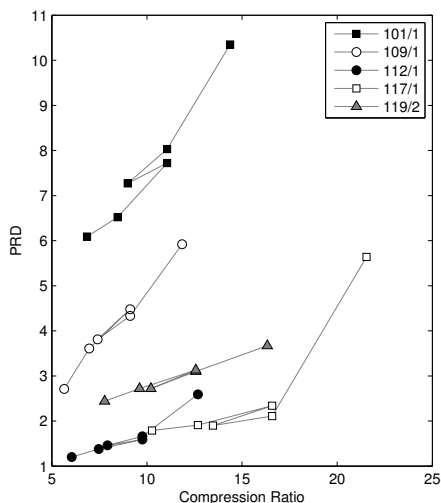
Az 1. táblázatban összefoglaltuk a CR, SS és PRD értékeket az egyik

		min	max	átlag	szórás
3 par., 2 mul.	CR	11.85	21.54	15.91	2.63
	SS	91.56	95.36	93.56	0.98
	PRD	2.59	19.45	8.86	3.70
3 par., 3 mul.	CR	9.12	16.58	12.24	2.03
	SS	89.03	93.97	91.63	1.28
	PRD	1.66	17.03	6.55	3.48
3 par., 4 mul.	CR	7.41	13.47	9.95	1.65
	SS	86.50	92.58	89.70	1.57
	PRD	1.46	13.93	5.21	3.15
4 par., 2 mul.	CR	9.12	16.58	12.24	2.03
	SS	89.03	93.97	91.63	1.28
	PRD	1.59	14.02	6.13	2.92
4 par., 3 mul.	CR	6.97	12.68	9.37	1.55
	SS	85.66	92.12	89.06	1.67
	PRD	1.38	12.60	4.75	2.76
4 par., 4 mul.	CR	5.65	10.27	7.58	1.26
	SS	82.29	90.26	86.49	2.06
	PRD	0.94	11.47	4.00	2.42

1. táblázat. EKG szegmensek közelítése: mérési eredmények.

kísérletünkben megvizsgált 29 EKG jel esetén (ezek a PhysioNet MIT-BIH Arrhythmia adatbázisából valók), különböző beállítások mellett. Például a „3 par., 4 mul.” azt jelenti, hogy 3 különböző $\mathbb{D}$ -beli paramétert kerestünk, az ortogonális rendszer konstruálásához pedig mindegyiket négyszer (4 multiplicitással) vettük. Összességében azt láthatjuk, hogy akár közel tízszeres tömörítési arányt is elérhetünk viszonylag kicsi hiba mellett. (5%-hoz közeli, vagy az alatti PRD érték már elég jó.) A fentiekből kiválasztott 5 jel esetén a 8. ábra mutatja grafikusan a CR és PRD értékeket. Nyilvánvalóan minél több paramétert alkalmazunk, annál pontosabb közelítést kapunk (a PRD csökken), viszont annál több értéket is kell tárolnunk (a CR is csökken).

Persze valós alkalmazások szempontjából a legfontosabb mérce EKG felvételek elemzéséhez jól értő szakorvosok véleménye volna, amit azonban munkánk keretében sajnos nem tudtunk nagyobb kapacitásban



8. ábra. EKG szegmensek közelítése: CR és PRD.

igénybe venni. Viszont kaptunk mind biztató, érdeklődő megjegyzéseket, mind ötleteket, javaslatokat továbbfejlesztési lehetőségekre.

6. Összefoglalás

Ebben az írásban egy tömör összefoglalását adtuk doktoranduszi kutatómunkánknak racionális függvényrendszerek alkalmazási lehetőségeinek vizsgálatáról a jelfeldolgozás területén, különös tekintettel EKG görbék analizésére.

Bemutattuk az alkalmazott komplex függvényeket, illetve függvényrendszereket (Blaschke-függvények, Malmquist–Takenaka-rendszerek). Ismertettük az ezekkel kapcsolatos matematikai eredményeinket, a közelítő algoritmusokkal kapcsolatos újításokat (pl. hiperbolikus Nelder–Mead-algoritmus), kifejlesztett programcsomagokat (pl. RAIT), továbbá mindezek gyakorlati alkalmazása során nyert tapasztalatainkat. Ez utóbbiakat röviden összefoglalva: EKG görbéknek egy igen elegáns leírási

módját adtuk meg, biztató numerikus eredményekkel alátámasztva.

Természetesen szinte bármelyik említett témakörrel kapcsolatban megemlíthetnénk további kutatási irányokat mind matematikusi, mind mérnöki szemmel nézve. A teljesség igénye nélkül pusztán egy-két példa nyitott kérdésekre: reciprok Blaschke-függvények kapcsán Apollóniusz köreinek hiperbolikus analogonjainak nyomaira bukkantunk, kidolgozásuk még előttünk áll; valamint a racionális FFT algoritmusok EKG görbék esetében való alkalmazása és hatékonyságának vizsgálata szintén egy hátrálévő feladat.

Hivatkozások

- [1] L. Lócsi, Approximating Poles of Complex Rational Functions, *Acta Univ. Sapientiae, Math.*, 1/2 (2009), pp. 169–182.
- [2] L. Lócsi, Discrete Approximation of ECG Signals, *Proc. 8th Int. Conf. on Appl. Inf.*, 2 (2010), pp. 45–52.
- [3] L. Lócsi, Calculating Non-Equidistant Discretizations Generated by Blaschke Products, *Acta Cybernetica*, 20 (2011), pp. 111–123.
- [4] L. Lócsi, Constructing Orthogonal Systems Using Blaschke Products, *Proc. 8th Joint Conf. on Math. and Comp. Sci.*, Math., (2011), pp. 43–50.
- [5] S. Fridli, L. Lócsi, F. Schipp, Rational Function Systems in ECG Processing, *Proc. 13th Inf. Conf. on Computer Aided Systems Theory*, LNCS 6927(1) (2011), pp. 88–95.
- [6] L. Lócsi, Rational FFT Implementation in Matlab, *Annales Univ. Sci. Budapest., Sect. Comp.*, 36 (2012), pp. 241–254.
- [7] S. Fridli, P. Kovács, L. Lócsi, F. Schipp, Rational Modeling of Multi-Lead QRS Complexes in ECG Signals, *Annales Univ. Sci. Budapest., Sect. Comp.*, 37 (2012), pp. 145–155.
- [8] P. Kovács, L. Lócsi, RAIT: the Rational Approximation and Interpolation Toolbox for Matlab, with Experiments on ECG Signals, *The International Journal of Advances in Telecommunications, Electrotechnics, Signals and Systems*, 1(2–3) (2012), pp. 67–75.

- [9] P. Kovács, L. Lócsi, RAIT: the Rational Approximation and Interpolation Toolbox for Matlab, *Proc. 35th IEEE Int. Conf. on Telecomm. and Sig. Proc.*, (2012), pp. 671–677.
- [10] L. Lócsi, P. Kovács, Processing ECG Signals Using Rational Function Systems, *Proc. 7th IEEE Int. Symp. on Medical Meas. and Appl.*, (2012), pp. 123–127.
- [11] L. Lócsi, An Inverse Problem with Compositions of Blaschke Products, *Mathematica Pannonica*, 24(1) (2013), pp. 141–156.
- [12] L. Lócsi, A hyperbolic Variant of the Nelder–Mead Simplex Method in Low Dimensions, *Acta Univ. Sapientiae, Math.*, 5(2) (2013), to appear.

Az EIT ICT Labs Master School

Lövei Péter*

Eötvös József Collegium**

petyalovei@gmail.com

2012 nyarán olvastam először az *EIT ICT Labs Master School* hirdetését az Informatikai Kar honlapján. Felkeltette az érdeklődésemet a képzés után, hiszen már korábban is külföldön szerettem volna folytatni a tanulmányaimat. Rövid „kutatómunkám” eredményeként kiderítettem, hogy 19 patinás európai egyetem és nagy nevű ipari partnerek részvételével az EIT ICT Labs egy olyan mesterképzést indított el, melyben a hallgatók az egy-egy évet az általuk kiválasztott egyetemen töltöttek. A képzés célja a magas színvonalú informatikai oktatáson túl az innováció elősegítése és a vállalkozási kedv támogatása, ezért a képzésnek része egy Innovációs és Vállalkozói tevékenységet oktató minor is. A 19 egyetem közül az egyik az *Eötvös Loránd Tudományegyetem*, a másik a *Budapesti Műszaki Egyetem*. Tudni kell hogy Budapest ad otthont az *EIT*-nak (*European Institute of Innovation and Technology*), melynek az EIT ICT Labs a leányintézménye.

Hét szakirány közül választhatnak az érdeklődő hallgatók, ezek közül három szakiránynak részesei a magyar egyetemek. A jelentkezés online történik. A jelentkezőknek egy motivációs levelet, önéletrajzot, a tanulmányi eredményük kivonatát kell csatolni. Két ajánlólevelet, akár egyetemi oktatóktól, akár korábbi munkáltatóktól kell beszerezni. Továbbá nyelvvizsga bizonyítványukat, valamint egy rövid üzleti ötletet kell elküldeniük. A beérkezett jelentkezési anyagok alapján az adott szakirányban résztvevő egyetemek képviselői döntenek a hallgatók felvételéről. A képzésben többféle ösztöndíj lehetőség van. A

*Aalvar Aalto Egyetem, Helsinki, Finnország

**2010–

legkiválóbbak havi ösztöndíjat, a képzési díj alóli mentességet, valamint egy egyszeri mobilitási csomagot kapnak.

Választásom a *Szolgáltatások Tervezése és Kivitelezése (Service Design and Engineering)* szakirányra esett. Jelentkeztem és a sikeres felvételt követően szeptemberben Helsinkiben az *Aalvar Aalto Egyetemen* tizenegy, köztük hat magyar társammal kezdtem meg tanulmányaimat ösztöndíjasként.

Augusztus közepén az egyetem szervezett minden nemzetközi mesterképzésen részt vevő diákjának egy nyolc napos *finn nyelvi és interkulturális kommunikációs kurzust*.

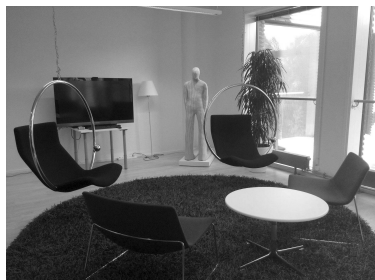
A szállásunk egy erdei konferenciaközpontban volt, nagyon közel egy tóhoz, így megtapasztalhattuk milyen csodás a finn természet. A finn nyelv hiába tartozik a finnugor nyelvcsaládba, mint a magyar, ez sajnos csak annyiban nyilvánul meg hogy mindkét nyelvtan nagyon nehéz, és így nem sikerült érdemi nyelvtudással gazdagodni ezen kurzus alatt. Az interkulturális kommunikációs kurzuson hallottak azonban nagyon hasznosak és érdekesek voltak. Az órán csoportokba voltunk osztva, minden csoport legalább kettő, de inkább három kontinens hallgatóiból állt össze. Ezekben a csoportokban dolgoztuk fel, hogyan reagálnak az adott kultúrákban bizonyos helyzetekre azok tagjai, mi magunk mit tenénk az adott helyzetekben, és végül de nem utolsósorban meghallgattuk az oktatóinktól, hogyan reagálnak a finnek. Ezután sok új ismerőssel és ismerettel gazdagabban kezdtem meg az egyetem által szervezett orientációs hetet.



1. ábra. Az egyetem főépülete

Az *orientációs hét* során nagyon sok hasznos információt hallhattunk az egyetemmel kapcsolatban, minden adminisztrációs teendőben kaptunk segítséget az egyetem dolgozóitól, megismerhettük az egyetem vezetését, hallhattunk egy nagyon érdekes előadást az európai agyuktatás jelenlegi helyzetéről, valamint megtanultuk hogyan kell ismerkedni.

Az EITs mesterképzésben résztvevő hallgatókat külön köszöntötték az EIT részéről is. Ekkor látogattunk el először az *Open Innovation Houseba*, mely az EIT ICT Labs helsinki irodájának, a Nokia kutatóközpontjának, valamint az App Campusnak ad helyet. Az épületet másfél éve építették. Megtalálhatóak a legmodernebb technológiák, amikhez az EIT ICT Labs dolgozóinak, az EIT által támogatott startupok



2. ábra. Az Open Innovation House

dolgozóinak, valamint nekünk mesterképzéses hallgatóknak is van hozzáférésünk. Például korlátlanul rendelkezésünkre áll egy 3D nyomtató. Amint kiderült, hogy még a hétvégén is lehetőségünk lesz ezen a helyen tanulni, minden hallgató tudta, hogy nagyon sok időt fogunk itt tölteni.

Az orientációs hét után megkezdődött *tanítás*. Finnországban öt darab hat hetes tanulmányi időszakra van beosztva a tanév, melyek között egy hetes vizsgaidőszakok vannak. A tantárgyakat egy, másfél, vagy két ötéven keresztül tanítják. A tanulmányokban nagyon meghatározóak a projektfeladatok, melyeket általában csoportosan kell elkészíteni. A csoportokat kurzustól függően vagy az oktató tanár állítja össze, vagy a hallgatók szabadon választhatják ki a csoporttársaikat. Az értékelés általában csoportszinten történik. A szakirányomon az első tanévben nagyrészt üzleti tárgyakat hallgatunk, sőt az informatikai témájú tárgyaknál is nagy hangsúlyt helyez az egyetem a tananyag gyakorlati, üzleti életben való hasznosíthatóságára. Ennek érdekében ipari környezetből érkező vendégelőadásokat szerveznek szinte minden kurzuson, amik alapján első kézből hallhatjuk hogyan alkalmazzák az általunk hallottakat a való életben.

Az első két ötéven általam legkedveltebb tantárgyán az első projektünkön az *F-Secure* nevű vírusirtó gyártónak (Finnország egyik legnagyobb szoftvercége) dolgozhattunk. Egy új alkalmazásuk fejlesztésére kellett javaslatot tennünk felhasználók és potenciális felhasználók megkérdezésével. A végső prezentációnkon a cég felső vezetősége is részt vett. A második projektben egy kisebb cégnek kellett teljesen új üzleti modellt kitalálnunk. A harmadikban pedig egy Finnországban tartott

sívilágkupa futam közösségi médiai jelenlétét kellett elemeznünk. Mindhárom projekt eredményeit a csoportok prezentálták a cégeknek, akik hasznosnak találták az ötleteinket.

A félév során több fontos eseményen is részt vehettünk az EIT ICT Labs jóvoltából. Péntekenként úgynevezett *Lunch Talkokra* vagyunk hivatalosak, ahol az Open Innovation Houseban működő cégek, de esetenként külföldi vendégek kb. 40 perces előadásait hallgathatjuk meg egy finom ebéd kíséretében. Az egyik alkalommal *Corinna Cortes*, a Google New York-i kutatóközpontjának a vezetője volt a meghívott vendég.

Egy másik, a szívemnek különösen kedves eseménye a félévnek a Berlinben tartott *Kick-off* volt. Ezen az eseményen az EIT ICT Labs Master School összes elsőéves, nem kevés másodéves hallgatója, valamint a Master School vezető oktatói vettek részt. A Berliini Műszaki Egyetem adott otthont a három napos rendezvénynek. Érdekes előadásokat hallhattunk egyetemi oktatóktól, startupok vezetőitől, a Master School felső vezetésétől, valamint az EIT ICT Labs igazgatójától. A Kick-offnak nagyon fontos részét képezte még egy úgynevezett *Student's Challenge* is, melyben 25 csapatban kellett a világ jelenlegi öt kihívásokkal teli problémájára üzleti ötleteket kitalálnunk. Fontos megemlíteni, hogy nem a technológiai megvalósíthatóság volt a legfontosabb kritérium, hanem hogy tényleges problémára adjon megoldást.

A hét fős csapatokat úgy állították össze, hogy a csapattagok minél több egyetemről, minél különbözőbb szakirányokról érkezzenek. Az utolsó napon a csapatoknak prezentálniuk kellett az ötleteiket, először csak az öt azonos témán



3. ábra. Prezentáció közben

dolgozó csapattal, majd az elődöntőt megnyerő csapatok prezentálhattak a teljes hallgatóság előtt a döntőben is. Csapatomból nekem volt erre lehetőségem, és a döntőbe is jutottunk az Interneten keresztül történő virtuális kézfogást megvalósító platformunkkal. A Kick-offon nemcsak a szakmai eseményeké volt a főszerep. Az első este az egyetem aulájában egy álló fogadáson vettünk részt, ahol megismerkedhettünk egy másodévesek által fejlesztett programmal, melynek segítségével mi választhattuk ki mely zeneszámokat hallhassuk az este folyamán. A második este pedig egy gálavacsorára voltunk hivatalosak. Egy nagyon elegáns étteremben, old-timer autók között fogyaszthattunk el egy nagyon ízletes menüt, miközben ismét lehetőségünk nyílt a hallgatótársainkkal, valamint az oktatókkal való ismerkedésre. A három napon után fáradtan, élményekkel telve tértünk vissza Finnországba.

A következő fontos esemény a *Slush konferencia* volt. Ez egy Aaltos diákok által megálmodott esemény, mely az alapítása óta az északi országok legnagyobb startup konferenciájává nőtte ki magát. Több mint hét ezren vettek részt a két napos eseményen, mely nem jöhetett volna létre a több mint hétszáz önkéntes segítségével. A konferencián startupokat lehetett megismerni, valamint figyelemreméltó előadásokat lehetett meghallgatni. Az esemény fontosságát jelzi, hogy Finnország miniszterelnöke tartotta a megnyitóbeszédet. Számomra az észt köztársasági elnök előadása volt a legérdekesebb, aki mint kiderült programozni is tud. Előadásában elmondta, hogy Észtországban minden állampolgár kap egy olyan személyi számot, mely minden szempontból képes az adott ember azonosítására és aláírás helyett is használható. Így lehetséges, hogy az észtek 25 százaléka az Interneten szavazhatott és Észtországban senkinek nem tart három percnél tovább az adóbevallás kitöltése.

Amint a fenti sorokból is látható nagyon élvezem finnországi tanulmányaimat és a számomra legmegfelelőbb választásnak találok az EIT ICT Labs Master Schoolt. Mindenkinek ajánlani tudom, hogy jelentkezzen, aki nyitott új kultúrák megismerésére, szívesen tanulna külföldön, és szeretne a legmodernebb informatikai tudás megszerzésén túl üzleti tudással is gazdagodni. Jelen pillanatban mi vagyunk a második évfolyam a képzés történetében, ezért még sok döntés meghozatala vár a Master School vezetésére a képzés még színvonalasabbá tételének érdekében, de véleményem szerint nagyon jó úton haladnak.

Funkcionális programozásban használt lencsék vizsgálata Agdában

Manninger Mátyás

Eötvös József Collegium*

`manninger.matyas@gmail.com`

1. Bevezetés

A lencsék, más néven funkcionális referenciák, összetett adatszerkezetek komponálható szerkesztésére. A komponálhatóság abban nyilvánul meg, hogy ha egy adatszerkezet tartalmaz egy másik adatszerkezetet, úgy a két adatszerkezethez tartozó lencsék kompozíciója szintén egy lencse, mely az összetett adatszerkezetet szerkeszti. Például, ha adott egy lencse, ami egy lista első elemét szerkeszti, és adott egy lencse, ami egy pár második elemét szerkeszti, akkor a két lencse kompozíciója egy olyan listát szerkeszt, melynek az első eleme egy pár, és ennek a listának az első elemének második elemét szerkeszti.

Így a lencsék, mint getter és setter függvények képzelhetők el, vagyis egy lencse nem más, mint egy adatszerkezet getter és setter függvénye egyben. Ennél azonban összetettebb egy lencse. A két függvény összhangja, és a komponálhatóság érdekében két függvény akkor lesz lencse, ha ezek teljesítenek három alapvető tulajdonságot. Az azonban bizonyításra szorul, hogy a lencsekompozíció művelete valóban megtartja ezeket a tulajdonságokat.

A bizonyítások elvégzéséhez először szükség van a maguknak a lencséknek, majd az állítások formalizálására. A későbbi használhatóság

*2012–

érdekében ezt Agdában tettem meg. Az Agda egy funkcionális programozási nyelv és matematikai tételbizonyító rendszer. Használatának előnyei, hogy, a Curry–Howard i zomorfizmust alapul véve, a bizonyítások helyességét és az állítások jólformáltságát a számítógép ellenőrzi. Ennek a lényege, hogy a típusok megfeleltethetők matematikai állításoknak és a bizonyítások pedig a típusértékek.

2. Lencsék matematikai definíciója

Több megközelítés létezik a lencsék definíciója kapcsán (például kategóriaelméleti megközelítés [3]). Egy matematikai definíciója lehet a lencséknek olyan függvénpárok, melyek teljesítik a korábban már említett, összhangot biztosító 3 tulajdonságot. Így ha adott két függvény:

$$get : A \rightarrow B,$$

$$set : (B \times A) \rightarrow A.$$

ahol A és B két tetszőleges halmaz, akkor a (get, set) függvénpár lencse, ha teljesítik a lencsetörvényeket. Legyen tehát a lencsék definíciója:

A és B tetszőleges halmazok, $f : A \rightarrow B$, $g : (B \times A) \rightarrow A$, ekkor az (f, g) függvénpár lencse, ha

1.

$$\forall a \in A : g(f(a), a) = a,$$

2.

$$\forall a \in A, b \in B : f(g(b, a)) = b,$$

3.

$$\forall a \in A, b, b' \in B : g(b, g(b', a)) = g(b, a).$$

A definícióban szereplő f függvény a getter, és a g függvény a setter. A getter függvény egyetlen paramétere az adat amit szerkeszteni szeretnénk, a setter paraméterei pedig egy érték, aminek a függvényében szeretnénk szerkeszteni a második paramétert, mint korábbi állapotot. Ilyen szemlélettel a három elvárás képletes jelentése a következő:

Az első szabály azt mondja ki, hogy ha egy adatot a saját részletének függvényében szerkesztünk, akkor nem változik az adat.

A második szabály átfogalmazása lehet, hogy ha egy adatot szerkesztünk valaminek a függvényében, akkor ezt a getter függvénnyel vizsgálva az lesz az eredmény, aminek a függvényében szerkesztettünk.

A harmadik szabály pedig azt írja le, hogy egy adatot kétszer szerkesztve, ugyan az az eredmény, mint amikor csak a második szerkesztés történik meg.

A három szabályt szemlélve érthető a lencse elnevezés. A lencse nem egyszerűen két függvény párja. A lencsetörvények miatt a két függvény úgy viselkedik mint egy fizikai lencse amin keresztül szemléljük és szerkesztjük az adatot.

3. Az Agda programozási nyelvről

Az Agda egy függő típusos funkcionális programozási nyelv és matematikai tételbizonyító rendszer [1]. Lencsék tulajdonságainak bizonyítására a Curry-Howard izomorfizmus teszi lehetővé, így a bizonyítások ezen alapszanak. Lényegük, hogy az állítás egy halmaz definíálása, az állítás bizonyítása pedig ennek a halmaznak egy (az egyetlen) eleme. Agdában minden program egy modul ezért a következő sorral indul:

```
module cikk where
```

A modulon belül a halmazdefiníálásra a következő szintaxissal van lehetőség:

```
data Bool : Set where
  true  : Bool
  false : Bool
```

Ebben a definícióban a **data**, **Set** valamint a **where** kulcsszavak, a **Bool** lesz a halmaz vagy típus neve, míg a **true** és **false** konstruktorok, jelen esetben paraméter nélkül, tehát úgy is tekinthetünk rájuk, mint a halmaz elemeire.

Halmaz konstruktorai lehetnek függvények is, ezt mutatja a természetes számok halmazának definíciója:

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

Itt a **zero** egy eleme a halmaznak és ha van egy eleme a halmaznak, ami legyen **n**, akkor a **suc n** is eleme a halmaznak. Ezek úgy hangzanak, mint a Peano-axiómák, és valóban, mivel nincs is más eleme ennek a halmaznak, így ez a definiált halmaz izomorf a természetes számok halmazával.

A lencsékkel kapcsolatos tulajdonságokra vonatkozó állításokban fontos szerepet játszik az egyenlőség. Ennek típusa az Agdában:

$_ \equiv _ : \{A : \text{Set}\} \rightarrow A \rightarrow A \rightarrow \text{Set}$

Vagyis az egyenlőség ebben a formában nem más, mint egy függvény, ami két paramétert vár, és ezekhez rendel egy halmazt. (Az $\{A : \text{Set}\}$ azt jelzi, hogy az **A** halmaz egy rejtett paramétere a függvénynek, ezt nem kell a függvényhíváskor megadni, azt a fordító igyekszik kitalálni. Ezek alapján az egyenlőséggel megfogalmazott állítás kimondásának és bizonyításának az elve tehát, hogy két elemhez az egyenlőséggel rendelünk egy halmazt, vagyis egy típust. Ez lesz maga az állítás és ebben az esetben akkor bizonyítottuk az egyenlőséget, ha ennek a halmaznak meg tudjuk adni egy elemét, vagyis definiálunk egy konstanst ami $a \equiv b$ típusú, majd megadjuk az értékét. Ezzel bizonyítottuk is az állítást.

Az egyenlőség bizonyításához érdemes használni az egyenlőség jól ismert tulajdonságait (nevezetesen azt, hogy ekvivalencia reláció). A reflexivitás típusa például:

$\text{refl} : \{A : \text{Set}\} \{x : A\} \rightarrow x \equiv x$

Ennek a függvénynek csak rejtett paraméterei vannak, a **refl** tehát nem más, mint egy bizonyítás arra, hogy valami egyenlő önmagával, azt pedig, hogy mi az a valami, azt a fordító igyekszik kitalálni.

A szimmetria típusa Agdában:

$\text{sym} : \{A : \text{Set}\} \{a\ b : A\} \rightarrow a \equiv b \rightarrow b \equiv a$

Ez egy olyan függvény, ami egy bizonyítást vár, hogy egy tetszőleges **a** egyenlő egy tetszőleges **b**-vel, és visszaadja annak a bizonyítását, hogy **b** egyenlő az **a**-val.

A tranzitivitás is definiálva van, a következő típussal:

$\text{trans} : \{A : \text{Set}\} \{a\ b\ c : A\} \rightarrow a \equiv b \rightarrow b \equiv c \rightarrow a \equiv c$

Fontos tulajdonság ezen kívül még a kongruencia, vagyis ha két egyenlő érték a paramétere ugyan annak a függvénynek, akkor a függvény

helyettesítési értékei is egyenlők lesznek, vagyis, hogy $\forall a, b \in A, f : A \rightarrow B : a = b \rightarrow f(a) = f(b)$. Ez a tulajdonság a következő képpen képzelhető el:

```
cong : {A B : Set} {m n : A} → (f : A → B) → m ≡ n →
                                         f m ≡ f n
```

Ezek a tulajdonságok, csak úgy mint maga az egyenlőség, importálhatók a standard könyvtárból:

```
open import Relation.Binary.PropositionalEquality
using (_≡_; refl; trans; cong; sym)
```

A bizonyítások szépsége érdekében érdemes definiálni még a bizonyítás kedetét jelző `begin`-t valamint a bizonyítások végét jelző `■` karaktert, ami nem más, mint a reflexivitás szinonímája:

```
begin_ : {A : Set} {a b : A} → a ≡ b → a ≡ b
begin x = x
```

```
infix 1 begin_
```

```
_■ : ∀{A : Set} (x : A) → x ≡ x
x ■ = refl
```

```
infix 2 _■
```

A definíciók után az operátorok kötési erőssége definiálható.

Ismét csak az olvashatóság érdekében hasznos bevezetni a következő operátort is, mely gyakorlatilag a tranzitivitás átnevezésének is tekinthető:

```
_≡⟨_⟩_ : {A : Set} (x : A) {y z : A} → x ≡ y → y ≡ z →
                                         x ≡ z
x ≡⟨ e ⟩ f = trans e f
```

```
infixr 2 _≡⟨_⟩_
```

Ezek használatával annak a bizonyítása, hogy $a \equiv b$ a következő sémára épül:

```
x : a ≡ b
x =
  begin
```

```

a
≡ ⟨ lemma1 ⟩
c
≡ ⟨ lemma2 ⟩
d
...
≡ ⟨ lemman ⟩
b
■

```

Ebben a lemmák részbizonyítások, a bizonyítás vége az, hogy $b \equiv b$, és innentől a lemmákat valamint a tranzitivitást alkalmazva adódik, hogy $a \equiv b$. (A példában például `lemma2` annak a bizonyítása, hogy $c \equiv d$.)

4. Lencsék definiálása Agdában

Az Agda előnyei közé tartozik, hogy mivel az állítások típusok, a bizonyítások pedig típus értékek, így egy rekord definíciójánál felvehetünk olyan típusú elemeket, mint az elvárt tulajdonságra vonatkozó állítás, és így amikor egy konkrét példányát definiáljuk a rekordnak, akkor a megfelelő állítások bizonyítását is meg kell adni. Így például a lencse típust rekordként definiálva biztosak lehetünk, hogy minden ilyen típusú lencse teljesíti a lencsetörvényeket. Legyen tehát a lencse típus definíciója:

```

record Lens (A B : Set) : Set where
  constructor lens
  field
    get  : A → B
    set  : B → A → A
    getSet : {a : A} → set (get a) a ≡ a
    setGet : ∀ a b → get (set b a) ≡ b
    setSet : ∀ {a b1 b2} → set b1 (set b2 a) ≡ set b1 a

open Lens

```

A rekord két paramétere az a két típus amivel a lencse dolgozik, konkrét lencse pedig a `lens` kulcsszóval, majd a mezők értékeinek

felsorolásával hozható létre. Az `open Lens` biztosítja, hogy a rekord mezői nem privát adattagok, a rekordon kívül is elérhetők. Érdekes még megjegyezni, hogy egyes szakirodalmak más elnevezéseket használnak, így használatos a `set` helyett a `put` [2], vagy akár a `putback` [4] elnevezés is.

5. Lencsék kompozíciója

Lencsék kompozíciójához szükséges a függvénykompozíció definiálása:

```
comp : {A B C : Set} → (B → C) → (A → B) → (A → C)
comp g f = λ z → g (f z)
```

A definícióban a `\lambda` névtelenfüggvényt definiál, ami a paraméterre alkalmazza a `comp` függvény két paramétereként kapott függvényt.

Érdekes a függvénydefiníciókat és bizonyításokat a kompozícióval kapcsolatban egy új modulba rendezni, valamint priváttá tenni, így majd csak maga a művelet lesz használható a modulon kívül. Erre szolgál a következő névtelen modul:

```
module _ A B C : Set (l : Lens B C) (k : Lens A B) where

  private
```

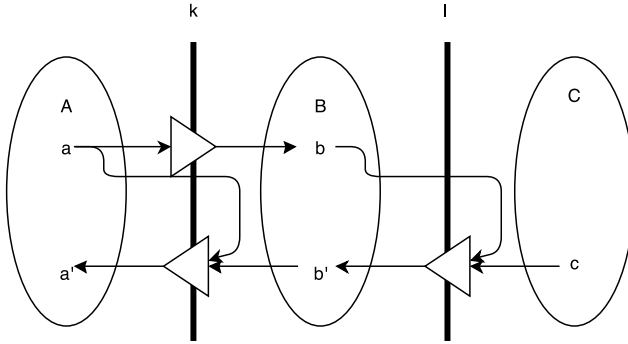
A modul három rejtett paramétere, valamint a két lencse mint paraméter az összes modulon belüli függvény paramétere lesz.

Lencsék kompozíciójának a `get` függvénye nem más mint a két lencse `get` függvényének a kompozíciója:

```
get' : A → C
get' = comp (get l) (get k)
```

Itt a `get l` az `l` lencse `get` függvénye.

A kompozíció `set` függvényének megértését segíti az 1. ábra [4]: Látszik, hogy az új érték kiszámításához (`a'`) szükségünk van a `b'` érték kiszámítására, amihez a `b` értéket a `get k` a szolgáltatja, vagyis a `k`



1. ábra. A kompozíció **set** függvénye

lencse **get** függvénye. Az **l** lencse **set** függvénye a **c** és **b** értékekből kiszámolja a szükséges **b'** értéket, majd a **k** lencse **set** függvénye az **a** és **b'** függvényében pedig visszaadja a kívánt **a'** értéket. Ennek a megvalósítás Agdában:

```
set' : C → A → A
set' c a = set k (set l c (get k a)) a
```

Ezekről a függvényekről kell belátni, hogy rendelkeznek a megfelelő tulajdonságokkal a kompozíció definiálásához. A **getSet** bizonyítása a következő képpen végezhető:

```
getSet' : {a : A} → set' (get' a) a ≡ a
getSet' {a} =
  begin
    set k (set l (get l (get k a)) (get k a)) a
  ≡ ⟨ cong (λ x → set k x a) (getSet l) ⟩
    set k (get k a) a
  ≡ ⟨ getSet k ⟩
    a
```

■

A bizonyításnál szükséges elnevezni a rejtett paraméter, hogy lehessen rá a bizonyítás közben hivatkozni. A bizonyításban az egyenlőség kifejtett bal oldalából indulva a tranzitivitást és bizonyításokat használva kapjuk a bizonyítást. Használhatók korábbi bizonyítások, így mivel a lencsék rekordjában a megfelelő bizonyítások is benne vannak, használhatók a lencsék tulajdonságainak bizonyításai, mint például a `getSet 1` vagy a `getSet k`. Az első lemma pedig az egyenlőség kongruencia tulajdonságát használja ki, vagyis a `getSet 1` tulajdonság egyenlőségének mindkét oldalát “csomagolja be” a névtelen függvény segítségével.

Hasonlóan bizonyítható a `setGet` valamint a `setSet` tulajdonság is lencsekompozíció esetén:

```

setGet' : ∀ a b → get' (set' b a) ≡ b
setGet' a b =
  begin
    get 1 (get k (set k (set 1 b (get k a)) a))
  ≡ ⟨ cong (get 1) (setGet k _ _) ⟩
    get 1 (set 1 b (get k a))
  ≡ ⟨ setGet 1 _ _ ⟩
    b
  ■

setSet' : ∀ {a c c'} →
  set' c (set' c' a) ≡ set' c a
setSet' {a} {c} {c'} =
  begin
    set k (set 1 c (get k (set k (set 1 c'
      (get k a)) a)))
    (set k (set 1 c' (get k a)) a)
  ≡ ⟨ cong (λ x → set k (set 1 c x)
    (set k (set 1 c' (get k a)) a))
    (setGet k _ _) ⟩
    set k (set 1 c (set 1 c' (get k a)))
    (set k (set 1 c' (get k a)) a)
  ≡ ⟨ cong (λ x → set k (x) (set k (set 1 c'
    (get k a)) a))
    (setSet 1) ⟩
    set k (set 1 c (get k a)) (set k (set 1 c'

```

$$\begin{aligned} & \text{(get k a)) a)} \\ \equiv & \langle \text{setSet k} \rangle \\ & \text{set k (set l c (get k a)) a} \\ \blacksquare \end{aligned}$$

A bizonyítások után már csak a lencsék rekordjának a konstruktorát kell használni a megfelelő függvény definiálásához, ami nem lesz privát tagja a modulnak, így később is használható:

```
_ o _ : Lens A C
_ o _ = lens get' set' getSet' setGet' setSet'
```

Ezzel beláttuk, hogy két lencse kompozíciója is rendelkezik a kiindulási lencsék tulajdonságaival.

6. Összefoglalás

A tanulmányban láttuk, hogyan bizonyíthatunk egyenlőséget az Agda segítségével. Definiáltuk a lencsét, mint különleges függvényt, és kihasználva az Agda sajátosságait garantáltuk, hogy minden definiált lencse teljesíti a három tulajdonságot, amit elvárunk tőlük. Végül definiáltuk két lencse kompozícióját mint lencsét, ezzel belátva, hogy ez is megőrzi mindhárom tulajdonságot.

Hivatkozások

- [1] A. Bove, P. Dybjer, U. Norell, A brief overview of agda - a functional language with dependent types, *TPHOLs*, 2009, pp. 73–78.
- [2] J. Gibbons, M. Johnson, Relating algebraic and coalgebraic descriptions of lenses, *ECEASST*, 49, 2012.
- [3] R. O'Connor, Functor is to lens as applicative is to biplate: Introducing multiplate, *CoRR*, abs/1103.2841, 2011.
- [4] B. C. Pierce, The weird world of bi-directional programming, ETAPS invited talk, March 2006.

Önszervező kritikus rendszerek az idegtudományban

Nádor István*

Eötvös József Collegium**

i.a.nador@vu.nl

Az idegtudomány területén az önszerveződő kritikus rendszerek az utóbbi évtizedekben komoly érdeklődésre tartanak számot, hiszen az agy, mint rendkívül komplex rendszer, számtalan módon mutatott kritikusságra utaló jeleket. Az idegi lavinák skálafüggetlensége, illetve az $1/f$ zaj gyakori előfordulása mind erre szolgáltatnak példát. A kritikusság gazdagítja az agy dinamikáját, és lehetővé teszi, hogy nagy érzékenységgel reagáljon az őt ért hatásokra. Ezen a főként ismeretterjesztő cikk az idegtudományi alkalmazásokra fókuszálva bemutatja az önszervező kritikus rendszerek eszméjét, és ismerteti egy neurális modellt, a Critical Oscillations modellt, ami egyedülálló módon mind a neurális lavinák, mind a neurális rezgések hosszútávú korrelációja terén kritikus, ezáltal segíti ezen két jelenség kapcsolatának közelebbi megértését.

1. Hatványtörvény

Két mennyiség hatványtörvény szerint viselkedik, amennyiben a kettejük közti kapcsolat hatványfüggvénnyel arányosan leírható, azaz valamely α , β párosra $y = f(x) = \beta \frac{1}{x^\alpha}$. Ezen összefüggések kettős logaritmikus skálán szemlélve lineárisak, és a skálától függetlenek, hiszen

*Center for Neurogenomics and Cognitive Research, Amszterdam, Hollandia;
Vrije Universiteit, Amsterdam, Hollandia

**2009–

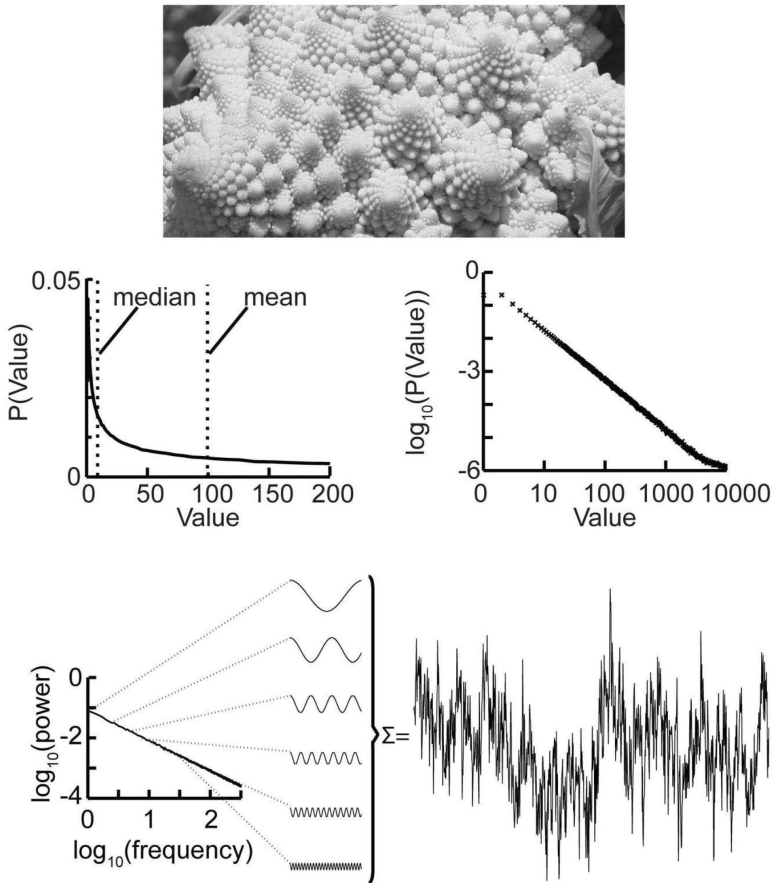
$$f(c \cdot x) = \beta \frac{1}{(c \cdot x)^\alpha} = \frac{1}{c^\alpha} \cdot \beta \frac{1}{x^\alpha} = \frac{1}{c^\alpha} \cdot f(x)$$

amely szintén a hatványtörvénynek engedelmeskedő összefüggés. A *skála-invariancia* következménye, hogy egy hatványtörvény jelenség mérésekor a kapott eloszlás független a vizsgálat felbontásától, továbbá az átlag és medián érték nem kifejező mértékek, hiszen nincsen a jelenségre jellemző skála. Ezért a hatványtörvényt az α exponenssel szokás jellemezni. Eme fraktáltulajdonságot szemlélteti az 1 ábra.

2. Önszerveződő kritikus rendszerek

Bak et al. vetette fel először az önszerveződő kritikus rendszereket (Self-Organized Criticality, röviden SOC), mint a természetben oly gyakran előforduló $1/f$, vagyis hatványtörvény zaj egy lehetséges magyarázatát [1]. Az elgondolást pontos definíció helyett egy modellel szemléltették. A homokszem modell imitálja homokszemek mozgását ahogy azok lassú ütemben a földre hullanak, és ennél nagyságrendekkel gyorsabban szétterülnek: kezdetben apránként kupacot alkotnak, majd ahogy annak lejtése növekszik, a homokszemek időnként egymást arrébb lökve szétterülnek, míg végül a rendszer el nem ér az úgy nevezett *kritikus állapotba*, ahol a kupac lejtése többé-kevésbé stabil marad. Ebben a törékeny egyensúlyi állapotban (pontosabban azok halmazában) azon *lavínák* mérete és élettartama, amik egyetlen homokszem hatására keletkeznek, $\propto \frac{1}{f^\alpha}$ eloszlású. Ebből az is levezethető, hogy egy adott pillanatban mozgásban lévő homokszemek számának Fourier spektruma szintén hatvány függvény, egészen a leghosszabb élettartamú lavina nagyságrendjéig bezárólag. Cikkükben Bak et al. a folyamatot a következőképpen szimulálták: egy 50×50 -es kezdetben üres négyzetrács véletlenszerűen kiválasztott pontjára homokszemet ejtünk, és amennyiben valamely rácsponton a homokszemek száma eléri a 4-et, az ott található szemeket szétosztatjuk annak négy szomszédjára, ezt addig ismételve, amíg minden rácsponton a szemek száma kisebb nem lesz, mint 4.

A jelenség az önszerveződő kritikus rendszerek nevet kapta, mivel hasonlóan a termodinamikai kritikus pont környezetében tapasztaltak-



1. ábra. A hatványtörvény skála-invarianciája. Fent: a pagoda karfiol (*Romanesco broccoli*), és a rajta lévő virágok méretének egy lehetséges eloszlása. Lent: skála független idősor és frekvencia spektruma. Forrás: [3].

kal, az $\frac{1}{f^\alpha}$ függvény lassú lecsengése miatt a kritikus állapotban csekély, lokális perturbáció akár az egész rendszeren végigterjedhet. Ezzel szemben a szuper-, vagy szubkritikus állapotokban (ebben az esetben túl telített, illetve ritka kupacok esetén) a behatás exponenciálisan lecsengve

terjed. A rendszer önszerveződő is, hiszen minimális behatásra, belső kölcsönhatásokon keresztül szerveződik metastabil egyensúlyi állapotba.

Flyvbjerg hozakodott elő a minimális önszerveződő kritikus rendszerrel, a flipper modellel [2]. Ahogyan a neve is mutatja, Flyvbjerg a flipper gép analógiáját használta a folyamat leírására: egy golyót a rendszerbe löve az egyenletes eloszlás szerint vagy azonnal elhagyja rendszert az M rés valamelyikén, vagy megreked az N rekesz valamelyikében. Ha az egyik rekeszben a golyók száma eléri a kettőt, akkor az kienged, és a golyók egyesével újra kilövének. Amennyiben $M/\sqrt{N} \rightarrow 0$, az $N, M \rightarrow \infty$ határértékben a rendszer önszerveződő kritikus. Ezen rendszer minden állapota leírható mindössze az (N_0, n) párossal, ahol N_0 az üres rekeszek, n a rendszerben lévő golyók számát jelöli.

A flipper modell segítségével Flyvbjerg a következőképpen definiálta az önszerveződő kritikus rendszereket: a SOC egy *kívülről hajtott, disszipatív* rendszer, aminek része egy

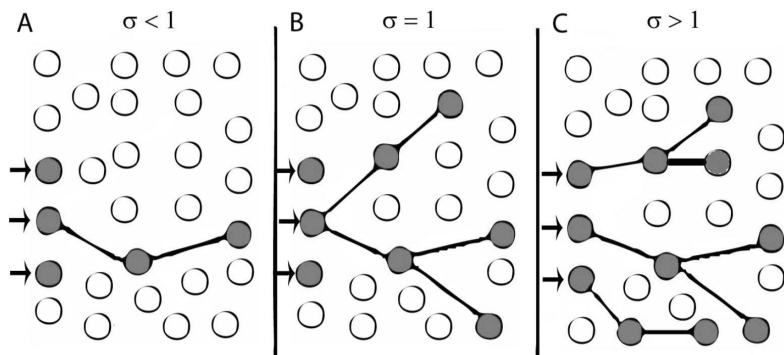
1. *közeg*, ami lehetővé teszi
2. *perturbációk* terjedését, amik
3. úgy *változtatják* a közeget, hogy azok végül
4. *kritikusak* lesznek, és a közeg – statisztikailag –
5. *nem változik tovább*.

3. Elágazó folyamatok

Az elágazó folyamatok segítségével könnyebben megérthetjük a lavinák terjedését. Az elágazó folyamat egy előrecsatolt háló, amiben minden egyes aktivált csúcs várhatóan σ utód csúcsot aktivál. A folyamat markovi, hiszen az állapota minden időpillanatban leírható pusztán az aktivált csúcsok számával. A lavina méretét az aktivált csúcsok számaként, míg élettartamát az aktivált részfa mélységeként definiálhatjuk. Amennyiben a rendszer korrelálatlan, azaz kezdetben csak egy csúcs aktivált és a csúcsok nem hatnak kölcsön egymással¹, akkor a lavinák mérete aszimptotikusan akkor és csak akkor hatványtörvény eloszlású, ha $\sigma = 1$ [7], egyébként exponenciális eloszlású. Az is belátható, hogy

¹A teljes definícióért lásd [4]

$\sigma = 1$ esetén a lavinák élettartama szintén hatványtörvény eloszlású [4]. Bár az önszerveződő kritikus rendszerekről nem mondhatjuk el, hogy korrelálatlanok, mégis a σ kritikus értéke jól szemlélteti a kritikus állapotban környezetében végbemenő fázistranszformációt.



2. ábra. A σ várható értéktől függően az elágazási folyamat szubkritikus, kritikus, illetve szuperkritikus. Forrás: [9].

4. Detrendált fluktuáció analízis

A detrendált fluktuáció analízis (DFA) idősorok önhasonlóságát értékelő statisztikai mennyiség [8]. Az alternatív módszerekkel szemben, mint például autokorrelációs függvény, nem stacionárius (időben változó dinamikájú²) függvények esetén is megbízhatóan alkalmazható. A DFA analízis első lépésben kiszámítja a bemeneti jel prefix összegét, majd azt egyenlő részekre (ablakokra) osztva átlagolja az egyes jelrészletek szórását, végül $\beta \cdot f^\alpha$ alakú függvénnyel közelíti az ablakok mérete és az átlagolt szórás közti összefüggést. A prefix összeggel megkapjuk a jel akkumulált értékét, hasonlóan ahogy a véletlen bolyongás esetén az előző döntések összege megadja az ügynök aktuális pozícióját. A módszer stabilitását növeli, hogy az egyes ablakokban a szórás kiszámítása előtt a jelrészletekből eltávolítja a lineáris trendeket, így az ablak méreténél nagyobb skálán végbemenő változások nem befolyásolják

²A pontos definícióért lásd [6]

a szórást. Például fehérzaj jel esetén az ablakmérettel hatványozottan nő az átlagolt szórás, $\alpha = 0.5$ exponenssel. Az eredményül kapott α exponenstől függően a folyamat a következőképpen kategorizálható:

- $0 < \alpha < 0.5$ esetben a folyamat önmagával negatívan korrelál, emlékezik és stacionárius
- $0.5 < \alpha < 1$ esetben a folyamat önmagával pozitívan korrelál, emlékezik és stacionárius
- $\alpha = 0.5$ esetben a folyamat nem megkülönböztethető egy nem emlékező, véletlen jelsorozattól
- $1 < \alpha < 2$ esetben a folyamat nem stacionárius

Emlékezés alatt itt a jel előző értékeitől való statisztikai függést értjük, ám a rendszer továbbra is lehet stacionárius, amennyiben ezen eloszlás állandó. Érdekes megjegyezni, $0 < \alpha < 0.5$ esetén az eredeti jel hatvány-törvény zaj, azaz Fourier spektruma $\propto \frac{1}{f^\alpha}$.

Az idegtudomány területén a DFA analízis gyakran használt eszköz az oszcilláló idegi jelek hosszútávú korrelációjának feltérképezésére. Azt már az előbbiekben láttuk, hogy hatvány-törvény lavina eloszlásból következik a jel korrelációja, de csak a leghosszabb élettartamú lavináig bezárólag. Célszerűen az ennél hosszabb távú korreláció kimutatásához a lassabb ütemű ingadozásokat érdemes vizsgálni. Linkenkaer-Hansen et al. pihenő emberek magnetoencefalográfia (röviden MEG, az egy elektromos aktivitása által gerjesztett mágneses teret mérő készülék) jeleinek amplitúdómodulációját vizsgálta az alfa³ frekvencia sávban [5]. $A \sim 5 - 300$ másodperc széles ablakokra kapott DFA exponens értéke $\alpha \approx 0,71 \pm 0,06$, ami emlékező folyamatra utal.

5. Critical Oscilations (CROS) modell

Az önszerveződő kritikus rendszerek és a hosszan tartó korreláció kapcsolata körül jelenleg is sok kérdés tisztázatlan. Ugyan a lavinák $1/f^\alpha$ élettartam eloszlása okozhat $1/f^\alpha$ alakú frekvencia spektrumot,

³Jellemzően az alfa sávban ($\sim 8 - 13$ Hz) találhatók a legnagyobb energiájú agyhullámok.

jellemzően az $1/f^\alpha$ skála csak a leghosszabb lavina élettartamának megfelelő frekvenciáig tart ki. Ezen felül oszcilláló rendszerekben, mint amilyen az emberi agy is, a hosszútávú korreláció a frekvencia modulációban is jelen lehet. A CROS neurális háló modell [10] egyedülálló abban a tekintetben, hogy egyrészt az agyhoz kvalitatíve és kvantitatíve is hasonló oszcillációkat produkál, másrészt egyszerre több szinten is mutat kritikalitásra utaló jeleket: rövidtávon a homokszem modellben látottakhoz hasonlóan $1/f^\alpha$ lavina méret és élettartam eloszlást, hosszútávon az alfa hullámok amplitúdómodulációjában pozitív korrelációt.

5.1. Topológia

A rendszert alkotó neuronok egy 50×50 -es rácshálón helyezkednek el. Ezen neuronok 1 : 3 arányban gátlók, illetve serkentők, amik pusztán az őket összekötő szinapszisok súlyában térnek el: a gátló neuronokból induló szinapszisokhoz negatív, a serkentőkhöz pozitív súly van rendelve. Minden neuron – az agyi kapcsolatok nagymértékű lokalizációját szem előtt tartva [11] – csak a 7×7 -es szomszédságában találhatókhöz kapcsolódik a távolsággal exponenciálisan csökkenő valószínűséggel. A gátló, illetve serkentő neuron típusok várható kimenő szinapszisainak száma a modell változtatható paraméterei.

5.2. Neuron modell

A teljes neuron modell egy szinaptikus és egy ingerlési modellből tevődik össze. A szinaptikus modellt az alábbi egyenletek írják le:

$$I_i = I_i + \sum_j W_{ji} S_j \quad (1)$$

$$\tau_I \frac{dI_i}{dt} = I_0 - I_i \quad (2)$$

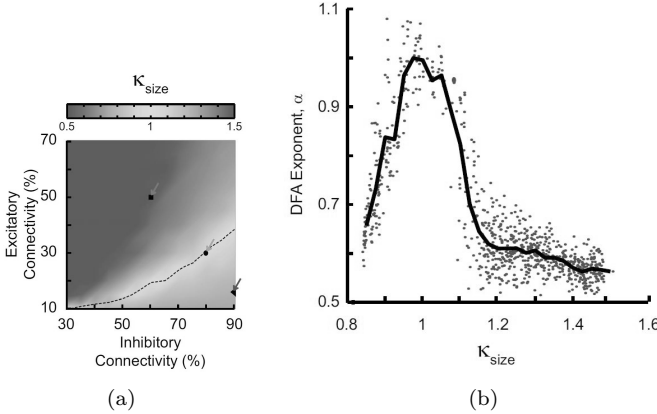
$S_j = 1$, ha az adott időpillanatban ingerület érkezik a j . neurontól, egyébként $S_j = 0$. W_{ji} a pre- és a posztzinaptikus neuron típusától függ: $W_{SG} = 0,011$, $W_{SS} = 0,02$, $W_{GS} = -2$ és $W_{GG} = -2$. A második egyenlet az I_i érték exponenciális lecsengésért felelős, és $I_0 = 0$, $\tau_I = 9$

ezredmásodperc. Az ingerület átadás P_S valószínűsége ezek után a következőleg módosul:

$$P_{Si} = P_{Si} + I_i \quad (3)$$

$$\tau_P \frac{dP_{Si}}{dt} = P_0 - P_{Si} \quad (4)$$

ahol a serkentő, illetve gátló neuronok esetén rendre $\tau_P = 6$, illetve $\tau_P = 12$ ezredmásodperc, és $P_0 = 0,000001$, illetve $P_0 = 0$. Az alacsony P_0 érték direkt megfelelője a homokszem modellben a leejtett szemek folyamatos, ám csekély behatása.



3. ábra. (a): a serkentő és gátló kapcsolatok egyensúlyától függően a lavinák szub-, szuper-, illetve kritikusak. A κ mennyiség egy minta hatvány eloszlástól való távolsággal arányos. $\kappa < 1$, $\kappa \sim 1$, $\kappa > 1$ rendre szub-, szuper-, illetve kritikus állapotot jelez. (b): Az α tartományú rezgésekben pontosan akkor figyelhetünk meg pozitív korrelációt (azaz az amplitúdómoduláció DFA exponense ~ 1), amikor a lavinák is közel kritikusak. Forrás: [10].

5.3. Kritikusság a CROS modellben

A kapcsolódási valószínűségektől függően a rendszer jól elkülöníthetően a szubkritikus, kritikus, illetve szuperkritikus állapotban van.

Továbbá a neuronháló pontosan akkor produkál hatvány-törvény lavi-
nákat, amikor az α sáv modulációja pozitív korrelációt mutat³.

6. Összefoglalás

A kritikusság előnyei intuitívek: a kritikus állapotban az agy könnyedén válthat két üzemmód között, és érzékenyen reagálhat impulzusok széles skálájára. Mindezek ellenére még továbbra sem teljesen tisztázott, hogy az evolúció és az egyedfejlődés miért alakította az agyat (közel) kritikussá, és milyen körülmények között lesz inkább szub-, illetve szuperkritikus. Továbbá az agy felépítése életünk során folyamatosan formálódik: a hebbi elveknek megfelelően a neurális kapcsolatok gyengülnek és erősödnek, így az agy képes tanulni, mégis a kritikusságát fenntartja, ezzel szemben a CROS modellben jelenleg a neuronok közötti súlyok konstansak. Tanulási mechanizmusok beépítésével remélhetőleg közelebbről megismerhetjük a kritikusság és a neuroplaszticitás összefüggéseit, és a neuronháló képes lesz önszerveződésre. Végül a modell jelenlegi topológiája csak egy szűk halmaza az összes lehetségesnek, és továbbra is ismeretlen, hogy annak mely aspektusai szükségesek rövid, illetve hosszútávú korreláció létrejöttéhez. A Center for Neurogenomics and Cognitive Research-ben töltött szakmai gyakorlatom során reményeim szerint közelebb kerülök ezen kérdések megválaszolásához.

Hivatkozások

- [1] P. Bak, C. Tang, K. Wiesenfeld, Self-organized criticality: An explanation of the $1/f$ noise, *Phys. Rev. Lett.*, 59 (1987), pp. 381–384.
- [2] H. Flyvbjerg, Self-organized critical pinball machine, *Physica A: Statistical Mechanics and its Applications*, 340(4) (2004), pp. 552–558.
- [3] R. Hardstone, S-S. Poil, G. Schiavone, R. Jansen, V. V. Nikulin, H. D. Mansvelder, K. Linkenkaer-Hansen, Detrended fluctuation analysis: A scale-free view on neuronal oscillations, *Frontiers in Physiology*, 3(450) (2012).

-
- [4] T. Harris, *The Theory of Branching Processes*, Die Grundlehren der Mathematischen Wissenschaften in Einzeldarstellungen, Springer-Verlag, 1963.
 - [5] K. Linkenkaer-Hansen, V. V. Nikouline, J. M. Palva, R. J. Ilmoniemi, Long-range temporal correlations and scaling behavior in human brain oscillations, *J. Neurosci*, 21 (2001), pp. 1370–1377.
 - [6] B. B. Mandelbrot, *The fractal geometry of nature*, Freeman, 1982.
 - [7] R. Otter, The multiplicative process, *The Annals of Mathematical Statistics*, 20(2) (1949), pp. 206–224.
 - [8] C-K. Peng, S. V. Buldyrev, S. Havlin, M. Simons, H. E. Stanley, A. L. Goldberger, Mosaic organization of dna nucleotides, *Phys. Rev. E.*, 49(2) (1994), pp. 1685–1689.
 - [9] S. S. Poil, Temporal correlations and criticality in models of neuronal net-works, Master’s thesis, Niels Bohr Institute, University of Copenhagen, Center for Neurogenomics and Cognitive Research (CNCR), Vrije Universiteit Amsterdam, 2007.
 - [10] S. S. Poil, R. Hardstone, H. D. Mansvelder, K. Linkenkaer-Hansen, Critical-State Dynamics of Avalanches and Oscillations Jointly Emerge from Balanced Excitation/Inhibition in Neuronal Networks, *The Journal of Neuroscience*, 32(29) (2012), pp. 9817–9823.
 - [11] O. Sporns, J. Zwi, The small world of the cerebral cortex, *Neuroinformatics*, 2(2) (2004), pp. 145–162.

Konszenzus szekvenciaillesztések hatékony előállítása

Novák Ádám*

Eötvös József Collegium**

novadam1@gmail.com

1. Előszó

Az Informatikai Műhely „alapító” tagjainak egyikeként nagyon kellemes emlékeket őrzök a műhelyről és az itt töltött éveimről. Különösen jóleső érzés visszaidézni a kezdeteket, amikor a műhelyórák és -kirándulások szinte családi légkörben zajlottak, és minden egyes tagnak a műhelyben saját beosztás jutott. Nemigen fogom feledni az első ismerkedésünket a λ -kalkulus szépségeivel, amelyet Csörnyei Zoltán tanár úr kezdettől fogva igazi átéléssel adott tovább a fiatalabb generációknak.

Hosszú idő telt el, és nagy öröömre szolgál, hogy az igencsak megnőtt műhely valamint annak volt és jelenlegi tagjai sikerekben gazdag éveket hagynak maguk mögött – mára hosszú a díjak, kitüntetések, és szép tudományos eredmények listája. Csak azt remélhetem, hogy hasonlóan remek tíz év követi majd az első tízet, és ehhez kívánok jó munkát, kitartást és szerencsét az Informatikai Műhelynek.

Ami saját életutamat illeti, harmadéves programtervező matematikus hallgatóként – kicsit rendhagyó mód – a biológus tanulmányaimat is megkezdtem. A valódi érdeklődési körömet végül a kettő ötvöztetésében, a bioinformatika területén találtam meg. A tanulmányaim befejeztével az

*Sophia Genetics SA, Lausanne, Svájc

**2001–2007

Oxfordi Egyetem Statisztika Tanszékén folytattam kutatásaimat, ahol genomikával, szekvenciaillesztéssel, génannotációval foglalkoztam.

A Műhely tízéves évfordulója alkalmából az itt végzett kutatásaimból egy olyan témakört szeretnék bemutatni, amely ugyan biológiához kapcsolódik, de sokkal inkább a Műhelyhez illő matematikai és algoritmikai konstrukciói miatt érdekes. Bízom benne, hogy a munkám érzékelteti a bioinformatika néhány nehéz kihívását, mindemellett bemutat valamit e tudományterület szépségéből.

2. Bevezetés

A genomika korszakában, amikor az ismert DNS-szekvenciák száma rendületlenül, exponenciálisan növekszik, a szekvenciák összehasonlításának, azaz a szekvenciaillesztésnek egyre inkább központi szerep jut.

Mivel sok szekvencia esetén a legvalószínűbb együttes illesztés meghatározása igen nehéz (NP-teljes) feladat, ezért gyakran csak különböző közelítő algoritmusokkal előállított illesztések állnak rendelkezésünkre, amelyek egyike sem megbízható teljes egészében.

A munkám tárgya annak vizsgálata, hogy nagyszámú ill. méretű szekvenciaillesztés hogyan reprezentálható tömören, valamint ebből a reprezentációból hogyan állítható elő hatékonyan olyan konszenzus illesztés, amely pontosságában felülmúlja az egyedi illesztéseket.

2.1. Szekvenciaillesztés

Legyen adott M darab szekvencia, $S_1, \dots, S_M$, ahol $S_i \in \Sigma^*$. A biológiában leggyakrabban DNS, RNS vagy fehérjeszekvenciákkal találkozunk: előbbiek négy különböző alkotóelemből (A, C, G és T/U) felépülő nukleinsavak, utóbbiak $|\Sigma| = 20$ aminosav kombinációiból álló makromolekulák. Közös tulajdonságuk tehát, hogy egy véges ábécé betűiből álló véges szekvenciával leírhatók.

A hasonló funkcióval rendelkező szekvenciák különböző élőlényekben nagy hasonlóságot mutatnak, de általában nem teljesen egyeznek meg. A szekvenciaillesztés az összehasonlított szekvenciák karaktereinek olyan táblázatszerű elrendezését jelenti, ahol a közös eredetű, homológ karakterek egy oszlopba kerülnek (1. ábra).

```

HBA_BOVIN -MVISAAARGNVKAAMGKVGCHAAAYGAALRMFLSPFTTKTYFFHS--LSHGSX---CVRGNGAKVAAATK
HBA_HUMAN -MVISPAARNTNKAAMGKVGHRGAYGAALRMFLSPFTTKTYFFHS--LSHGSX---CVRGNGKRVAAATN
HBB_BOVIN --MHTD--GSAVTRFGKV--NV--GGALGRLLVYVNTQFFS--SGSLSTFAVGNPKVKAHGRKVLGATG
HBB_HUMAN --MHTD--GSAVTRFGKV--NV--GGALGRLLVYVNTQFFS--SGSLSTFAVGNPKVKAHGRKVLGATG

```

1. ábra. Az emberi és a szarvasmarha hemoglobin fehérje α és β alegységeinek egy lehetséges szekvenciaillesztése

Ha egy adott oszlopban valamely szekvencia nem tartalmaz homológ karaktert, ebbe a pozícióba kötőjelet (beszúrás jelet) írunk, ezzel egyforma hosszúvá téve a szekvenciákat. A szekvenciaillesztés így azt is leírja, hogy egy szekvenciából a változtatások milyen sorozatával (pontmutáció, beszúrás, törlés) állítható elő egy másik szekvencia.

2.2. Konszenzus illesztés

Tegyük fel, hogy adott a fenti szekvenciák N darab, nem feltétlenül különböző illesztése, $A_1, \dots, A_N$. Ezek készülhettek akár különböző szekvenciaillesztő algoritmusokkal, akár statisztikus mintavételezéssel az illesztések egy eloszlásából, amelyet evolúciós modellek írnak le [6]. Mindkét esetben igaz, hogy a nagyobb gyakorisággal előforduló illesztési mintázatok nagyobb valószínűséggel helyesek. A feladatunk egyetlen *konszenzus illesztés* előállítása, amely a kiinduló illesztések legmegbízhatóbb részeit egyesíti.

A továbbiakban először bevezetjük az *illesztési oszlopok* egy célravezető kódolását, majd megmutatjuk, hogy a szekvenciaillesztések halmaza az oszlopok egy irányított körmentes gráfiájával tömören leírható (ld. később 3. ábra), amely a kódolt oszlopokból hasítótáblák használatával hatékonyan felépíthető.

Ezután bemutatunk egy dinamikus programozási elven működő algoritmust, amellyel különböző szempontok szerint optimális konszenzus illesztések az oszlopok számában lineáris idő alatt előállíthatók. Végül megvizsgáljuk, hogy a bemutatott módszer más konszenzus illesztési eljárásokkal összevetve milyen pontosságot képvisel, illetve kitérünk a konstrukció további felhasználási területeire [4].

3. Módszerek

A rendelkezésre álló szekvenciaillesztések tömör ábrázolásához és összegzéséhez szükséges, hogy valamilyen módon azonosítsuk az illesztések közös elemeit. Más szerzők e célra a lehető legkisebb egységet, az illesztett karakterpárokat választották [10]. Ennek a döntésnek a hátrányos következménye az, hogy a konszenzus illesztés előállítása ugyanolyan nehéz (NP-teljes) problémává válik, mint maga a szekvenciaillesztés: nincs remény egy adott szempont szerint optimális konszenzus megkeresésére, és csak heurisztikus módszerekkel állíthatók elő közelítő eredmények.

Ebben a munkában az *illesztési oszlopot* választjuk egységként, és megmutatjuk, hogy az így előálló optimalizálási problémák nagyon hatékonyan megoldhatók. Ám ehhez először fel kell derítenünk az illesztési oszlopok alkotta hálózatok struktúráját, valamint meg kell adnunk a legcélravezetőbb kritériumot, ami szerint az optimális konszenzust választjuk.

3.1. Illesztési gráfok

3.1.1. Illesztések kódolása

Adottak az $S_1, \dots, S_M$ szekvenciák, ahol az S_k szekvencia hosszát jelölje l_k , a j -edik karakterét $s_{k,j}$. E szekvenciák $\bar{A}$ illesztése egy $M \times L$ méretű mátrix, amelynek $\bar{a}_{k,i}$ eleme vagy S_k valamely eleme, vagy pedig a '–' beszúrás (gap) karakter, azzal a megszorítással, hogy $\bar{A}$ -nak k . sorából a beszúrás karaktereket elhagyva pontosan S_k -t kapjuk. Kizárólag beszúrásból álló oszlopok nem fordulnak elő az illesztésben.

Bevezetjük az illesztések egy olyan kódolását, amelyben a nem-beszúrás karakterekhez a soron következő páros számot rendeljük, a beszúrás karakterekhez pedig közbeeső (akár többször ismétlődő) páratlan számokat. Formálisan leírva, az $\bar{A}$ illesztéshez azt az A kódolt illesztést rendeljük, amelynek elemei a következők:

$$a_{k,i} = \begin{cases} a_{k,i-1} + (1 - a_{k,i-1} \bmod 2) + 1 & \text{ha } \bar{a}_{k,i} \neq \text{'–'} \\ & \text{vagy } i = L + 1 \\ a_{k,i-1} + (1 - a_{k,i-1} \bmod 2) & \text{ha } \bar{a}_{k,i} = \text{'–'} \end{cases} \quad (1)$$

minden $i = 1, \dots, L + 1$ és $k = 1, \dots, M$ -re, ahol $a_{k,0} = 0$. Megjegyezzük,

hogy a kódolt illesztést bővítettük egy iniciális és egy záró oszloppal, amelyek minden szekvencia képzeletbeli nulladik ill. $(l_k + 1)$ -edik karakterét tartalmazzák. Ennek a későbbiekben lesz jelentősége.

A definícióból teljes indukcióval könnyen belátható a következő:

$$a_{k,i} = \begin{cases} 2j & \text{ha } \bar{a}_{k,i} = s_{k,j} \\ 2j + 1 & \text{ha } \bar{a}_{k,i} \text{ beszúrás } s_{k,j} \text{ és } s_{k,j+1} \text{ között} \end{cases} \quad (2)$$

minden $i = 1, \dots, L$ és $a_{k,L+1} = 2l_k + 2$ -re.

A	B	-	C	-	D	E
A	-	-	B	C	D	E
A	-	B	C	-	D	E

$\bar{A}$

↓

0	2	4	5	6	7	8	10	12
0	2	3	3	4	6	8	10	12
0	2	3	4	6	7	8	10	12

A

2. ábra. Három azonos, $ABCDE$ karakterekből álló szekvencia egy illesztése és az abból számított kódolt illesztés. A beszúrás karaktereket félkövér kiemeléssel jeleztük a kódolt illesztésben, amely egy kezdő és egy záró oszloppal van bővítve

Példaképp bemutatjuk három azonos, $ABCDE$ karakterekből álló szekvencia egy lehetséges illesztését és annak kódolását a 2. ábrán. E kódolási séma célja később világos lesz, de ezen a ponton érdemes megjegyezni, hogy a kódolás megkülönbözteti a beszúrásokat attól függően, hogy a szekvencia mely karakterei között történnek. Ez a megkülönböztetés követi néhány evolúciós modell, pl. a TKF91 [9] tulajdonságait, de ennél is fontosabb, hogy a beszúrások helyfüggetlen kezelése NP-teljes optimalizálási problémához vezet [5], és emiatt nem célravezető.

Ha adott az illesztési oszlopok egy lehetséges halmaza, annak eldöntéséhez, hogy valamely két oszlop szerepelhet-e együtt egy érvényes illesztésben, meg kell vizsgálnunk minden oszlop *megelőző* és *követő* oszlopait. Ennek formalizálásához bevezetjük a $\ltimes$ *megelőzési relációt*, amelyet úgy definiálunk, hogy $i \ltimes j$ akkor és csak akkor áll fenn, ha létezik olyan érvényes illesztés, amelyben az i oszlopot *közvetlenül* a j oszlop követi. Bármely érvényes A illesztésre igaz emiatt, hogy

$A^0 \ltimes A^1 \ltimes \dots \ltimes A^{L+1}$, ahol A^i a kódolt illesztés i . oszlopát jelenti. Mivel a szekvenciák minden karaktere egyszer fordulhat elő egy illesztésben, ezért $A^i \not\ltimes A^i$.

A megelőzési reláció jellemezhető a megelőző (f_P) és a követő (f_S) függvényekkel is. Az f_P és f_S függvények az A^i oszlophoz egy azonos elemszámú vektort rendelnek, és definíciójuk a következő:

$$f_P(A^i)_k = a_{i,k} - (1 - a_{i,k} \bmod 2) \quad (3)$$

$$f_S(A^i)_k = a_{i,k} + (1 - a_{i,k} \bmod 2) \quad (4)$$

Intuitíve az f_P függvény A^i elemeit lefelé kerekíti a legközelebbi páratlan számhoz, az f_S pedig felfelé a legközelebbi páratlan számhoz, az eleve páratlan elemeket változatlanul hagyva. Ekkor érvényes a következő tulajdonság:

3.1. Lemma. *Az A^i és A^j oszlopokra a megelőzési reláció átfogalmazható a megelőző és követő függvényekkel, és pedig: $A^i \ltimes A^j \iff f_S(A^i) = f_P(A^j)$.*

Bizonyítás. Az $f_S(A^i) = f_P(A^j)$ egyenlőségből adódik, hogy $a_{k,i} + (1 - a_{k,i} \bmod 2) = a_{k,j} - (1 - a_{k,j} \bmod 2)$ fennáll minden k sorra. Ha $a_{k,i}$ páros (azaz nem beszúrási karakternek felel meg), akkor bármely azt követő karakter vagy egy beszúrási karakter, amelyre $a_{k,j} = a_{k,i} + 1$, vagy pedig a k . szekvencia következő karaktere, amelyre $a_{k,j} = a_{k,i} + 2$. Ha viszont $a_{k,i}$ páratlan (beszúrási karakter), a következő karakter vagy beszúrási lehet ($a_{k,j} = a_{k,i}$) vagy a szekvencia következő karaktere ($a_{k,j} = a_{k,i} + 1$). Minden esetben fennáll az egyenlőség. $\square$

3.1.2. Illesztési oszlopok illesztések egy sorozatából

Ha adottak *ugyanazon* bemeneti szekvenciák nem feltétlenül különböző $A_1, \dots, A_N$ kódolt illesztései, az illesztésekből előforduló oszlopok halmazát jelölje $\mathcal{X} = \{A_j^i \mid A_j^i \text{ az } A_j \text{ egy oszlopa, } 1 \leq j \leq N\}$. Itt az illesztési oszlopok egyenlőségét a szokásos módon definiáljuk, azaz $A^i = A^j \iff a_{k,i} = a_{k,j}$ minden $k = 1, \dots, M$ sorra. Esetünkben az illesztéseket statisztikai szimulációból mintavételezzük, de a módszer ugyanazon szekvenciák tetszőleges módon kapott különböző illesztéseire is alkalmazható, beleértve a különböző illesztési algoritmusok által adott illesztéseket.

Az $X \in \mathcal{X}$ oszlopok egy vagy több illesztésben fordulhatnak elő. Az iniciális $A_j^0 = (0, \dots, 0)$ és záró $A_j^{L_j+1} = (2l_1 + 2, \dots, 2l_M + 2)$ oszlopok viszont minden illesztésnek részei. Ez azonnal adódik a definíciójukból és abból a tényből, hogy minden illesztés ugyanazokat a szekvenciákat tartalmazza, így azok hossza is azonos. Ezekre a közös oszlopokra X^0 -lal és X^T -vel fogunk hivatkozni.

A megelőzési reláció segítségével egy adott X oszlop megelőző oszlopaikat megadhatjuk $\mathcal{P}(X) = \{X' \in \mathcal{X} \mid X' \bowtie X\}$ -szel, és hasonlóan a követő oszlopaikat $\mathcal{S}(X) = \{X' \in \mathcal{X} \mid X \bowtie X'\}$ -szel. Mivel $\mathcal{X}$ -et teljes illesztésekből állítottuk elő, ezért igazak a következő állítások:

$$\begin{aligned} \mathcal{P}(X^0) &= \emptyset \text{ és bármely } X \neq X^0 : \mathcal{P}(X) \neq \emptyset \\ \mathcal{S}(X^T) &= \emptyset \text{ és bármely } X \neq X^T : \mathcal{S}(X) \neq \emptyset \end{aligned} \quad (5)$$

Így már belátható az alábbi fontos tulajdonság.

3.2. Tétel. *Ha az $X', X'' \in \mathcal{S}$ oszlopoknak legalább egy megelőző oszlopuk közös, akkor a megelőző osztályuk egybeesik, és hasonlóan, ha legalább egy követő oszlopuk közös, a követő osztályuk egybeesik, azaz formálisan:*

$$\begin{aligned} \exists X \in \mathcal{S}(X') \cap \mathcal{S}(X'') &\implies f_S(X') = f_S(X'') \iff \mathcal{S}(X') = \mathcal{S}(X'') \\ \exists X \in \mathcal{P}(X') \cap \mathcal{P}(X'') &\implies f_P(X') = f_P(X'') \iff \mathcal{P}(X') = \mathcal{P}(X'') \end{aligned}$$

Bizonyítás. Az első állítást vizsgáljuk, a második analóg módon bizonyítható. A bal oldali implikáció a 3.1. lemma kétszeri alkalmazásával áll elő. Ugyanebből a lemmából következik, hogy ha $f_S(X') = f_S(X'')$ és $X \in \mathcal{S}(X')$, akkor $X \in \mathcal{S}(X'')$, és fordítva, amely megadja a jobb oldali ekvivalencia egyik irányát. A fordított irány a bal oldali implikáció egy speciális esete, kivéve ha $\mathcal{S}(X') = \emptyset$, de ez esetben $X' = X'' = X^T$ az 5. egyenletből, amellyel a bizonyítás teljes. $\square$

A fentiekből következik, hogy a megelőző és követő függvények teljes egészében meghatározzák a megelőző és követő halmazokat, és ami még lényegesebb, hogy két oszlopnak csakis akkor lehet közös megelőző (ill. követő) oszlopa, ha az összes megelőző (követő) oszlopuk megegyezik. Ez a tulajdonság később hasznos lesz.

3.1.3. Az illesztési gráf felépítése

3.3. Definíció. $A \bowtie$ megelőzési reláció egy $\mathcal{D}(\mathcal{X})$ irányított gráfot határoz meg az $\mathcal{X}$ oszlopok mint csúcsok felett: X és X' közé akkor és csak akkor húzunk (irányított) élt, ha $X \bowtie X'$.

Szintén bevezetjük oszlopok lexikografikus összehasonlítását: $X < X' \iff X_k \leq X'_k \ \forall k$ és $X \neq X'$.

3.4. Lemma. . Bármely $\mathcal{D}(\mathcal{X})$ gráfbeli $\pi = X^0 \bowtie \dots \bowtie X^T$ útra és annak $X \neq X' \in \pi$ oszloppaira igaz, hogy $f_P(X) \neq f_P(X')$ és $f_S(X) \neq f_S(X')$.

Bizonyítás. Az X kódolt illesztési oszlop páros x_k elemeire $f_P(X)_k = f_S(X)_k - 2$. A páratlan, beszúrásokat reprezentáló elemekre pedig $f_P(X)_k = f_S(X)_k$. Mivel a csak beszúrást tartalmazó oszlopok nem megengedettek, az előbbiekből adódik, hogy $f_P(X) < f_S(X)$. A 3.1. lemmából tudjuk, hogy minden $X \bowtie X'$ -re $f_S(X) = f_P(X')$. Az előző állítással együtt ez azt jelenti, hogy $f_P(X) < f_P(X')$ és $f_S(X) < f_S(X')$. $\square$

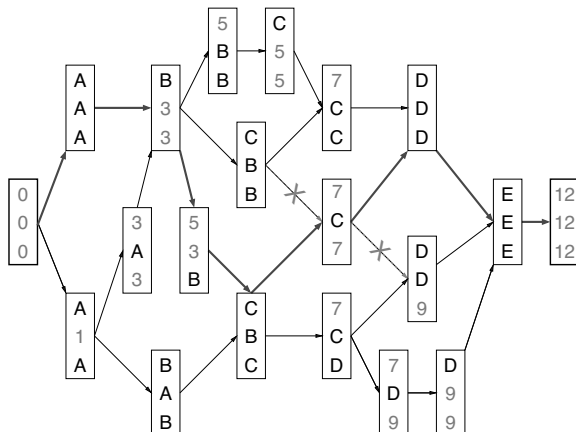
3.5. Tétel. $\mathcal{D}(\mathcal{X})$ körmentes.

Bizonyítás. A 3.4. lemma szerint egyetlen oszlop sem fordulhat elő kétszer a gráf egy irányított útján. $\square$

Könnyen látható, hogy bármely érvényes illesztés az illesztési gráfban egy olyan irányított útnak felel meg, amely az X^0 -ból indul és X^T -ben ér véget. Jelöljük $\Pi(\mathcal{X})$ -szel az érvényes illesztések e halmazát. Az illesztéseknek az $\mathcal{A}$ halmaza, amelyből a $\mathcal{D}(\mathcal{X})$ gráfot előállítottuk, természetesen szerepel $\Pi(\mathcal{X})$ -ben, azaz $\mathcal{A} \subseteq \Pi(\mathcal{X})$. A két halmaz viszont általában véve nem azonos, hiszen előfordulhat, hogy két (vagy több) oszlop és azok követő oszloppai között nem minden megelőző–követő oszloppár fordul elő a bemeneti illesztésekben. A 3. ábrán bemutatunk egy 18 oszlopból álló illesztési gráfot, amelyben a kék színnel kiemelt út egy érvényes illesztést jelöl.

3.2. Valószínűségi eloszlások a gráfon

Az illesztési gráf célja, hogy a mintavételezett illesztéseken keresztül az illesztések teljes poszterior eloszlásának egy összegzését adja.



3. ábra. Három azonos szekvencia néhány illesztéséből készített illesztési gráf. A kiemelt útvonal a 2. ábrán lévő illesztéssel azonos. Az áthúzott nyilakkal összekötött oszloppárok nem egymás rákövetkezői, mert a C és D közötti beszúrásokat (5) megkülönböztetjük a D és E közötti beszúrásoktól (7).

Ehhez felhasználhatjuk az illesztési oszlopok bemeneti illesztésekből becsült valószínűségeit mint a poszterior eloszlás peremeloszlásait. Jelöljük $\delta(X, A_j) \in \{0, 1\}$ -gyel, hogy X oszlop szerepel-e az A_j illesztésben, ill. $p(X) = \Pr\{\delta(X, A_j) = 1\}$ -gyel annak a marginális valószínűségét, hogy az X oszlop része valamely illesztésnek. A legegyszerűbb módja $p(X)$ becslésének, ha meghatározzuk az oszlop relatív gyakoriságát a bemeneti illesztésekben:

$$f(X) = \frac{\sum_{j=1}^N \delta(X, A_j)}{N}$$

Az X^0 kezdő- és X^T záróoszlop marginális valószínűsége 1.

3.2.1. A poszterior dekódolás

Nem foglalkoztunk eddig azzal a kérdéssel, hogy az oszlopok által meghatározott gráfból, amelyen adottak az oszlopok marginális valószínűségeinek becslései, milyen szempontok szerint válasszuk ki a konszenzus illesztést.

Kézenfekvő módon adódik a lehetőség, hogy azt az utat keressük meg a gráfban, amelyen az oszlopok peremvalószínűségeinek valamely függvénye optimális értéket vesz fel. Gyakori stratégia pl. rejtett Markov modellek területén, hogy a teljes poszterior helyett annak peremeloszlásainak egy veszteségfüggvényét optimalizálják. Ezt hívják maximum poszterior decoding (MPD) eljárásnak is.

Esetünkben minden lehetséges oszlopra a „helyi veszteséget” kifejezhetjük a fals pozitív (FP) valószínűséggel (az oszlop része a kiválasztott illesztésnek, pedig a helyes, v. referencia illesztésben nem szerepel) és a fals negatív (FN) valószínűséggel (fordítva, az oszlop nem része a kiválasztott illesztésnek, pedig a helyes illesztésben szerepel). Ha ezeket két súlyparaméterrel, a λ_{FP} -vel és a λ_{FN} -nel büntetjük, [3] szerint a következő veszteségfüggvényt kapjuk:

$$\mathbb{E}[L(A, A_r|S)] = \text{const.} - (\lambda_{FP} + \lambda_{FN}) \sum_{i \in A_r} (p(A^i) - \lambda_{FP}/(\lambda_{FP} + \lambda_{FN})) \quad (6)$$

E függvény minimalizálása maximalizálja a helyesen megválasztott oszlopok várható számát, megfelelően súlyozva a fals pozitív és a fals negatív oszlopokat. Ez ekvivalens azonban a marginális oszlopvalószínűségek és egy oszloponkénti konstans büntetés, a $g = \lambda_{FP}/(\lambda_{FP} + \lambda_{FN})$ összegének maximalizálásával. Ezzel egyszerű kritériumot kaptunk az optimalizáláshoz, ahol a g paraméter statisztikailag közvetlenül értelmezhető. A $g = 1$ érték a fals pozitív oszlopok várható számának minimalizálását jelenti, ami az oszlopok számának büntetésén keresztül tömörebb illesztést eredményez. A $g = 0$ érték ellenkezőleg, hosszabb illesztést preferál. A $g = 0.5$ egy kiegyensúlyozottabb választás, ahol a FP és a FN oszlopokat azonosan büntetjük, és egyéb előzetes információ híján megfelelő lehet.

3.3. Az MPD illesztés meghatározása

Ha adott az $\mathcal{X}$ megfigyelt oszlopok $\mathcal{D}(\mathcal{X})$ illesztési gráfja és az oszlopok peremeloszlásának egy $\tilde{p}(X)$ közelítése, a célunk most már az MPD illesztés, vagyis annak a $\pi = X^0 \ltimes \dots \ltimes X^T$ útvonalnak a meghatározása, amelyre $\sum_{X \in \pi} (\tilde{p}(X) - g)$ maximális. Ez ekvivalens a 6. kifejezés minimalizálásával, és dinamikus programozás segítségével egyszerűen elvégezhető, az 1. algoritmus szerint.

Megjegyezzük, hogy az algoritmus első, egyszerűsített változatában feltesszük, hogy az illesztési gráf már rendelkezésünkre áll, azaz egy adott oszlop rákövetkezőit hatékonyan fel tudjuk sorolni. Továbbá az oszlopokat tetszés szerinti indexekkel azonosítjuk, és nem foglalkozunk ennek az indexelésnek a megvalósításával. Később visszatérünk ezekre problémákra.

Algoritmus 1 MPD illesztés (implicit topologikus rendezéssel)

```

1:  $F \leftarrow (-\infty, \dots, -\infty)$   $\triangleright$  dinamikus programozási (DP) vektor
2:  $V \leftarrow (0, \dots, 0)$   $\triangleright$  DP optimális rákövetkezési vektor
3:
4: function MPDALIGNMENT( $g$ )
5:    $F[\text{lastCol}] \leftarrow 0$   $\triangleright$  a DP vektor utolsó elemének inicializálása
6:   SCOREFROM( $\text{firstCol}$ ,  $g$ )  $\triangleright$  DP pontszámok és rákövetkezők
     számítása
7:
8:    $A \leftarrow ()$   $\triangleright$  optimális DP útvonal rekonstruálása
9:    $\text{col} \leftarrow V[\text{firstCol}]$ 
10:  while  $\text{col} \neq \text{lastCol}$  do
11:     $A.\text{ADD}(\text{col})$   $\triangleright$  aktuális oszlop MPD-be
12:     $\text{col} \leftarrow V[\text{col}]$   $\triangleright$  optimális útvonal követése
13:  return  $A$ 
14:
15: function SCOREFROM( $\text{col}$ ,  $g$ )
16:  if  $F[\text{col}] = -\infty$  then  $\triangleright$  oszloponként csak egyszer
17:    for  $\text{succ} \in \mathcal{S}(\text{col})$  do  $\triangleright$  rákövetkezők iterálása
18:       $\text{score} \leftarrow \tilde{p}(\text{col}) - g + \text{SCOREFROM}(\text{succ}, g)$ 
19:      if  $\text{score} > F[\text{col}]$  then
20:         $F[\text{col}] \leftarrow \text{score}$   $\triangleright$  optimális pontszám
21:         $V[\text{col}] \leftarrow \text{succ}$   $\triangleright$  és ahhoz tartozó útvonal
22:  return  $F[\text{col}]$ 

```

3.3.1. Az MPD algoritmus helyessége

Akárcsak a jól ismert legrövidebb út probléma esetében, egy adott X csúcsból X^T csúcsba vezető $\mathcal{D}(\mathcal{X})$ -beli optimális útvonalnak mindig

része X valamely rákövetkezőjéből X^T -be vezető egyik optimális útvonal. Ebből következik, hogy az X -ből induló út optimális pontszáma kifejezhető az X oszlop pontszámának és a rákövetkezőiből induló optimális utak pontszámmaximumának összegeként.

Az 1. algoritmus SCOREFROM(col, g) eljárása pontosan eszerint határozza meg a col oszlopból induló optimális út pontszámát. A helyesége abból a tényből következik, hogy a $\mathcal{D}(\mathcal{X})$ irányított gráf körmentes, ezért létezik a csúcsoknak egy $X^1, X^2 \dots X^K$ topologikus rendezése, amelyre $X^i \bowtie X^j \implies i < j$. A gráf speciális szerkezete miatt továbbá $X^1 = X^0$ és $X^K = X^T$ minden esetben. A csúcsok fordított topologikus sorrendjében teljes indukcióval érvelünk: SCOREFROM(col, g) meghatározza col-tól X^T -ig az optimális pontszámot, ha SCOREFROM(succ, g) megadja ugyanezt a col oszlop minden succ rákövetkezőjére. Viszont a lastCol (X^T) oszlopra biztosan visszaadja a 0 optimális pontszámot, hiszen az 5. sorban így inicializáltuk. Innen fordított topologikus sorrendben haladva az indukciós feltétel minden esetben teljesül, mert a col oszlop rákövetkezőit már feldolgoztuk, így az optimálist azokra már beláttuk.

Az egyetlen hiányzó lépés annak igazolása, hogy ha $F[\text{col}] \neq -\infty$ az eljárás első sorában, akkor az csakis az optimális pontszám lehet. Valóban, kizárólag akkor lehet ettől eltérő érték, ha részleges optimumot tartalmaz, ami ellenben csak úgy lehetséges, ha egy korábbi SCOREFROM(col, g) hívás ugyanarra a col oszlopra még folyamatban van. Ez viszont azt jelenti, hogy létezik olyan útvonal, ami a col oszlopon kétszer halad át, de ez ellentmond a körmentességnek.

Innen már könnyen látható, hogy megkaphatunk egy lehetséges optimális útvonalat úgy, hogy az A-ban eltárolt optimális rákövetkezőket követjük.

3.3.2. Az MPD algoritmus időigénye

A SCOREFROM(firstCol, g) hívás időigénye határozza meg az 1. algoritmus teljes aszimptotikus futási idejét. A 18. és 27. sor annyiszor kerül végrehajtásra, ahányszor a SCOREFROM eljárás rekurzívan meghívódik. Ez csakis a 20. sorból történhet. A 19. sort illesztési oszloponként egyszer érjük el: ha egyszer elértük, ugyanarra az oszlopra nem hívódhat meg újra a SCOREFROM, mielőtt a ciklus befejeződné, hiszen ez azt jelentené, hogy a gráf nem körmentes. Minden további

SCOREFROM hívásra pedig az $F[\text{col}]$ már meg van határozva. Ez azt jelenti, hogy a 20–21. sorok, így következképp a 18. és 27. sorok is minden oszlop minden rákövetkezőjére egyszer hajtódna végre, ami alapján az algoritmus teljes futási ideje $\Theta(E)$, ahol E a gráf éleinek száma. Feltételeztük, hogy az oszlop rákövetkezőinek felsorolása azok számával arányos időt vesz igénybe.

Az élek E száma viszont a csúcsok (egyedi illesztési oszlopok) számával, $V = |\mathcal{X}|$ -szel négyzetesen nőhet, amely pedig közel lineárisan nő legrosszabb esetben a bemenet méretével, $S = LM$ -mel, ahol $L = \sum_{i=1}^N L_i$ a bemeneti illesztési oszlopok száma és M a szekvenciák száma. Precízebben vizsgálva, a $V = \Theta(L) = \Theta(S)$ legrosszabb esetben valóban csak úgy állhatna elő, ha korlátlanul tudnánk növelni az egyedi oszlopok számát fix M mellett is, ami nem lehetséges. Az oszloptér mérete nem végtelen, ám meglehetősen gyorsan növekszik M -mel: a legrosszabb esetben akkor áll elő, ha $L = \Omega(2^M)$, amelyből $S = O(L \log L)$ és $L = \Omega(S/\log S)$, ezért a pontos futási idő a legrosszabb esetben $\Theta(E) = \Theta(V^2) = \Theta(L^2) = \Omega(S^2/\log^2 S)$.

A közel négyzetes futási idő mellett további gyakorlati probléma, hogy a gráf természetes, csúcslistás ábrázolásához szükséges tár mérete szintén közel négyzetesen nő a bemenettel, amely nagyon sok illesztési minta esetén meghaladhatja a rendelkezésre álló erőforrásokat. Továbbá nem is vizsgáltuk még, hogy mily módon határozható meg hatékonyan egy oszlop összes rákövetkezője, amely szükséges a gráf felépítéséhez. Szerencsére a gráf speciális szerkezetét kihasználva e problémák mindegyike megoldható, amit a következőkben tárgyalunk.

3.4. Hatékony MPD számítás a követő osztályokkal

Az 1. algoritmus négyzetes futási ideje onnan ered, hogy az illesztési oszlopok rákövetkezőit expliciti tároljuk és iteráljuk. A követő halmazok előnyös tulajdonságai ennél hatékonyabb megoldást is lehetővé tesznek. A 3.2. tétel mutatja, hogy a követő oszlopok osztályokat alkotnak: két oszlopnak vagy az összes rákövetkezője azonos, vagy egyetlenegy sem. Ezenfelül bármely követő halmazhoz könnyen készíthető egy azonosító f_S segítségével. Ha minden oszlop helyett követő osztályonként egyszer adjuk meg a követő halmazok listáját, ezt a listát a megelőző oszlopok együttesen használhatják. Hasonlóan, a rákövetkező oszlopok optimális pontszámai közül a legnagyobb kiválasztását elegendő követő

halmazonként egyszer meghatározni, és az eredményt minden megelőző oszlop felhasználhatja. A 2. és a 3. algoritmusok erre az ötletre épülnek.

A BUILD DAG eljárás felépíti a gráfot a bemeneti illesztések egy listájából. A colHash hasítótábla segítségével tartja nyilván a már előfordult oszlopokat és azok számát. Amikor egy új oszlop kerül a gráfba, meghatározzuk annak követőosztályát. Az első lépés az f_S követőfüggvény értékének kiszámítása, ezt követően megadjuk az osztály azonosítóját a succHash hasítótáblával. Ha az osztály először fordul elő, üres követőlistával inicializáljuk, ezt a SUCCCLASSID eljárás végzi. Amint adott a követőosztály azonosítja, elmentjük azt, mert ez a követő oszlopok megelőző osztálya is egyben (ld. 3.1. lemma és 3.2. tétel). Így a következő új oszlophoz érve, amikor annak már ismert megelőző osztályából egy élt húzunk az oszlophoz (22. sor), ezzel egyúttal az oszlop összes lehetséges megelőzőjét is összekötjük az oszloppal, beleértve a csak később előforduló, új megelőző oszlopokat is. Ezzel az adatszerkezettel egy oszlop rákövetkezői egyszerűen, két lépésben megkereshetők: (1) a oszlop követőosztály-azonosítójának betöltése, (2) az osztály követőlistájának olvasása; vagy röviden SL[S[col]].

E gondolatmenetet követve könnyen látható, hogy a 12–27. sorban lévő ciklus minden iterációjával a BUILD DAG eljárás kiterjeszti az illesztési gráfot a feldolgozás alatt álló illesztés oszlopaival, és helyesen frissít minden követőlistát, oszlopszámot és oszlop/osztály hasítótáblát.

Vizsgáljuk meg most a 3. algoritmus helyességét. Mivel az 1. algoritmusmal azonos elven működik, elegendő belátnunk a dinamikus programozási táblázataik közti kapcsolatot. A $\max_{\pi=X^0 \bowtie \dots \bowtie X^T} \sum_{X \in \pi} (\tilde{p}(X) - g)$ útvonal meghatározásához az 1. algoritmusban az $F(X)$ DP vektort használtuk, amelyre a következő rekurzív összefüggés áll fenn:

$$F(X) = \max_{\substack{X^1, \dots, X^R \\ \pi=X \bowtie X^1 \bowtie \dots \bowtie X^R \bowtie X^T}} \sum_{X' \in \pi} (\tilde{p}(X') - g) = \max_{\substack{X^1 \\ X \bowtie X^1}} (\tilde{p}(X) - g + F(X^1)) \quad (7)$$

valamint $F(X^0)$ megadja az optimális MPD pontszámot. Ezt tovább alakítva a 3.1. lemma segítségével:

$$F(X) = \tilde{p}(X) - g + \max_{\substack{X^1 \\ f_S(X) = f_P(X^1)}} F(X^1) = \tilde{p}(X) - g + F'(f_S(X)) \quad (8)$$

Algoritmus 2 Az illesztési gráf felépítése

```

1: colHash  $\leftarrow \{ \}$   $\triangleright$  kódolt oszlopokat az indexükhöz rendeli
2: succHash  $\leftarrow \{ \}$   $\triangleright$  követő osztályokat az indexükhöz rendeli
3:  $X \leftarrow ( )$   $\triangleright$  a gráf összes kódolt oszlopa
4:  $C \leftarrow ( )$   $\triangleright$  oszlopszámok
5:  $S \leftarrow ( )$   $\triangleright$  követő osztály minden oszlopra
6:  $SL \leftarrow ( )$   $\triangleright$  követő oszlopok listája minden osztályra
7: fsucc  $\leftarrow 0$   $\triangleright$  első oszlop követőosztály-azonosítója
8: lpred  $\leftarrow 0$   $\triangleright$  utolsó oszlop megelőzőosztály-azonosítója
9:
10: procedure BUILD DAG(alignList)
11:   fsucc  $\leftarrow$  SUCCCLASSID( $f_S(\text{firstCol})$ )  $\triangleright$  első oszlop feldolgozása
12:   for align in alignList do
13:     pred  $\leftarrow$  fsucc  $\triangleright$  következő oszlop megelőző osztálya
14:     for col in alignList do  $\triangleright$  minden oszlop kiv.  $X^0$  és  $X^T$ 
15:       if col  $\in$  colHash then  $\triangleright$  ha már előfordult
16:         id  $\leftarrow$  colHash[col]  $\triangleright$  oszlopindex betöltése
17:         C[id]  $\leftarrow$  C[id] + 1  $\triangleright$  oszlopszám növelése
18:       else
19:         id  $\leftarrow$  X.ADD(col)  $\triangleright$  oszlop és index mentése
20:         colHash[col]  $\leftarrow$  id  $\triangleright$  hasítótáblába helyezés
21:         C[id]  $\leftarrow$  1  $\triangleright$  oszlopszám inicializálása
22:         SL[pred].add(id)  $\triangleright$  megelőző követőlistájába teszi
23:         S[id]  $\leftarrow$  SUCCCLASSID( $f_S(\text{col})$ )  $\triangleright$  követőosztály számítása
24:         pred  $\leftarrow$  S[id]  $\triangleright$  következő megelőző osztálya
25:       lpred = pred  $\triangleright$  utolsó oszlop megelőző osztálya
26:
27: function SUCCCLASSID(key)
28:   if key  $\notin$  succHash then
29:     id  $\leftarrow$  SL.ADD( $()$ )  $\triangleright$  új osztály, üres követőlista
30:     succHash[key]  $\leftarrow$  id  $\triangleright$  új osztály indexe
31:   return succHash[key]

```

Algoritmus 3 Gyors MPD illesztés (implicit topologikus rendezéssel)

```

1:  $F \leftarrow (-\infty, \dots, -\infty)$   $\triangleright$  dinamikus programozási vektor
2:  $V \leftarrow (0, \dots, 0)$   $\triangleright$  DP optimális rákövetkezési vektor
3:
4: function FASTMPDALIGN( $g$ )
5:    $F[lpred] \leftarrow 0$   $\triangleright$  DP utolsó elemének inicializálása
6:   FASTSCOREFROM( $fsucc, g$ )  $\triangleright$  DP pontszámok és következők
     számítása
7:
8:    $A \leftarrow ()$   $\triangleright$  optimális DP útvonal rekonstruálása
9:    $id \leftarrow V[fsucc]$ 
10:  while  $id \neq 0$  do
11:     $A.ADD(X[id])$   $\triangleright$  aktuális oszlop MPD-be
12:     $id \leftarrow V[S[id]]$   $\triangleright$  optimális útvonal követése
13:  return  $A$ 
14:
15: function FASTSCOREFROM( $id, g$ )
16:  if  $F[id] = -\infty$  then  $\triangleright$  oszloponként csak egyszer
17:    for  $succ \in SL[id]$  do  $\triangleright$  osztály rákövetkezői
18:       $score \leftarrow \tilde{p}(X[succ]) - g + FASTSCOREFROM(S[succ], g)$ 
19:      if  $score > F[id]$  then
20:         $F[id] \leftarrow score$   $\triangleright$  optimális pontszám
21:         $V[id] \leftarrow succ$   $\triangleright$  és ahhoz tartozó útvonal
22:  return  $F[id]$ 

```

ahol $F'(z)$ -t a következőképp definiáljuk:

$$F'(z) = \max_{\substack{X^1 \\ f_P(X^1)=z}} F(X^1) \quad (9)$$

A 8. és a 9. állítások összevonásából:

$$F'(z) = \max_{\substack{X^1 \\ f_P(X^1)=z}} (\tilde{p}(X^1) - g + F'(f_S(X^1))) \quad (10)$$

amely egy rekurzív összefüggés a követőosztályokon definiált $F'(z)$ -re. Továbbá, a 8. összefüggésből $F(X^0) = 1 - g + F'(f_S(X^0))$, azaz az optimális MPD pontszám közvetlenül megadható $F'(f_S(X^0))$ -ból.

A 3. algoritmus F vektora megegyezik $F'(z)$ -vel, és a FASTSCOREFROM eljárás pontosan a 10. egyenlet szerinti rekurzív számítást végzi, biztosítva, hogy $F(\text{id})$ minden követő osztályra pontosan egyszer számíttódjon ki. A 3. algoritmus helyessége így már következik a 3.3.1. szakaszban adott érvelésből.

3.4.1. A javított algoritmusok időigénye

Az illesztési gráfot felépítő 2. algoritmus a bemeneti illesztések minden oszlopát pontosan egyszer dolgozza fel. A futási ideje ezért azonnal adódik abból, hogy minden művelet $O(1)$ időigényű, kivéve a követő osztály azonosítójának számítása f_S -sel, amely $\Theta(M)$ időt vesz igénybe, ahol M a szekvenciák száma. Feltettük, hogy a dinamikusan növekvő tömbhöz egy elem hozzáadása konstans idejű művelet – valóban léteznek amortizált $O(1)$ idejű megoldások. A teljes futási idő ezért $\Theta(LM) = \Theta(S)$ ahol $L = \sum_{i=1}^N L_i$ a bemeneti illesztési oszlopok száma, $S = LM$ pedig a bemenet teljes mérete.

A „tömörített” gráf tárigénye $\Theta(VM)$ ahol V az egyedi illesztési oszlopok (a gráf csúcsainak) száma. A megtakarítás az 1. algoritmus oszloponkénti csúcslistás ábrázolásához képest onnan származnak, hogy a listákat követőosztályonként egyszer tároljuk. A követőosztályok számának felső korlátja V , hiszen minden oszlopnak egyetlen követőosztálya van, valamint a követőlisták elemszámának összege szintén legfeljebb V , mert minden oszlopnak egyetlen megelőző osztálya van, amelynek a követő listája tartalmazza az adott oszlopot.

A gyors MPD számítást végző FASTMPDALIGN algoritmus futási ideje hasonlóan alacsony. Analóg módon a 3.3.2. szakaszhoz megmutatható, hogy a 19. sor követőosztályonként pontosan egyszer hajtódik végre, következésképp a 20–21. sorok egyszer futnak le minden osztály minden rákövetkező oszlopára, azaz röviden minden oszlopra, hiszen az osztályok rákövetkező oszlopai között nincs átfedés. Végül, a 18. és 27. sorok szintén $\Theta(V)$ -szer hajtódnak végre, továbbá mivel minden művelet $O(1)$ idejű és a 10–13. sorokban lévő ciklus $O(V)$ -szer fut le legrosszabb esetben, a FASTMPDALIGN algoritmus teljes futási ideje $\Theta(V)$.

3.5. További felhasználási lehetőségek

Az eddigiekben kizárólag egyszerű dinamikus programozási algoritmusokat mutattunk be, amelyek az illesztési gráfból egy optimális utat (illesztést) választanak ki. E tömör ábrázolásmódban azonban számos egyéb lehetőség rejlik.

Az előkészületek alatt álló munkánkban [4] részletesen tárgyaljuk, hogy miképpen terjeszthetők ki filogenetikai rejtett Markov-modell algoritmusok, amelyek hagyományosan egyetlen illesztést vesznek figyelembe, úgy, hogy a gráf által leírt, exponenciális számú lehetséges illesztésen hatékonyan átlagoljanak. A regulációs annotáció területén ez egy lényeges alternatívája a kombinált evolúciós–annotációs modellel dolgozó eljárásoknak [7], amelyek sokkal hosszabb mintavételezési időt igényelnek.

Az illesztési gráfok felhasználhatósága nem ér itt véget. Bonyolultabb bioinformatikai algoritmusok, például a sztochasztikus környezetfüggetlen nyelvtanok (SCFG) belső–külső (inside–outside) algoritmusai is adaptálható illesztési gráfokra.

4. Eredmények

A következőkben röviden összefoglaljuk a bemutatott konszenzus illesztési módszer értékelésének eredményeit. Részletesebben tárgyalja az eredményeket a [5] munka.

4.1. Illesztések összevetése

A többszörös illesztések pontosságát egy referencia illesztéshez viszonyítva számos mérőszámmal jellemezhetjük. A BALIBASE adatbázis készítői [8] által adott SP (sum of pairs) mérőszám, amire *bali* értéként fogunk hivatkozni, a helyes karakter–karakter illesztések v. homológia-állítások arányát méri. Értéke a helyesen illesztett karakter–karakter párok száma, H^* , osztva a referencia illesztésben lévő összes karakter–karakter párok számával, H -val. Ez a mérőszám nem veszi figyelembe a helyes beszúrás karaktereket.

Egy sokkal kiegyensúlyozott módszert készítettek Bradley és társai [1], amely a nem-homológia állításokat (karakter–beszúrás párokat) is vizsgálja. Legyen N a referencia illesztésben lévő karakter–beszúrás párok száma és N^* a helyesen illesztett karakter–beszúrás párok száma. Az *fsa* értéket ekkor a következőképp számoljuk: $(2H^* + N^*)/(2H + N)$. A homológia állítások látszólagos túlsúlyozásának az oka, hogy ezek két karaktert tartalmaznak a bemeneti szekvenciákból, míg a nem-homológia állítások csak egyet. Másképp szemlélve, $(2H + N)$ -ben az összes szekvencia minden karaktere pontosan annyszor fordul elő, ahány más karakterhez/beszúráshoz illesztve van, ami a szekvenciák számánál eggyel kevesebb. Így könnyen belátható, hogy ez az érték adott illesztéstől független, és csak a szekvenciák számától és hosszától függ, ezért az *fsa* érték független attól, hogy melyik illesztést választjuk referenciának és melyiket tesztnek.

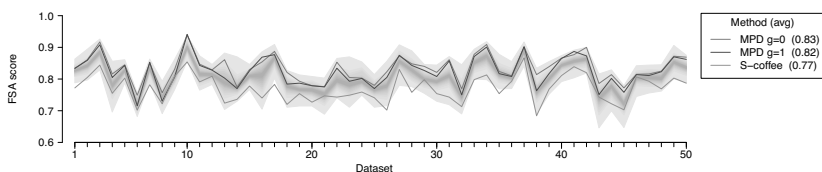
A teljesség kedvéért egy harmadik mérőszámot is bevezetünk, amely a *bali* érték „ellenkezője”. Az *nhom* értéket (N^*/N) -ként definiáljuk, ami tehát a helyes nem-homológia állítások (karakter–beszúrás párok) arányát adja meg.

4.2. Eredmények szimulált illesztéseken

A konszenzus illesztő módszereink értékeléséhez a DAWG nevű, biológiai szekvencia-evolúciót szimuláló eszközzel [2] állítottunk elő referencia illesztéseket. Ez az eljárás a STATALIGN-nál valóságosabb beszúrás–törlés modellt alkalmaz, amelyből statisztikailag mintavételezni nehéz, viszont a pontosság értékelésére jól alkalmazható.

Véletlenszerűen választottunk egy 10 levelű törzsfát, amelyen a DAWG eszközzel szekvenciákat szimuláltunk a GTR szubsztitúciós modellel (G+I rátaheterogenitással) és negatív binomiális indelhossz-

eloszlással. 50 referencia illesztést állítottunk elő, és minden esetben a STATALIGN v1.1 programmal 1000 illesztést mintavételeztünk az illesztési térből. Az illesztésmintákat ezután az MPD algoritmussal és a T-COFFEE konszenzus illesztőjével [10] összegeztük, és a konszenzus illesztés pontosságát összevetettük a minták pontosságával. Az egyes minták és a konszenzus illesztések pontosságát is a kiegyensúlyozott *fsa* értékkel jellemeztük, a referencia illesztéshez összehasonlítva.



4. ábra. Konszenzus illesztések *fsa* pontossága az MPD algoritmussal ($g = 0$, $g = 1$) és a T-COFFEE konszenzus illesztőjével (S-coffee) 50 különböző szimulált adathalmazon, minden esetben 1000 darab STATALIGN-nal mintavételezett illesztésre alkalmazva. Az adathalmazokat a DAWG szekvencia evolúciós szimulátorral állítottuk elő, rögzített 10-szekvenciás törzsfán, G+I rátaheterogenitással. A szürke sáv az egyedi illesztési minták pontosságának határait jelöli, a pontossági értékek mediánját sötétszürkével emeltük ki.

A eredmények alapján elmondhatjuk, hogy az MPD algoritmussal előállított konszenzus illesztések általában pontosságukban közelítik a legpontosabb illesztési mintát (ld. 4. ábra). Ezzel szemben a T-COFFEE konszenzus illesztései általában a minták pontosságának mediánja alatti értéket képviselnek. Az MPD konszenzus átlagos pontossága $g = 0$ -ra 0.83, ami 0.06-tal magasabb a T-COFFEE átlagos pontosságánál. Ez a különbség első látásra nem tűnik lényegesnek, de figyelembe véve, hogy ez a minták pontossági szórásának 3.2-szerese (0.018), ill. 60%-a a legmagasabb és legalacsonyabb pontosság közti átlagos különbségnek (0.11), az eltérés igen jelentős. A g paraméter értékének ebben a vizsgálatban nem volt jelentős hatása.

4.3. Eredmények valódi fehérjeszekvenciákon

Terjedelmi okokból a további eredményeinket nem részletezzük, de összefoglaljuk a legfontosabb megállapításokat:

- Az MPD algoritmus ismert fehérjeszekvenciák illesztésein mérve is általában pontosabb konszenzust állít elő, mint a T-COFFEE és a MERGEALIGN eljárások.
- Valódi szekvenciákon a pontosság érzékenyebb a g paraméter értékére: konzerváltabb, kevésbé diverz szekvenciák esetében a magasabb, 1 közeli g értékek adtak jobb eredményeket, míg nagyon eltérő szekvenciák esetében az alacsony, 0 közeli g értékek.
- A pontosság nem objektív fogalom: a *bali* mérőszámot véve az MPD eljárás $g = 1$ paraméterrel relatíve jobban teljesít (és fordítva, az *nhom* mérőszámmal $g = 0$ teljesít jobban), ami logikus következménye a mérőszámok definíciójának és annak, hogy $g = 1$ rövidebb illesztéseket állít elő, minimalizálva a fals pozitív (FP) oszlopok számát.

5. Összefoglalás és kitekintés

E tanulmányban bemutattunk egy új adatszerkezetet, amellyel adott szekvenciák nagyszámú illesztése ábrázolható tömören az illesztési oszlopok egy irányított, körmentes gráfján. Megadtunk továbbá egy algoritmust, amely a gráf felépítését optimális, $O(S)$ idő alatt elvégzi, ahol $S = LM$ a bemeneti illesztések teljes mérete. A takarékos reprezentáció $O(VM)$ tárigényű, ahol V az egyedi oszlopok száma a gráfban.

A módszer lehetővé teszi konszenzus illesztések előállítását dinamikus programozással $O(V)$ idő alatt. Megfelelő veszteségfüggvényt választva az MPD eljárásban a kívánt relatív fals pozitív–fals negatív ráta egy paraméteren keresztül állítható. Megmutattuk, hogy az MPD eljárással előállított konszenzus illesztés az összegzett illesztési minták közül a legpontosabbat közelíti, és általában pontosabb a korábbi, ismert konszenzus módszereknél.

A konstrukció szélesebb körben is felhasználható: megemlítettük, hogy bioinformatikai algoritmusok különböző osztályai hasonló adatszerkezetek segítségével kiterjeszthetők úgy, hogy egyetlen illesztés helyett a lehetséges illesztések egy halmazán hatékonyan átlagoljanak, ezáltal pontosítva az eredményeiket és számszerűsítve az illesztésből származó hibát. E módon általánosíthatóak például a rejtett Markov-modell

algoritmusok, amelyeket előszeretettel használnak génfelismerésre és -annotálásra, illetve a sztochasztikus környezetfüggetlen nyelvtan algoritmusok, amelyeket például térszerkezet-becslésre alkalmaznak.

Hivatkozások

- [1] R. K. Bradley, A. Roberts, M. Smoot, S. Juvekar, J. Do, C. Dewey, I. Holmes, L. Pachter, Fast statistical alignment, *PLoS Comput Biol*, 5(5):e1000392+ (2009).
- [2] R. A. Cartwright, DNA assembly with gaps (Dawg): simulating sequence evolution, *Bioinformatics*, 21(Suppl 3):iii31–iii38 (2005).
- [3] P. J. Green, K. V. Mardia, Bayesian alignment using hierarchical models, with applications in protein bioinformatics, *Biometrika*, 93(2) (2006), pp. 235–254.
- [4] J. Herman, Á. Novák, J. Hein, Incorporating alignment uncertainty into sequence annotation, *In prepration*, 2014.
- [5] J. Herman, Á. Novák, I. Miklós, R. Lyngsø, J. Hein, Efficient representation of uncertainty in multiple sequence alignments using directed acyclic graphs, *IEEE Transactions on Computational Biology and Bioinformatics*, 2014, elbírálás alatt.
- [6] Á. Novák, I. Miklós, R. Lyngsø, J. Hein, StatAlign: An Extendable Software Package for Joint Bayesian Estimation of Alignments and Evolutionary Trees, *Bioinformatics*, 24(20) (2008), pp. 2403–2404.
- [7] R. Satija, Á. Novák, I. Miklós, R. Lyngsø, J. Hein, BigFoot: Bayesian alignment and phylogenetic footprinting with MCMC, *BMC Evolutionary Biology*, 9(1) (2009), pp. 217+.
- [8] J. D. Thompson, F. Plewniak, O. Poch, BALiBASE: a benchmark alignment database for the evaluation of multiple alignment programs, *Bioinformatics*, 15(1) (1999), pp. 87–88.
- [9] J. L. Thorne, H. Kishino, J. Felsenstein, An evolutionary model for maximum likelihood alignment of DNA sequences, *Journal of Molecular Evolution*, 33 (1991), pp. 114–124.

- [10] I. M. Wallace, O. O'Sullivan, D. G. Higgins, C. Notredame, M-Coffee: combining multiple sequence alignment methods with T-Coffee, *Nucleic Acids Research*, 34(6) (2006), pp. 1692–1699.

Rendszerszintű fejlesztés funkcionális nyelven

Páli Gábor János*

Eötvös Loránd Tudományegyetem, Informatikai Kar

`pgj@elte.hu`

A funkcionális nyelvek az ipari alkalmazások területén is egyre jobban előretör, egyre több alkalmazásra lel. Ez alól a rendszerszintű fejlesztések sem képeznek kivételt. Napjaink felhőszolgáltatása olyan rendszereken futnak, amelyeket eredetileg nem erre a célra terveztek. Tele vannak olyan korlátozásokkal és tervezési döntésekkel, amelyek olykor feleslegesen bonyolítják és egyúttal teszik sebezhetővé a rájuk épülő rendszereket. Mindemellett a felhő alapú szolgáltatások létrehozásakor az utóbbi időben elterjedt az a megoldás, hogy az egészet kisebb, egy-egy célra finomhangolt virtuális gépek hálózataként alakítják ki. Ennek forradalmasítására vállalkozott a „Mirage OS”, amely egy OCamlben írt könyvtárak összességéként megvalósított, nyílt forráskódú kísérleti operációs rendszer. A Mirage biztonságos, nagy teljesítményű hálózati fejlesztésre kínál megoldást. Segítségével az alkalmazások kódja tetszőleges POSIX operációs rendszer felett fejleszthető és tesztelhető, majd egyetlen kapcsoló átállításával mindez lefordítható egy – napjaink egyik közkedvelt felhőtechnológiája – Xen felett futó, specializált mikrokernellé. Mivel a Xen számos felhőszolgáltatás alapját képezi, ezért a Mirage lényegében egy egyszerű, az eddigi megoldásoknál biztonságosabb és jobban kezelhető alternatívát valósít meg. Ezt és ennek fejlesztés alatt álló FreeBSD portját, illetve rajta keresztül az implementációs kihívásokat, mutatjuk be röviden ebben az írásban.

*University of Cambridge, Computer Laboratory, `gabor.pali@cl.cam.ac.uk`

1. Bevezetés

A Mirage biztonságos, nagyteljesítményű, felhő és mobil platformokon futó hálózati alkalmazások fejlesztésére alkalmas kísérleti operációs rendszer [1], amelyet a Cambridge University Computer Laboratory-ban fejlesztenek, valamint 2013 októbere óta a Xen Project és a Linux Foundation hivatalos pártfogása alatt áll. A projekt célkitűzése, hogy a gyakorlatban is alkalmazható, hatékony és könnyen kezelhető eszközt ajánljon fel mindazoknak, akik szeretnének megbízható módon specializált operációs rendszereket készíteni illetve felhőszolgáltatásokat építeni.

A projekt komolyságát mutatja egyébként, hogy a saját honlapja [2] is egy ilyen alkalmazásként fut, rendelkezik egy komplett (IPv4) TCP/IP hálózati stackkel, amelynek felhasználásával a fejlesztői készítették vele teljes értékű HTTP, SSH és DNS szervereket is, a C implementáció méretének mintegy töredékéből.

1.1. Az alkalmazás mint operációs rendszer

A rendszer alapvetően exokerneles [3] felépítésű, amelynek lényege, hogy lehetőleg minél kevesebb absztrakciót kényszerítsenek a fejlesztőkre, ezáltal lehetőség nyílik szinte teljesen saját megközelítésükben felépíteni az alkalmazásaikon keresztül az operációs rendszert. Ezért az ilyen típusú megoldások leginkább csak egyszerűbb kényelmi szolgáltatásokat nyújtanak, például a rendszerben fellelhető erőforrások elérését teszik lehetővé, de az hozzájuk kapcsolódó protollokat illetően az alkalmazás szabad kezet kap. Gyakran hivatkozzák emiatt az ilyen módon elkészített alkalmazásokat "library operating system" [4] néven is.

A fejlesztés során az alkalmazások kódja tetszőleges POSIX kompatibilis operációs rendszeren, például FreeBSD, Mac OS X vagy Linux alatt megírható és kipróbálható, amely aztán a futtató rendszer lecserélésével lefordítható egy önálló, specializált miniatűr operációs rendszerré. Ilyen miniatűr rendszer lehet például egy Xen hypervisor [5] felett futtatható mikrokernél, de a megoldás absztrakt felépítésének köszönhetően tulajdonképpen sokféle formában előállítható.

A Xen választását ebben az esetben valószínűleg az indokolja, hogy ez szintén a Cambridge-i Egyetemről került ki, és az ott dolgozó kutatók a lehetséges alternatívák közül ezt ismerik a legjobban. Emellett

a Xen definiál egy kényelmes hardverfüggetlen interfészt, amelyre anélkül tudunk alkalmazásokat illeszteni, hogy el kellene vesznünk a hardverek megszólításával és zökkenőmentes üzemeltetésével kapcsolatos problémákban. Másrészt a Xen napjainkban egy igen népszerű ipari virtualizációs megoldássá nőtte ki magát, ezért számos felhőszolgáltatás megvalósításának alapját képezi.

Amikor viszont ilyen virtualizált alapokra építünk szolgáltatásokat, akkor akaratlanul is további rétegeket halmozunk fel a rendszer felépítése során: szükségünk van egy hagyományos, általános célú operációs rendszerre, különböző (DNS, SSH, HTTP, stb.) szerverekre, esetleg ezek felett további bővítési lehetőségekre (Python, PHP, Ruby, stb.), szkriptelési lehetőségekre és így tovább. Ez a ún. "LAMP" ("Linux-Apache-MySQL-PHP") konfiguráció, amely ugyan sokak számára könnyen kezelhető, ám a megvalósítás szemszögéből jelentős pazarlás és bonyolítás egyszerre, amely jelentős biztonsági kockázatokkal jár [6].

2. Alkalmazások fejlesztése

A Mirage kidolgozóinak és kutatóinak egyik alapvetése, hogy valamilyen módon csökkentenünk kell tudnunk rendszerünk rétegezettségét és bonyolultságát is, miközben megtartjuk annak rugalmasságát, javítjuk a teljesítményét és megbízhatóságát. Ezen a ponton lép be a funkcionális programozás, mivel ennek az alapját ebben a technológiai megoldásban az OCaml programozási nyelv [7] képzí. Az OCaml – vagy eredetileg Objective Caml – a Caml (Categorical Abstract Machine Language) nyelv referencia implementációja, amely az eredeti nyelvet objektumorientált eszközökkel terjeszti ki. Ez, jellegét tekintve a Standard ML (SML) nyelvhez hasonlít, de szoros kapcsolatban áll a F# nyelvvel, valamint jelentős hatással volt a Scala és Rust nyelvek megalkotására. Az iparban és a kutatásban egyaránt népszerű eszköz, nagyon hatékony programfejlesztést tesz lehetővé.

A funkcionális programozási paradigma ugyanis már a kezdetektől absztrakt, matematikai gondolkodás mentén épül fel, lehetővé téve olyan szintű specifikációk megfogalmazását és futtatását, amelyek erős garanciákat adnak helyes és összefüggő programok kidolgozására, ezáltal eleve megbízhatóbbak. Másrészt, absztrakt jellegüknél fogva ilyen típusú nyelvek mögött nagyon hatékony és intelligens futtató rendszerek és fordítóprogramok állnak, amelyek nagy mértékben tudják segíteni a

programozó munkáját. Továbbá ezekben a nyelvekben meglehetősen elfogadott módszer különböző, csupán egy adott szakterület fogalmaival dolgozó, nem általános célú programozási (al)nyelvek ("domain-specific language", DSL) létrehozása. Ennek köszönhetően egy magasabb szemantikai szinten fejezhetjük ki a programjainkat, így a fordítóprogram nagyobb optimalizációs szabadsággal rendelkezik, illetve a nyelv tervezői logikailag ki tudják zárni a helytelen programokat a fordításból. Az OCamlnek van egy saját, programozható előfeldolgozója, a Camlp4, amely egy önálló fordítóprogramnak is tekinthető, és segítségével szinte teljesen testreszabható az OCaml szintaxisa.

A funkcionális nyelvek világán belül többféle megközelítés létezik a fentebb említett tulajdonságok teljesítésére, amelyek értelemszerűen eltérő kompromisszumokkal járnak az implementáció és a kifejezési készség tekintetében. Ezek közül testesít meg egyet az OCaml, amely egy erős, statikus típusozású nyelv. Az erős típusozás itt azt jelenti, hogy a háttérben nem történik automatikus implicit konverzió az egyes alkalmazott típusok közt, hanem a programozónak mindig egyértelműen jeleznie és kérnie kell ezeket. Ezzel elkerülhetőek például az implicit konverzióból fakadó kellemetlen meglepetések, de egyúttal segítik a programozóban is tudatosítani az adatáramlást. A statikus típusozás pedig arra utal, hogy a fordításkor (tehát lényegében csupán a program szövegét nézve) tisztában kell lennünk azzal, hogy melyik ponton melyik változó milyen típusú értéket képvisel. Ez pedig azért hasznos, mert így a program szerkezete automatikusan ellenőrizhetővé válik és segít rávilágítani a benne rejlő hibákra. Talán nem is kell hozzátennünk, hogy egy fontos feladatot ellátó, komoly megbízhatósági elvárásokkal szembenező rendszer esetén ezek az előnyök tulajdonképpen felbecsülhetetlenek.

Ebben a paradigmában jellemző lehet még a mellékhatások erős elszigetelése és a végtelen adatszerkezetek támogatása is (ld. például a Haskell nyelvet), viszont ez jelentős mértékben el tudja bonyolítani a futtatáshoz szükséges támogatás megvalósítását. Az OCaml ezért ez utóbbi lehetőségekhez nem nyújt közvetlen támogatást, cserébe viszont egy egyszerű futtató rendszerrel rendelkezik, valamint a fordító által előállított kód sebessége sem marad el túlságosan a C nyelvű változatétól. A típusok állandósított jelenléte egyébként messze nem zavaró az erősen statikus típusos funkcionális nyelvekben, ugyanis a fordító képes magától kikövetkeztetni a programbeli nevekhez tartozó típusokat és csak az

olykor előforduló félreértések tisztázása esetén (vagy dokumentációs céllal) kényszerül a programozó ezeket megadni. Ennek köszönhetően képes felvenni a versenyt a dinamikus nyelvek megszokott kényelmével. Természetesen szükség esetén az OCaml kódrészleteket ki lehet egészíteni C nyelvű programrészekkel is a hatékonyság növelése érdekében.

2.1. Funkcionális hálózati programozás

A Mirage esetén tehát OCamlben írt funkcionális programok lesznek azok az alkalmazások, amelyeket aztán operációs rendszerként futtathatunk. Mögöttük természetesen egy komoly modulrendszer áll, ahol implementáltak többek közt a hálózati kapcsolatok kezelését, így nekünk, az alkalmazás fejlesztőjének ezeket csak hivatkozni kell. Maga a Mirage elnevezés valójában pontosan erre a komponenskészletre vonatkozik. Ez jelenleg nagyjából 40 különböző kisebb-nagyobb csomagot jelent, de ezek száma folyamatosan növekszik. Ezeknél a fejlesztők célja, hogy a komponensek minél jobban újrafelhasználhatóak legyenek, és mivel a fordítás során úgyszintén csak az aktuálisan használt részek kerülnek a lefordított binárisba, ez a gyakorlatban nem jelent többletköltséget.

Példaként tekintsük a következő kódrészletet a 1. ábrán! Ez egy egyszerű, TCP-n keresztül kommunikáló „visszhang” szolgáltatást valósítunk meg, amely a 8081 `tcp4` porton várja a klienseket és a tőlük kapott adaokat visszaírja nekik.

Közvetlenül az első sorban az `Lwt` modult nyitjuk meg, amely kooperatív ütemezésű szálak létrehozását valósítja meg. Ilyenkor minden szál törzse egy monadikus akció lesz, amely az `Lwt` monádban fog futni. Így a futtató rendszer a különböző blokkolást kiváltó hívások (pl. a futás felfüggesztésével járó `sleep`) mentén fel tudja darabolni a párhuzamosan futtatni kívánt programrészeket és ütemezni azokat.

OCamlben a `let` (és `lwt`) kulcsszó segítségével tudunk neveket, tehát változókat vagy függvényeket definiálni. Ekkor lényegében a definíciók sorrendjüknek megfelelően kiértékelődnek és létrehozzák a nekik megfelelő típusú objektumot. A felső szinten megadott `let` definíciók az egész modulban látszanak, viszont a bennük szereplő `let` definíciók csak az ezeket befoglaló egységben, ezért itt a nevek után még egy, hatáskört hozzákötő `in` kulcsszó megadása is kötelező.

Az alkalmazásunk belépési pontja (ld. lentebb) a `main` függvény lesz, amely paraméterként a futtató rendszertől megkapja a hálózati

```
open Lwt

let main mgr interface id =
  let src = None, 8081 in
  Net.Flow.listen mgr ('TCPv4 (src,
    (fun (addr, port) t ->
      let address = Ipaddr.V4.to_string addr in
      OS.Console.log (Printf.sprintf "From %s:%d\n" address port);
      let rec loop () =
        lwt res = Net.Flow.read t in
        match res with
        | None -> OS.Console.log "Connection closed\n";
          return ()
        | Some data -> Net.Flow.write t data >>=
          loop
      in loop ()
    )))
```

1. ábra. Egy visszhang szolgáltatás a Mirage eszközeivel.

kapcsolatot kezelő menedzser (`mgr`) referenciáját, valamint annak a hálózati interfésznek nevét és azonosítóját, ahonnan a konkrét hálózati adat érkezett. A Mirage a gépben elérhető minden hálózati interfészre külön rákapcsolódik, így ezeket párhuzamosan tudja kezelni.

Ezután látható egy másik fontos kulcsszó a `fun`, amely λ -függvények definícióját vezeti be. Ezek tulajdonképpen egy törzset rendelünk hozzá a `Flow.listen` függvényhez, amely megnyit egy TCP (v4) kapcsolatot a 8081-es porton, majd az adatok feldolgozását és generálását ezzel a törzssel írjuk le. Itt megkapjuk paraméterként a küldő címét és portját, valamint a kapcsolat állapotát. A címet az `Ipaddr.V4` modul segít feldolgozni, amit aztán az `OS.Console.log` függvény ki is írunk a konzolra. A cím és a port alapján végül beolvassuk az adatot a küldővel a háttérben kialakított TCP-csatornán keresztül a `Net.Flow.read` függvény meghívásával. Ezután a konkrét eredmény alapján tudunk arról dönteni mintaillesztés (`match .. with`) segítségével, hogy valóban kaptunk-e adatot, vagy sem. Amennyiben nincs már több adat (vagyis `None` értéket kaptunk vissza), akkor befejeződik a feldolgozás, ha viszont érkezett valamennyi (`Some`) adat, akkor a mintaillesztés segítségével kiemeljük, és a `Net.Flow.write` függvénnyel visszaírjuk a küldőnek. Utána pedig a `loop` belső függvény függvény saját magát hívja meg,

miközben egy monadikus `>>=` (bind) operátorral egyúttal átadja a belső állapotot.

Miután megírtuk az alkalmazás kódját, készítenünk kell még hozzá egy konfigurációs állományt. Ezt a Mirage egyik segédprogramja, a **mirari** fogja feldolgozni és segítségével le tudjuk gyártani az adott platformra szánt binárist. A **mirari** célja, hogy a fordítást és a beállítások átadását függetlenné tegye az egyes platformoktól.

Az iménti szolgáltatásunkhoz a 2. ábrán látható konfigurációs állományt adtuk meg. Ez tartalmazza az létrehozandó példány IP-címét (statikus vagy dinamikus, esetleg nincs), a tárolt állományokat, a belépési pontot és a függőségeket. Az IP-cím magától értetődő: mivel a miniatűr rendszereinket hálózatba akarjuk kapcsolni, ezért nyilatkoznunk kell arról, hogy pontosan miként is kapjanak logikai hálózati címet. Ha nem akarjuk megadni, akkor választhatjuk a dinamikus címzet. Ilyenkor a fizikai hálózaton keresztül kapja meg minden miniatűr rendszer a címet szabványos DHCP kéréseken keresztül.

A binárisunk igény szerint tartalmazhat állományokat is, amelyeket a fordítás során összeszerkesztődnek a programkóddal és a betöltéskor vele együtt bekerülnek a memóriába. Ennek előnye, hogy gyorsan elérhetőek, viszont statikusak, tehát a hagyományos értelemben nem állományként látjuk majd ezeket. Ammenyiben valóban állományrendszerre van szükségünk, úgy a Mirage megfelelő komponensét kell hozzá használnunk. Ezen a téren még nincs akkora támogatottsága, de már megengedi `.vhd` kiterjesztésű (XenServer és Hyper-V) állományok, valamint szabványos UNIX blokkeszközök használatát is.

Ezután például a kFreeBSD platformra (ld. később) csupán ennyi paranccsal le is tudjuk fordítani, majd futtatni az alkalmazást:

```
$ mirari configure --kfreebsd echo.conf
$ mirari build --kfreebsd echo.conf
$ mirari run --kfreebsd echo.conf
```

Xen vagy a UNIX backend használatához egyszerűen csak a `--kfreebsd` kapcsolót kell a `--xen` vagy a `--unix` értékre lecserélni.

3. Röviden az implementációról

Ahogy már az előbb is láthattuk, miután elkészítettünk az alkalmazásunkat, többféle módon tudunk fordítani. Az egész módszer lényege,

```
# IP configuration (optional)
# ip-use-dhcp: true
# Static IP (optional)
ip-address: 10.0.2.235
ip-netmask: 255.255.255.0
ip-gateway: 10.0.2.2

# File system (optional)
# fs-static: ../fs

# Main function
main-ip: Echo.main

# Dependencies
depends: mirage-net, ipaddr
packages: mirage, mirage-net, ipaddr
```

2. ábra. A visszhang szolgáltatás konfigurációs állománya.

hogy az OCaml programokat nem fordítjuk le teljesen, hanem csak egy futtató rendszer nélküli tárgykódot állítunk elő, tehát a programunk még linkelés előtt álló változatát. Az OCaml fordító ilyenkor kiválogatja a programunkban hivatkozott modulokat és függvényeket és csak azokat teszi bele a tárgykódba, amelyeket valóban használunk is. Ennek köszönhető az, hogy az alkalmazásunk a Mirage beépített elemei közül mindig csak szükségesek kerülnek bele a mikrokernelbe.

Ezt egyébként a következő módon érhetjük el:

```
$ ocamlpt -output-obj -o app.o echo.ml
```

Ezt követően már csak annyi a teendőnk, hogy kiválasszuk a megfelelő backendet, vagyis a platformot, ahol futtatni akarjuk a programot. Ennek különböző szintjei lehetnek:

- A Xen virtuális interfészére támaszkodva, paravirtualizált kernelként futtatjuk. Ilyenkor a működéshez szükséges elemeket, mint például a hardvereszközök vagy a memória elérést javarészt egy leegyszerűsített rendszerrel, a "Xen MiniOS" [8] egy módosított változatának felhasználásával érjük el. Ilyen az alkalmazásunk a komplett memóriát egyben, 64 biten éri el. (Ezzel lentebb még foglalkozunk hamarosan.)

- Egy meglevő POSIX rendszer alrendszerére támaszkodunk, és az Ethernet forgalmat TUN/TAP alkalmazásával megcsapoljuk. Ilyenkor a hálózati kártyát állományhoz hasonlóan kezeljük, ahova írva Ethernet kereteket tudunk küldeni, olvasva pedig fogadni. Ennek nyilván a közvetlen Xen feletti futáshoz képest nagyobbak a költségei, azonban az alkalmazás viselkedése így könnyebben is nyomomonkövethető.
- Egy POSIX rendszer hálózati alrendszerére támaszkodunk, és a kommunikációt egy TCP/UDP socketen keresztül bonyolítjuk le. Ez az előbbinél egy lassabb módszer, szintén a kezdeti prototípus elkészítésekor érdemes leginkább használni.

Természetesen ezen kívül még további backendek is léteznek, a Mirage futtatható akár WebSockets felett az `ocamljs` fordítónak köszönhetően, de akár a Google AppEngine vagy Android platformokra is telepíthető az `'ocamljava'` és az OCaml fordító ARM backendjén keresztül. Bizonyára érezhető, hogy egy magasabb szintű programozási nyelv ilyenkor sokkal könnyebben kezelhető, hiszen a fejlesztőnek ekkor nem kell az egyes platformok eltéréseivel foglalkozni, hanem elegendő csupán a programlogikát leírnia, és a megfelelő modulok elvégzik a feladat többi részét.

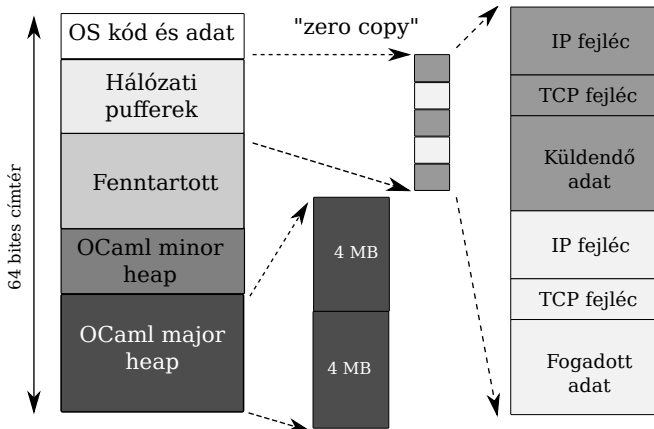
3.1. Futtatás Xen felett

Mielőtt azonban még továbbhaladnánk, egy kicsit térjünk vissza a natív, vagyis a Xen feletti futtatáshoz! Azért érdemes erre még időt és helyet szánnunk, mert az itt kialakított rendszer igyekszik minél jobban idomulni az OCaml futtató rendszer igényeihez és működéséhez. Ez leginkább a memóriakezelésben mutatkozik meg, mivel hasonlóan a többi funkcionális nyelvhez, itt is automatikus szemétgyűjtögetés történik a program futtatása során. Ez pedig egy operációs rendszer szerves részeként alapjaiban meghatározza annak teljesítményét.

Az ezzel kapcsolatos hiedelmek és panaszok ellenére ez alapvetően egy remekül működő módszer, amellyel a programozót meg lehet szabadítani a az erőforráskezeléssel járó problémáktól és ezáltal kiküszöbölni az ebből fakadó hibalehetőségeket. Fontos azonban hozzátenni, hogy ez funkcionális nyelvekben azért működhet valóban, mert a programokban a memóriakezelés teljesen implicit marad, a programozónak nem kell

tudnia annak jelenlétéről, hiszen az adatszerkezeteket is absztrakt matematikai eszközök, algebrai adattípusok segítségével építheti fel.

Az OCaml személggyűjtőgetője egyébként az adatokat két osztályba sorolja: a rövidtávú információkat az ún. *minor heap*be rendezi, a hosszútávúakat pedig a *major heap*be. Emiatt a minor heap tulajdonképpen a verem fogalmához áll közelebb, viszont jóval általánosabb, és szinte teljesen automatikusan, a felhasználás és viselkedés alapján kerül eldöntésre, hogy melyik objektumok landolnak ott. Emellett a major heap pedig nagy lépésekben, akár 4 MB-os superpage-enként is képes bővülni, ezzel mérsékelve a működéshez szükséges laptáblák méretét, amely így javulást eredményez a teljesítményben is.



3. ábra. A Mirage memóriakezelése Xen felett.

A másik fontos tényező, hogy a hálózati (és általánosságban az I/O) műveleteknek is minél kevesebb költséggel kell járniuk. Erre a Mirage az "IO page" nevű absztrakciót használja, amely tulajdonképpen a külvilággal történő kommunikáció alapköve. Egy ilyen page egy OCaml tömböt rejt maga mögött, amely tetszőleges memóriabeli helyre mutathat. Ezt természetesen az alkalmazás fejlesztője nem tudja módosítani, úgy látja, mintha az valamilyen programbeli számítás eredménye lenne. Ez lehet memóriába leképezett I/O terület, vagy akár a hálózati kártya által a memóriában valahol elhelyezett adat.

Az OCaml tömbökből pedig lehet kérni különböző nézeteket,

altömböket, amelyek az eredeti tömb valamelyik szeletét ábrázolják – szintén tömbként. Ez a tulajdonság kapóra jöhet akkor, amikor például hálózatról érkező kereteket akarunk feldolgozni, hiszen így folyamatosan el tudjuk hagyni a csomag már feldolgozott részeit, miközben a háttérben valóban csak egy mutatót állítgatunk. Az IO page-ek emiatt alkalmasak a C nyelvi implementációban megjelenő (FreeBSD rendszereken `mbuf(9)`, Linux alatt `sk_buff`) adatszerkezet működésének utánzására, vagyis általuk egy ún. másolás nélküli ("zero copy") csomagfeldolgozást tudunk megvalósítani.

Ahogy korábban már említettük, mindezek mellett lehetőségünk van szálak használatára is az alkalmazásokban, amelyek segítségével logikai szinten fel tudjuk osztani a programot különböző párhuzamosan futó folyamatra. A szálak létrehozását és kezelését az 'lwt' (mint "light-weight threading") nevű, szintén OCaml nyelven írt könyvtár végzi [9]. Ez a könyvtár kooperatív szálkezelést valósít meg, vagyis nem található benne dedikált ütemező, a szálak egymásnak önkéntesen adják át a vezérlést, például akkor, amikor várakoznak valamilyen adatra. Az szálak törzsét egyébként ilyenkor monadikus blokkok formájában fogalmazzuk meg, amelyeket futás közben össze tudunk fésülni más hasonló blokkokkal. Noha a monádok, a monadikus programozás inkább a Haskell funkcionális nyelvben terjedt el, OCamlben is gyakran alkalmazzák, hogy tisztább, matematikailag jobban kezelhető programokat kapjanak. Ez a típusú szálkezelés jobban kiszámítható, ám nem jelent túlzottan nagy mértékű visszaesést a teljesítményben. Például ilyen módon valósították meg a projekt saját honlapját kiszolgáló HTTP szervert, a `cohttp`-t is.

Az iméntiekből lesűrhető, hogy igazából egyetlen Mirage kernel sem valósít meg igazi párhuzamosságot, amikor a szálak ténylegesen egymással egy időben futnak. Ennek a kérdését ismét a gazdakörnyezetre, vagyis jelen esetben a Xenre bízva, és ha erre van szükségünk, akkor egyszerűen csak több ilyen rendszert kell elindítanunk felette. Ekkor a különböző Mirage példányok osztott memória vagy adatbázisok (pl. XenStore) segítségével tudnak kommunikálni egymás között. Ez a megközelítés viszont talán annyiban hasznos, hogy így az egyes funkciók teljesen elszigetelhetőek egymástól, a rendszer egyes részei akár külön újra is indíthatóak.

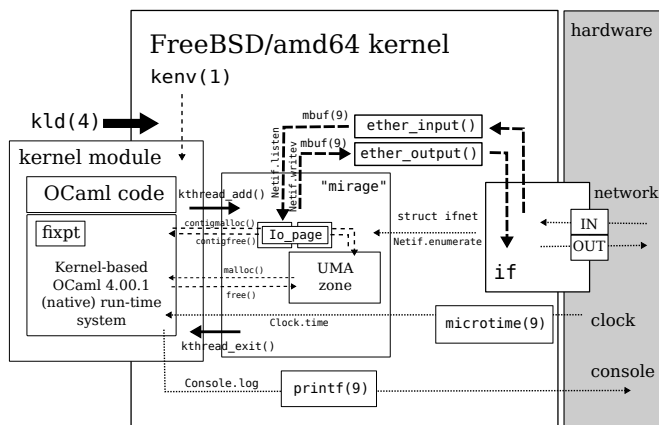
4. Mirage/kFreeBSD

A korábban felsoroltak mellett a Mirage célplatformja lehet még a FreeBSD operációs rendszer [10] kernelje, amely a szerző a projektben történt részvételének eredménye. Hasonlóan a Xenhez, a Mirage fejlesztői mellett több FreeBSD fejlesztő is a Computer Laboratory munkatársaként dolgozik, amelynek következtében felmerült a lehetőség hogy a két technológiai irány egyesítsék egy kísérletben. A FreeBSD mint az egykori egyetemi UNIX, a BSD reinkarnációja, remek táptalaj az operációs rendszerek témakörében végzett tudományos kutatásoknak. Mindezek mellett a FreeBSD egy nyílt forráskódú operációs rendszer, amelyet egy bár relatíve kis méretű, de nagyon összetartó és profi közösség gondoz, ezáltal könnyebb megismerni és hozzájárulni.

A kísérlet elsődleges célja az, hogy a Mirage ne csak Xen, hanem a közvetlenül a FreeBSD kernelje felett is tudjon futni. Ezt úgy oldjuk meg, hogy az alkalmazásokból kernelmodulokat készítünk, ezáltal a kernel címtérben és privilégium szintjén fog futni az alkalmazás. Természetesen mint hagyományos POSIX rendszer, a FreeBSD felett egyszerű OCaml alkalmazásként TUN/TAP vagy socketek segítségével enélkül is tudunk Mirage programokat futtatni, itt a cél most ezek kernelbe történő beemelése. Az így előállított kernelmodulok magasszintű viselkedését a 4. ábra foglalja össze. Ahogy látható, a megoldás csupán 64 bites rendszerrel működik. Ennek oka az, hogy az OCaml az egész számokat technikai okokból a megszokottnál eggyel kevesebb biten ábrázolja, és így egy 32 bites rendszeren nem garantálható a megbízható működés. De ez nem tekinthető akkora megszorításnak, mivel a 64 bites processzorok napjainkban már szinte mindenhol megtalálhatóak.

4.1. Futtatás a FreeBSD kernelében

A működés megvalósításához először fel kell építeni a kapcsolatot a rendszerben található hálózati eszközök és az OCaml kód között. A FreeBSD kernel ugyanis tartalmazza a megfelelő meghajtókat, amelyek a hálózatról érkező adatokat `mbuf(9)` formájában egy érkezési sorba pakolja, valamint hasonló stílusban továbbítja egy küldési sorból. Ezek eléréséhez viszont elsőként meg kell találnunk a rendszerben fellelhető hálózati interfészeket, amelyekre aztán a betöltött kernelmodul rá tud telepíteni és a C hálózati stack elől kiemelni az Ethernet keretek



4. ábra. A Mirage/kFreeBSD backend felépítése.

tartalmát.

Erre az `ether_input()` függvényen keresztül van lehetőségünk, amely minden beérkező keret esetén meghívódik a meghajtó és a keret adataival paraméterezetten. Hasonló módon az `ether_output()` függvény alkalmas arra, hogy rajta keresztül szabályosan megszerkesztett `mbuf(9)` adatokat küldjünk. Erre jelenleg a `netgraph(4)` alrendszer használjuk, de léteznek erre más lehetőségek is.

A többi funkció esetében csupán arra volt szükség, hogy a modulhoz kapcsolt OCaml futtató rendszer tudja használni a kernel által felkínált `malloc()` és `free()` függvényeket, valamint üzeneteket megjeleníteni a konzolon és elérni a rendszerórát. (Ne feledjük, ilyenkor lényegében közvetlenül maga a kernel lesz az OCaml kód futtató környezete!) Hogy a betöltött modul a rendszer többi részével együtt, a háttérben tudjon futni, a futtatandó kód egy megfelelően létrehozott kernelszállra kerül (`kthread_add()` és `kthread_exit()` függvények).

Az implementációval kapcsolatban érdemes hozzátenni, hogy a legtöbb erőfeszítést a megfelelő fordítási konfiguráció megtalálása és elkészítése jelentette. A FreeBSD kernel ugyanis igen szigorúan bántik mind a betölthető modulok fajtaival, mind a Floating-Point Unit (FPU) használatával: a kernelmodulok nem tartalmazhatnak pozíciófüggetlen kódokat (Position-Independent Code, PIC) és az FPU használata kernel oldalról nem támogatott. Ez utóbbi különösen azért fontos, mert az

OCaml alkalmazások a lebegőpontos műveleteket tulajdonképpen már a futtató rendszer szintjén adottnak tekintik. Az OCaml alapkönyvtárában ugyanis az időt valós számként ábrázolják, illetve a szemétgyűjtőgető is lebegőpontos értékek felhasználásával határozza meg a működéséhez szükséges paramétereket. Ez a Xen backend esetén nem okoz problémát, ugyanis a mikrokernelnek nem kell egy nagyobb kernelhez, a Xen gazdarendszer ezeket teljesen elszigetelti tőle. FreeBSD esetén azonban ez lesz a gazda (tehát nem a Xen felett paravirtualizálunk), így annak szabályainak megfelelően kell létrejönnie és viselkednie.

Az imént összefoglalt elvárások kielégítéséhez esetenként magát az OCaml fordítót és a futtató rendszert is módosítani kellett. Ebben nagy segítséget jelentett az OCamlhez, illetve a Mirage fejlesztéséhez kialakított, OPAM nevű eszköz [11]. Az OPAM alapvetően egy csomagkezelő, amellyel tetszőleges OCaml könyvtárat vagy alkalmazást, vagy éppen például a Mirage komponenseit tudjuk könnyűszerrel telepíteni. De remekül alkalmazkodik a fejlesztői igényekhez és segítségével a komponensek fejlesztői változatait is tudjuk menedzselni, valamint lehetőséget teremt a fordító több változatának párhuzamos használatára.

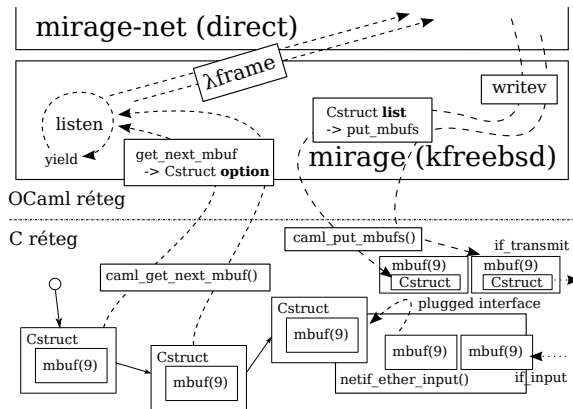
A lebegőpontos korlátozások megkerüléséhez a futtató rendszerbe egy fixpontos ábrázolást megvalósító, apró könyvtár – a `fixpt` – került importálásra. Ennek célja, hogy a valós számok ábrázolását és a velük kapcsolatos műveleteket 64 bites egészekre képi el. Ennek köszönhetően a kernel lényegében csak egész számokat lát, nincs szüksége az FPU segítségére. Előnye, hogy nagyon gyors számolásokat tesz lehetővé, illetve megtartható általa a többi célplatformmal közös interfész. Hátránya viszont, hogy kevésbé pontos ábrázolást tesz lehetővé, így komolyabb számításokra egyáltalán nem javasolt. Azonban a korábban említett feladatokhoz (idő ábrázolása, a szemétgyűjtőgető működtetése) ez tökéletesen elegendőnek bizonyult.

A kernelmodulok a FreeBSD-ben, hasonlóan más rendszerekhez, futás közben kidobhatóak és betölthetőek, ezt a `kld(4)` programozói interfész teszi lehetővé. (Alapvetően a FreeBSD kernel teljesen moduláris felépítésű, azonban bizonyos modulok – érthető okokból – nem kezelhetőek dinamikusan.) A moduláris jelleg miatt az egyes alkalmazásaink a fordítás után közvetlenül telepíthetőek is, vagy frissítés után cserélhetőek. Ehhez viszont a memóriakezeléssel kapcsolatban még orvosolni kellett az OCaml futtató rendszer egyik (kényelmi) hiányosságát. Felhasználói programként a szemétgyűjtőgető a program befejezésekor

ugyanis effektíve semmit sem csinál, hiszen kilépéskor az általa futás közben lefoglalt területek automatikusan visszakерülnek az operációs rendszerhez. Kernelmodulok esetén ez azonban nem teljesül, amely a modul cseréje során a memória fokozatos szivárgásához vezet. Ezért ki kellett egészíteni a futtató rendszert az összes használt memóriaterület szabályos felszabadításával, illetve ehhez megtalálni, hogy ezek közül mikor melyiket foglalja le.

A modulok által felhasználható maximális memória méretét is tudjuk továbbá korlátozni (hasonlóan egyébként a Xen mikrokernelhez rendelt virtuális gépekéhez) a `kenv(1)` kernelszintű környezeti változók beállításával. Ezzel a gazdarendszert tudjuk megóvni attól, hogy kifogyjon a memóriából, illetve a Mirage példányokat kényszeríthetjük kicsit szorgosabb szemégyűjtésre.

4.2. A hálózati kommunikáció



5. ábra. A rétegek közti kommunikáció a Mirage/kFreeBSD backendben.

Másik fontos tényező a hálózatról érkező csomagok kezelése. Ahogy a 5. ábra mutatja, a modulok – szükségszerűen – egy C és egy OCaml nyelvű rétegből állnak össze. A C réteg feladata, hogy a hálózati interfészekre beérkező kereteket elfogja és továbbítsa az OCaml réteg felé. Ehhez a `netif_ethernet_input()` függvényt fűzzük fel a hálózati interfészek NetGraph csatlakozására, így minden beérkező keret esetén

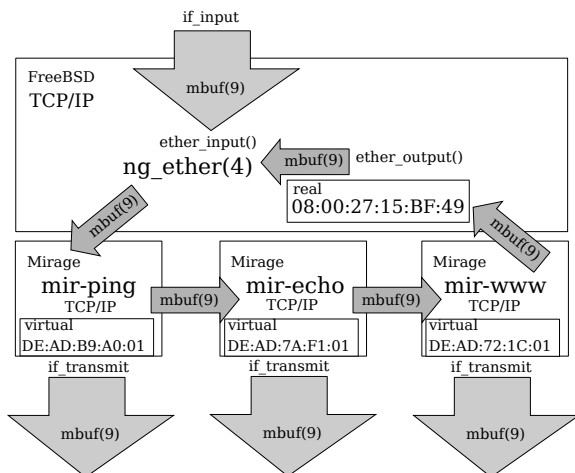
meghívódik. (Illetve ez nem teljesen igaz, erről részletesebben valamivel lentebb fogunk még értekezni.) Megjegyzendő, hogy a C és OCaml rétegek a FreeBSD rendszerekben szokásos módon egymástól függetlenül futnak és a köztük levő kommunikációt egy FIFO sor valósítja meg, hasonlóan egy eszközmeghajtóhoz. A C réteg feladata tehát fogadni az `mbuf(9)` formátumú kereteket és egymás után felfűzni ezeket a feldolgozási sorban. Továbbá, ezeket az OCaml rétegben alkalmazott `Cstruct` típusú elemekké alakítja. Ezek azok a hatékony, tömörszerű típusok, amelyekről már a Xen implementáció kapcsán is szót ejtettünk.

Az OCaml réteg a `caml_get_next_mbuf()` külső (foreign-function, FF) híváson keresztül a `listen` függvényében folyamatosan felhossa az időközben beérkező kereteket; már amennyiben voltak az utolsó hívás óta, ezek parcialitását jelöli az `option` a típusban – hasonló a Haskell `Maybe` típusához. Ez azért kiemelendő, mert így nem fog várni keretre, ha nincs az adott időpillanatban, akkor egyszerűen a `yield` hívásával – kooperatív módon – továbbadja a vezérlést. Amennyiben viszont kaptunk keretet az alsóbb C rétegtől, úgy meghívjuk azt egy magasabbrendű függvényként a felsőbb szint – jelen esetünkben ez a `mirage-net` komponens – megfelelő feldolgozó függvényével.

Küldés esetén nagyjából ugyanez játszódik le, csak visszafele. Ilyenkor az OCaml réteg egy listát továbbít a C réteg felé, amelyben `mbuf(9)` értékeként elküldendő IO page-ek szerepelnek és ezeket képezzük végül le, majd küldetjük el a hálózati interfésznek. Itt, a hatékonyság kedvéért az egyszerre érkező page-eket egyetlen `mbuf(9)` láncba kapcsoljuk össze és küldjük tovább a hálózati interfésznek. Ez gyakran azért hasznos, mert ezek egyetlen keretet szoktak jelenteni, csupán kisebb darabokból raktuk össze.

Végezetül röviden összefoglalnánk, hogy a FreeBSD kernelben futó Mirage példányok miként jutnak hozzá a hálózatról érkező keretekhez. Ezt azért fejtjük ki, mert a FreeBSD kernelben jelenlevő bizonyos optimalizációk miatt például a gazdarendszer belső forgalma a hálózati interfészre nem is jut el (mint ahogy elvárható is lenne), illetve valahogy szeretnénk a hálózati interfészeket több példány közt is megosztani. Ezt a 6. ábra foglalja össze.

A hálózati interfész megosztásához minden betöltött Mirage példánynak indításkor részben véletlenszerűen előállítunk egy fizikai (MAC) címet, így a rendszeren belül egyértelműen azonosíthatóvá és elérhetővé válik. A NetGraph (`ng_ether(4)`) segítségével egymás után felfűzzük a



6. ábra. A FreeBSD kernelben futó Mirage példányok viszonya.

futó példányokat a FreeBSD TCP/IP stackjének `ether_input()` függvényére, és ezek sorosan körbeadják egymásnak a fogadott `mbuf(9)` értéket. A címzett ilyenkor mindig leveszi a keretet magának és korábbiakban bemutatott szerint feldolgozza. Látható még az ábrán egy visszacsatolás is az `ether_output()` hívásával. Ez azért szükséges, hogy a modulok egymásnak is tudjanak kereteket küldeni. Küldés során a modulok egymástól függetlenül is, közvetlenül a hálózati interfészre tudnak adatokat továbbítani. Ez az egész tulajdonképpen egy manuálisan megvalósított bridging, amely gyakori megoldás a virtuális gépek és a gazdarendszer hálózati kapcsolatának megosztására.

A futó példányoknak ezután egy egyszerű címfordítással (Network Address Translation, NAT), például a FreeBSD-ben levő PF (Packet Filter) alkalmazásával, tudjuk a kívülről érkező IP-csomagokat továbbküldeni. Ennek beállítására egy példát a 7. ábra mutat, ahol a 80-as porton futó Mirage alapú webszervert kapcsoljuk össze a külvilággal.

5. Összefoglalás

Ebben a cikkben azt mutattuk be, hogy miként lehet napjainkban a funkcionális programozást összekapcsolni a rendszerszintű fejleszté-

```
HOSTIF=em0
MIRGW=10.0.0.1
MIRIP=10.0.0.2
NETMASK=255.255.255.0
NETWORK=10.0.0.0/24

ifconfig ${HOSTIF} alias ${MIRGW} netmask ${NETMASK}
sysctl net.inet.ip.forwarding=1
kldload pf
pfctl -f- << EOF
nat on ${HOSTIF} from { ${NETWORK} } to any -> (${HOSTIF})
rdr on ${HOSTIF} inet proto tcp to port 80 -> ${MIRIP} port 80
EOF
pfctl -e
```

7. ábra. Egy Mirage példány hálózati elérésének beállítása FreeBSD rendszeren.

sekkel, miként lehet funkcionális nyelven az ipari igényeket is kielégítő operációs rendszert fejleszteni. Ennek szemléltetésére a Cambridge University Computer Laboratory egyik projektjét, a Mirage OS-t mutattuk be, amely a Xen virtuálizációs megoldásaira épülő, OCamlben megvalósított, exokerneles, elosztott operációs rendszer és felhőszolgáltatások létrehozására alkalmazható. Rengeteg, a modern funkcionális nyelvekből megismert eszközt alkalmaz: algebrai adattípusokat, monádokat, magasabbrendű függvényeket, erős statikus típusozást és típuskikövetkeztetést, szemétygyűjtőtetést.

Továbbá bemutattuk a Mirage OS FreeBSD kernelre alkalmazott változatát, amelyen keresztül röviden felvázoltuk az implementáció során felmerült nehézségeket és azok megoldásait. Habár itt leginkább a C nyelvi részletekkel foglalkoztunk, ez a forráskódból nagyjából 2000 sort tesz ki. Az alkalmazások összes többi része már OCamlben íródik; viszont pontosan emiatt nem kerülnek annyira az átültetéssel foglalkozó kérdéskörök fókuszába. Ezek megoldásával a Mirage összes többi része szinte ugyanúgy használható, mint Xen vagy a UNIX backendek esetében.

Szeretnénk köszönetet mondani a lehetőségért és a segítségért a Cambridge University Computer Laboratory munkatársainak, Robert N. M. Watsonnak és Anil Madhavapeddy-nek, valamint a laborban dolgozó FreeBSD és Mirage fejlesztőknek.

Hivatkozások

- [1] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, J. Crowcroft, *Unikernels: Library Operating Systems for the Cloud*, Proceeding of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems, 2013.
- [2] <http://openmirage.org/>
- [3] D. R. Engler, M. F. Kaashoek, J. O'Toole Jr., *Exokernel: An Operating System Architecture for Application-Level Resource Management*, Proceedings of the 15th ACM Symposium on Operating Systems Principles, 1995, pp. 251–266.
- [4] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, G. Hunt, *Rethinking the Library OS from the Top Down*, Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, 2011.
- [5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, *Xen and the Art of Virtualization*, Proceeding of the 19th ACM Symposium on Operating Systems Principles, 2003, pp. 164–177.
- [6] A. Madhavapeddy, R. Mortier, R. Sohan, T. Gazagnaire, S. Hand, T. Deegan, D. McAuley, J. Crowcroft, *Turning down the LAMP: Software Specialisation for the Cloud*, 2nd USENIX Workshop on Hot Topics in Cloud Computing HotCloud, 2010.
- [7] D. Doligez, A. Frisch, J. Garrigue, D. Rémy, J. Vouillon, *The OCaml system release 4.00 – Documentation and user's manual*, Institut National de Recherche en Informatique et en Automatique, 2012.
- [8] <http://wiki.xen.org/wiki/Mini-OS>
- [9] <http://ocsigen.org/lwt/>
- [10] K. McKusick, G. Neville-Neil, *The Design and Implementation of the FreeBSD Operating System*, Pearson Education, 2004.
- [11] T. Gazagnaire, *OPAM, a package manager for OCaml*, OCaml User and Developer Workshop, 2012.

[12] <https://github.com/pgj/mirage-platform>

C++ template metaprogramozást támogató eszközök

Porkoláb Zoltán

Bolyai Kollégium
Eötvös Loránd Tudományegyetem

gsd@elte.hu

A C++ template metaprogramok Turing-teljes nyelvet alkotnak, melyeket a szabványos C++ fordítók hajtanak végre fordítási időben. A template metaprogramok használata folyamatosan növekszik mind az akadémiai, mind az ipari méretű szoftverekben, mivel segítségükkel ki lehet terjeszteni a C++ nyelv képességeit: akár aktív könyvtárakról, kifejezés template-ekről, vagy szakma-specifikus nyelvek beágyazásáról van szó. A metaprogramok fejlesztése azonban keserves feladat, amit részben a fordítási idejű programok sajátosságai, részben a támogató eszközök majdnem teljes hiánya okoz. E cikkben ismertetjük a C++ template-ek és template metaprogramok sajátosságait és alapvető használatukat, majd ismertetjük a Templight rendszert. A Templight egy C++ template metaprogram debugger és profiler rendszer, amely alkalmas arra, hogy a fordítás során végrehajtott template akciókat rekonstruálja, és azokat kliens programok segítségével megjelenítse, hasonló módon a futási idejű programokhoz alkalmazott eszközökhöz. A Templight nyílt forráskódú szoftver, forrásszintű kiegészítésként került implementálásra a népszerű LLVM/clang fordítóprogramhoz.

1. Bevezetés

A C++ programozási nyelv az egyik legnépszerűbb multiparadigmás programozási nyelv, amit széleskörűen használnak nagy számítási sebességet kívánó alkalmazásoktól kezdve beágyazott rendszerekig [4,16].

A nyelv sikerének egyik titka, hogy – az objektumelvű és a funkcionális programozás mellett – erős támogatást nyújt programok generikus elven történő elkészítéséhez is. A *generikus programozás* az a programozási paradigma, amely az adatszerkezetek és az eljárások absztrakciós szintjének emelésével lehetővé teszi magas fokon újrafelhasználható komponensek (adatszerkezetek és algoritmusok) elkészítését [9, 14]. Ezeket a komponenseket széleskörűen variálva tudjuk egymáshoz építeni és megfelelő paraméterekkel ellátva konkrét feladatokra alkalmazni. Miután a komponensek összeépítése fordítási időben történik, az újrafelhasználhatóságért nem kell futási idejű hatékonyságromlással fizetnünk [19]. A generikus paradigma segítségével, komponensekből, parametrizálással összeállított programok összevethető hatékonyságúak az adott feladatra alkalmilag, kézzel írt programokkal.

A generikus programozást a C++ nyelvben a template-ek használatával érjük el. A *template* (sablon) egy olyan programkonstrukció, melyben egy, vagy több paramétert fordítási időben határozzunk meg. Ezek a paraméterek többnyire típusparaméterek, de esetenként előfordul skalár (`bool`, `char`, `int`, stb.) konstans paraméterek használata is. Az általánosítást pont azáltal érjük el, hogy egy programkonstrukcióban egyes konkrét típus vagy értékek helyett egy vagy több általános template paramétert vezetünk be, ezáltal az adott programrészletet alkalmassá tesszük arra, hogy más paraméterek fordítási idejű alkalmazása esetén más programkörnyezetben is újra felhasználjuk. A kód újrafelhasználásának ezt a fordítási idejű paraméterekkel támogatott módját *parametrikus polimorfizmusnak* (parametric polymorphism) is szokás nevezni [5].

Template-ek alkalmazása nélkül ma már elképzelhetetlen egy C++ program. A szabványos C++ könyvtár legtöbb eleme maga is template, beszéljünk akár az `std::string` típusról (mely az `std::basic_string<>` template példányosítása `char` típusparaméterrel valamint megfelelő `char_traits`-el és `allocator`-ral), vagy akár a szabványos adatfolyam-típusokról. A szabványos C++ könyvtár része az STL (Standard Template Library) is, mely absztrakt adatszerkezetekkel és algoritmusokkal az újrafelhasználható generikus könyvtárak mintapéldánya [9].

Még 1994-ben, amikor az első ISO C++ szabványt kezdték kidolgozni, az egyik szabványbizottsági tag, Erwin Unruh bemutatót a biztonság számára egy C++ programot. Maga a program nem fordult le, viszont

a fordítóprogram által kiírt hibaüzenetek soraiban az első prím számok jelentek meg. Unruh ezzel a programmal azt a sejtését kívánta igazolni, hogy alkalmasan elhelyezett template-ek segítségével a C++ fordító fordítási időben képes általunk megadott algoritmusokat végrehajtani.

Ezt a módszert hívják *C++ template metaprogramozásnak*.

Általában metaprogramok alatt olyan programokat értünk, melyek programokat manipulálnak: azokból információt nyernek, azokat létrehozzák, vagy módosítják. A metaprogramok speciális esete, amikor a metaprogram és célpontja nem két különböző program, mint pl. egy fordítóprogram, parszer-generátor, aspektus-szövőgép, stb. esetben, hanem maga a program képes önmagáról információkat gyűjteni (introspection, reflection) vagy önmagát módosítani (intercession). Ez utóbbi eset – ahol a program saját magát módosítja – sem ritka, hiszen pl. a C előfordító a fordítás első lépéseként a makróhelyettesítéssel pontosan ilyen tevékenységet végez.

Azokat a programozási nyelveket, melyekben az előbb említett C előfordító példához hasonlóan a metaprogramozási lépések egymástól időben jól elkülönülnek *többszintű nyelveknek* nevezzük. A többszintű nyelvek a 2000-es év, a generatív programozás előtérbe kerülése óta erősen kutatott terület. Számos cikk tárgyalja ezen nyelvek szintaktikai és szemantikai sajátosságait, és különböző alkalmazási területeit, pl. a szakma-specifikus nyelvek beágyazásának képességében. A többszintű nyelvek egyik legújabb példája a scala programozási nyelv [10], melyben a fordítási tevékenység számos, a programozó által befolyásolható metaprogramozási lépésből áll.

A C++ template metaprogramok szabványos C++ programok, melyek lefordítása közben a template-ek specializációjára és példányosítására vonatkozó szabályoknak megfelelően vezérelhetjük a fordítóprogram működését. A C++ template metaprogramokról bebizonyították, hogy Turing-teljesek [5], tehát segítségükkel – a fordítóprogram számára elérhető erőforráskorlátokon belül – tetszőleges algoritmusokat végrehajthatunk.

Az elmúlt közel húsz évben számos kutatás célja volt a template metaprogramozás elméleti és gyakorlati lehetőségeinek meghatározása, és a metaprogramozás folyamatának támogatása. Kidolgozták a legfontosabb fordítási idejű algoritmusokat és adatszerkezeteket. Létrehoztak programkönyvtárakat, amelyek a leggyakrabban előforduló feladatok megoldására kínálnak támogatást [1, 3, 25]. Mégis, a C++ template

metaprogramozás ma is a szakmailag legnehezebb C++ feladatok közé tartozik. A template metaprogramok fejlesztése nehezen tervezhető, gyakran ad-hoc jellegű megoldásokra kényszerülnek a fejlesztők. Az esetlegesen elkövetett programozói hibák, vagy az adott fordítóprogram szabványtól való eltérése miatti problémák nehezen értelmezhető hibaüzenetekhez vezetnek. A metaprogramozás általános jellege (fordítási időben történő végrehajtás, az input/output erősen korlátozott volta) egyébként is gondokat okoz. Ehhez járul hozzá a fejlesztést támogató eszközök (debuggerek, profilerek, teszt eszközök, stb.) szinte teljes hiánya. Mindezek miatt a metaprogramozás nehezen kiszámítható fejlesztési folyamat. A döntéshozók számára ez a bizonytalansági faktor a nagy ipari projektekben még akkor is a template metaprogramozás alkalmazása ellen szólhat, ha egyébként más szakmai érvek, mint a hatékonyság, a fordítási idejű adaptáció, vagy (például DSL integráció esetében) a külső eszközöktől történő függés minimalizálása a template metaprogramozás mellett szól.

Az alábbiakban olyan eredményeket szeretnénk ismertetni, melyek eszköztámogatást nyújtanak a C++ template metaprogramok fejlesztéséhez. A hibásan futó template metaprogramok elemzéséhez *Templight* néven a hagyományos debuggerekhez hasonló elemző rendszert készítettünk, melynek segítségével az LLVM/clang C++ fordító által végrehajtott template példányosítások nyomon követhetőek és elemezhetőek. A Templight központja egy C++ forráskódú patch, amelyet a clang forrására kell alkalmazni. A módosítások után a fordítóprogram az egyes template-ekkel kapcsolatos műveletekről (például példányosítás kezdete, befejezése, memoization, stb.) bejegyzést készít, melyeket később a template metaprogram lefutásának rekonstruálására lehet felhasználni. A bejegyzéseket külső programok elemzik, vagy akár grafikus módon meg is jeleníthetik, hasonlóan a futási idejű debuggerek viselkedéséhez. Az egyes események időpecsétet is kapnak, ami profiler funkciók megvalósítását is lehetővé teszik.

A Templight rendszer és néhány grafikus felhasználói interfésszel rendelkező kliens program nyílt forráskódú szoftverként implementálásra került [27]. Az elmúlt közel egy évben a Templightről szóló előadások nagy látogatottsága és a szoftverek letöltési adatai, valamint egyéb visszajelzések igazolták, hogy a Templight rendszer hasznos eszköz template metaprogramozók és egyéb C++ fejlesztők számára. A szerző bízik benne, hogy a Templight és hasonló rendszerek egyszerűbbé

és jobban tervezhetőbbé teszik a C++ template-ek használatát és különösen a template metaprogramozást, evvel is segítve további elterjedésüket.

A dolgozat szerkezete a következő: a 2. fejezetben rövid informális bevezetést adunk a C++ template-ekről, használatukról. Nem törekszünk általános, tankönyv jellegű bemutatásra, csak a template metaprogramozás szempontjából fontos elemeket ismertetjük. A 3. fejezetben áttekintjük a template metaprogramozás főbb elemeit és alkalmazási területeit. A metaprogramozás különösen érzékeny az elkövetett hibákra: a fordítóprogramok gyakran közvetett, nehezen értelmezhető hibaüzeneteket adnak. Az 4. fejezetben a metaprogramok debuggálására és hatékonyságvizsgálatára (profilíng) alkalmas Templight rendszert mutatjuk be. A metaprogramok vizsgálatára irányuló kapcsolódó munkákról a 5. fejezetben adunk áttekintést. Végül a 6. fejezetben összefoglaljuk az elért eredményeket.

A szerző szeretne köszönetet mondani Sipos Ádámnak, és Mihalicza Józsefnek a Templight korábbi, kódinstrumentáción alapuló verziójában való részvételükért, Sinkovics Ábelnek ötleteiért és a Templight tesztelésért, legfőképpen pedig Borók-Nagy Zoltánnak a jelenlegi Templight rendszer implementálásáért.

2. Informális bevezetés a C++ template-ekhez

Az alábbiakban néhány példa segítségével szeretnénk bemutatni a C++ template mechanizmus legfontosabb elemeit: a sablonok definiálását, a példányosítás lépéseit és a specializációt. A példakódokban a tömörség érdekében némiképpen engedékenyek leszünk a C++ szintaxissal szemben; ahol az nem zavaró, ott elhagyjuk az egyébként szükséges `#include` direktívákat és névtér jelöléseket (mint pl `std::`).

Kezdjünk egy nagyon egyszerű feladattal: két azonos típusú szám közül a nagyobbbat szeretnénk kiválasztani egy `max` függvénnyel. A klasszikus procedurális programozás szerint az egyes `max` függvényeket eltérő névvel kell ellátnunk (pl. `imax`, `fmax`). Korszerűbb objektumelvű nyelvekben, ahol a függvények túlterhelése lehetséges, ugyancsak minden egyes típusra külön függvényt kell írunk. Az viszont a programozónak könnyebbé, hogy ezeket a függvényeket már nevezhetjük azonosan

(pl. `max`), csak a paraméterlistájukban kell, hogy különbözzenek. Az azonos nevű, de különböző paraméterlistájú függvények közül a fordító kiválasztja a legalkalmasabbat (ha van ilyen).

Az eredmény két okból sem kielégítő. Egyrészt az egyes `max` függvények csak a paraméterlistájukban és visszatérő érték típusában különböznek, ami nehezen kezelhető kódismétléshez vezet. Ha valamely `max`-ot módosítanunk kell, az összeset egyformán kell módosítani. A másik probléma, hogy függvényeket csak már létező típusok segítségével tudunk definiálni. Ha később egy új típust vezetnek be, ahhoz nem fog `max` függvény létezni, azt a típus bevezetése után lehet csak majd definiálni. Mindez avval jár, hogy a `max` függvényeink készletét nem tudjuk egy zárt könyvtárban megvalósítani, még akkor sem, ha előre tudjuk, a maximális értéket kiválasztó algoritmust (természetesen az új típus kisebb művelete függvényében).

Ügyes C programozók számára komoly kísértés előfordító által definiált makró létrehozása.

```
#define MAX(a,b) a > b ? a : b
```

Az előfordító típustalan, ezáltal a fenti makró az összes eddigi és majdan létrehozott, `<` művelettel rendelkező típusra működni fog. Azonban a típustalanság maga is probléma, ha például ideiglenes tárterületet kéne létrehozni valamely művelet implementálására, már gondban lennénk a típus megadásával. C/C++ makrók használatát számos egyéb káros mellékhatásuk miatt sem javasolhatjuk.

Olyan intelligens, a fordítási folyamatba beépített, a programozási nyelv típusrendszerével együttműködni képes – az alkalmazott típusokat felismerő és szükség esetén új típusokat is létrehozni tudó – módszerre van szükség, ami a makrókhoz hasonlóan az alkalmazás helyén, a paraméterektől függően generál kódot. A C++-ban ezt a mechanizmust *template*-eknek nevezik.

A *template*-ek lehetnek *osztálytemplate*-ek, vagy *függvénytemplate*-ek, attól függően, hogy valamely absztrakt adatszerkezet (osztály) vagy egy algoritmus (függvény) általánosítását szeretnénk definiálni. Az osztály- és függvénytemplate-ek nem osztályok vagy függvények abban az értelemben, hogy közvetlenül nem generálódik belőlük kód (ezért is nem *template* osztálynak vagy *template* függvénynek nevezzük őket); sokkal inkább tekinthetők sablonoknak, egyfajta gyártási eljárásoknak, melyek segítségével a fordítóprogram képes automatikusan legyártani

a konkrét típusokkal rendelkező osztályokat és függvényeket. Ezt a gyártási folyamatot *példányosításnak* (instantiation) nevezzük és kiváltódhat automatikusan, amikor egy template-re valamely addig nem használt típusparaméterrel hivatkozunk, vagy manuálisan is, a programozó explicit utasítására. A példányosítást gyakran úgy képzelhetjük el, hogy a template formális paraméterei fordítási időben valamely aktuális típussal vagy értékkel helyettesítődnek és ezekkel történik meg a fordítás és kódgenerálás. A valóságban legtöbbször ennél sokkal kifinomultabb mechanizmusok történnek a háttérben [23]. További fontos körülmény, hogy ha valamely programban egy template-et egyáltalán nem példányosítunk, akkor abból kód sem generálódik – ez nem valamilyen opcionális, hatékonyságnövelő optimalizáció, hanem a C++ szabványban szigorúan lefektetett szabály.

Az alábbi példában a `max` függvénytemplate-ként történő megvalósítását mutatjuk meg pár példányosítási esettel:

```
template <class T>
T max( T a, T b)
{
    if ( a > b )
        return a;
    else
        return b;
}

int i = 3, j = 4, k;
double x = 3.14, y = 4.14, z;

k = max(i, j);    // -> max(int, int)
z = max(x, y);    // -> max(double, double)
z = max(i, x);    // -> syntax error
```

Figyeljük meg, hogy az utolsó eset fordítási idejű hibát okoz. Ennek az az oka, hogy a `T` típusparaméter meghatározásakor, amit *paraméter kikövetkeztetésnek* (parameter deduction) nevezünk, a fordító ellentmondásra jut: az első paraméter szerint `T` `int`, a második paraméter szerint `T` `double` típusú. Vegyük észre egyrészt, hogy az értékadás baloldalán álló `z` változó típusa (`double`) nem befolyásolja a jobboldali template

paraméter kikövetkeztetését; a legtöbb programozási nyelv igyekszik elkerülni a környezetfüggő szabályok alkalmazását. Másrészt a template paraméterek között nem történik konverzió.

Megpróbálkozhatunk egy második típusparaméter bevezetésével is, viszont ekkor bajba kerülünk a visszatérő érték típusának megadásakor. Valójában egy harmadik típusparamétert is be kell vezetnünk:

```
template <class R, class T, class S>
R max( T a, S b)
{
    if ( a > b )
        return a;
    else
        return b;
}
```

Látszik, hogy a *T* és *S* paraméterek kikövetkeztethetők egy konkrét hívási környezetből. Azt azonban már korábban említettük, hogy a visszatérő érték típusán nincsen paraméter kikövetkeztetés, *R* típusát nekünk kell explicit megadnunk:

```
double z = max<double>(i, x);
```

Ezt *explicit specializációnak* nevezzük. Ugyanakkor kényelmetlen lenne az explicit specializációt alkalmazni azokban az esetekben is, amikor a visszatérő érték típusa egyértelmű, mert pl. a *max* két paramétere azonos típusú. Ennek a problémának megoldására a két *max* template-et, az egy template paraméteres és a három template paraméteres verzióját egyszerre is használhatjuk. A túlterhelt template-ekből a fordító választja ki a megfelelő példányosítandó sablont.

```
template <typename T> T max(T,T);
template <typename R, typename T, typename S> R max(T,S);
```

```
int i = 4, j = 5;
double x = 3.14;
```

```
int k = max(i, j);           // -> T max(T,T)
double z = max<double>(i, x); // -> R max(T,S)
```

Az eddigiekben a sablonok mindig azonos algoritmust hajtottak végre, az egyes verziók csak a paraméterek számban, illetve a visszatérő érték

megadása módjában különböztek. Előfordulhat azonban, hogy valamely típusra, vagy típuscsoportra a template egy eltérő implementációját szeretnénk megadni. Ez a helyzet például, ha két C stílusú string közül szeretnénk a lexikografikusan nagyobbát kiválasztani; a korábban definiált `max`-ok a paraméterként átadott mutatókat hasonlították volna össze. A lexikografikus maximum kiválasztáshoz egy *felhasználói specializációt* (user specialization) kell létrehozunk:

```
template <>
const char *max( const char *s1, const char *s2)
{
    return strcmp( s1, s2) < 0;
}

char *s1 = "Hello", *s2 = "world";
cout << max(s1, s2);
```

A fenti példák függvénytemplate-eken keresztül mutatták be a template definíció, a paraméter kikövetkeztetés, a példányosítás és a specializációk használatát. Teljesen hasonlóan működnek az osztálytemplate-ek is. Osztálytemplate-ek esetében megjegyzendő, hogy az egyes specializációk teljesen eltérhetnek az általános változat megvalósításától. Még az sem követelmény, – bár erősen ajánlott – hogy az osztálytemplate általános és specializált verzióinak azonos legyen a publikus interfésze.

A template-eket nem minden esetben példányosíthatjuk tetszőleges típusokkal vagy értékekkel. A maximumos esetben követelmény például, hogy a paraméterek összehasonlíthatók legyenek az `operator<` művelettel. Java nyelven változatos módon tudjuk a generikusok használatát korlátozni [17]. Sajnos ilyen követelményeket nem tudjuk a C++14 szabvány előtti szintaxisban leírni [6]. Úgy is szoktuk mondani, hogy a C++ nem rendelkezik *sablon-szerződés modellel* (constraint generic). Amennyiben helytelen paramétert alkalmazunk, a példányosítás elkezdődik és valahol – a programozó számára néha nehezen kiszámítható módon – hibát okoz. A C++0x tervezésekor nagy hangsúlyt fektettek a *concept*-ek kidolgozására [13], amely a sablon-szerződés modellt valószínűleg tartalmazta volna meg. Sajnos, ez az utolsó pillanatban kikerült a C++11 szabványból. A C++14 szabványban ennek egy egyszerűsített, de talán könnyebben használható megoldását vezetik be.

3. C++ template metaprogramok

1994-ben a C++ szabványosítási bizottság még javában dolgozott a template-ekkel kapcsolatos szabályok pontos megfogalmazásán, amikor Erwin Unruh, a bizottság egyik tagja bemutatott számukra egy C++ programot [18]. A program a lefordítására tett kísérlet során fordítási hibaüzeneteket generált, melyek 2-től növekvően tartalmazták a prím számok növekvő sorozatát. A program annak a demonstrációja volt, hogy a megfelelően definiált és meghivatkozott template-ek segítségével a szabványos C++ fordító kívülről vezérelhető és előre definiált algoritmusok végrehajtására alkalmas.

A template metaprogramokat és végrehajtásukat a klasszikus, faktoriális függvényt fordítási időben kiszámoló példán keresztül mutatjuk meg. A megoldás egy Factorial nevű függvénytemplate-et és annak specializációját tartalmazza:

```
template <int N>
struct Factorial
{
    enum { value = N * Factorial<N-1>::value };
}

template<>
struct Factorial<1>
{
    enum { value = 1 };
};

int main()
{
    int r = Factorial<5>::value;
    return 0;
}
```

A template metaprogram végrehajtása fordítási időben történik. A main függvényben hivatkozunk a `Factorial<5>::value` értékére. Mivel `Factorial` egy template, a fordítóprogram megkísérli példányosítani a `Factorial` sablont az `int 5` paraméterértékkel. E példányosítás során a sablon belsejében hivatkozunk a `Factorial<N-1>::value` értékére. Ekkor a fordítóprogram felfüggeszti `Factorial<5>` példányosítását és

megkezdzi `Factorial<4>` létrehozását. Ez a template hivatkozás tehát egyfajta rekurzióként működik és kikényszeríti az egyes `Factorial` osztályok példányosítását egyre csökkenő értékű paraméterekkel. Amikor a példányosítási lánc eljut `Factorial<1>`-hez, az általános sablon helyett a specializált template szolgál a példányosításhoz. Mivel ez nem tartalmaz további template hivatkozásokat a rekurzív leszállás megáll és a konkrét `Factorial<1>::value` értéke segítségével (ami 1) befejeződik `Factorial<2>` létrehozása. Ez a visszatérés a rekurzióból, így tovább, egészen `Factorial<5>` létrehozásának befejeződéséig. Ekkor mind az öt osztály a rendelkezésünkre áll, az összes `value` értékkel. A `main` függvényben az `r` változó kezdőértéket kap `Factorial<5>::value` azaz 120 értékkel. Végrehajtottunk egy egyszerű template metaprogramot fordítási időben.

Két fontos szabályt használtunk fel ennél az algoritmusnál: (1) a konstans kifejezések – azokat, melyeket fordítási időben kiszámíthatunk – kötelezően ki lesznek értékelve fordítási időben. Ez a szabály indította el a `Factorial<5>` példányosítását. Másrészt (2) azokat a template-ek, melyekre nem történik hivatkozás, nem szabad, hogy példányosuljanak. Ez a szabály akadályozta meg, hogy a `Factorial<0>`, `Factorial<-1>`, stb. példányosuljon, így kerültük el a végtelen rekurziót. Hangsúlyozni kell, hogy ez nem valamely optimalizáció eredménye, hanem explicit szabály, emlyet a C++ szabvány tartalmaz.

Rekurzív template-ek mellett szimulálhatjuk az egyes vezérlési szerkezeteket is. Az elágazást például az alábbi módon implementálhatjuk:

```
template <bool condition, class Then, class Else>
struct IF
{
    typedef Then RET;
};

template <class Then, class Else>
struct IF<false, Then, Else>
{
    typedef Else RET;
};

int main()
{
```

```
IF< sizeof(int)<sizeof(long), long, int>::RET i;  
cout << sizeof(i) << endl;  
return 0;  
}
```

A fenti példában *részleges specializációt* (partial specialization) alkalmaztunk. Az IF template első paramétere egy bool konstans (igaz vagy hamis), míg a második és harmadik paraméterek tetszőlegesek. Amennyiben az első paraméter igaz érték, az általános változat példányosul, és a template által definiált RET típus a második paraméter, azaz Then szinonimája lesz. Amennyiben viszont az első paraméter hamis, akkor a specializáció szerint a RET a harmadik, Else paraméterrel egyezik meg. A konstrukció természetesen szimmetrikus, azaz definiálhattuk volna az Else-t RET-nek az általános esetben és specializálhattuk volna az első paraméter igaz értékére is.

A fenti IF konstrukció lényegében egy mintaillesztés, és hasonlóképpen használható a klasszikus programozási nyelvek futási idejű elágazásához. A korábban látott rekurziós mechanizmussal és a mintaillesztéssel egy teljes programozási nyelv áll rendelkezésünkre. Valóban, a C++ template metaprogramokról belátták, hogy *Turing-teljes* nyelvet alkotnak, melyet fordítási időben hajthatunk végre [5]. Természetesen a gyakorlatban – ahogy ez a futási idejű programoknál is fennáll – a fordító számára rendelkezésre álló erőforrások (elsősorban a memória és futási idő) erősen korlátozhatja a végrehajtható algoritmusokat.

Mikor alkalmazunk template metaprogramokat?

A generikus programok újrafelhasználása úgy történik, hogy az általános, parametrikus polimorfizmus segítségével megírt kódotkat valamely típusparaméterekkel (konkrét típusokkal vagy értékekkel) szeretnénk alkalmazni. C++-ban ez a generikus kód, template példányosítása segítségével történik. Azonban sokszor előfordul, hogy a template megtervezésekor olyan döntéseket kéne hoznunk, melyeket a konkrét típusparaméterek ismerete nélkül nem tudunk megtenni. Ha például a max függvényünk egy double és egy int értéket hasonlít össze, akkor – függetlenül a konkrét paraméterek értékeitől – biztonságosabb a double típussal visszatérni. A max template definiálásakor azonban a paraméterek típusait csak a T és S template paraméterekkel jelöljük, nem tudhatjuk, hogy egy konkrét példányosítási környezetben mi lesz T és S konkrét típusa. Hasonlóképpen, lehetséges hogy valamely T típusú objektumokat szeretnénk tárolni programunkban. Ha T (futási idejű)

polimorfikus, például C++-ban tartalmaz *virtuális függvényeket*, akkor nagy valószínűséggel nem magukat az objektumokat, hanem a rájuk mutató pointereket szeretnénk tárolni, egyébként pedig magukat az objektumokat. A tároló megtervezésekor azonban még nem tudhatjuk, milyen aktuális értékei lesznek a T típusparaméternek. Ez majd csak a tároló konkrét típusokkal történő felhasználásakor derül ki

Ezeket és a hasonló a döntéseket tehát nem tudjuk meghozni a program tervezési idejében, hiszen hiányoznak a template-ek példányosításakor ismert adatok. Fel tudunk azonban előre készülni: template metaprogramokat hozhatunk létre, melyek majd a példányosításkor aktivizálódnak, algoritmusokat futtatnak és döntéseket hoznak.

```
template <class T, class S>
IF< sizeof(T)<sizeof(S), S, T>::RET max(T x, S y)
{
    if ( x > y )
        return x;
    else
        return y;
}
```

A `max` template ezen verziója képes fordítási időben, a template példányosításakor, a konkrét `T` és `S` paraméterek ismeretében döntést hozni, és kiválasztani a „szélesebb”, nagyobb méretet foglaló típust, mint biztonságosabb visszatérő típus. A template metaprogramot a program megírásakor definiáltuk, és majd akkor fog végrehajtódni, amikor a `max` template példányosul. Akkor a metaprogram a `sizeof` operátor segítségével összehasonlítja a `T` és `S` típusparamétereket helyettesítő aktuális típusok méretét, és az összehasonlítás eredményétől függően választja ki, hogy az `IF` sablon `RET` tagtípusa `T` vagy `S` szinonimája legyen (`typedef`).

Meg kell jegyezni, hogy semmi garancia nincsen általánosságban, hogy a „szélesebb” típus a jobb. Ugyanakkor Turing-teljes nyelvként, a template metaprogramok lehetővé tesznek jóval kifinomultabb algoritmusokat is.

A C++11 szabvány szerint a `max`-os példát megoldhattuk volna template metaprogramok segítségével is az `std::common_type` alkalmazásával, de ez nem csökkenti a template metaprogramok általános alkalmazhatóságának lehetőségeit. Másrésről most sem a

futási időben nagyobbik érték típusát választottuk ki, hanem csak a szélesebb típust, függetlenül attól, hogy ez a nagyobbik érték típusa volt-e.

Azokat a generikus könyvtárakat, melyek példányosulásukkor döntéseket hoznak *aktív könyvtáraknak* (active libraries) nevezzük, és a template metaprogramok egyik fő alkalmazási területe [7, 20]. A másik terület a *kifejezés-sablonok* (expression template) területe, ahol a hagyományos, objektumelvű szintaxissal felírt programok használatának futási idejű hatékonyságát lehet növelni a fordítási időben történő optimalizációkkal [22]. A kifejezés-sablonok rendszeresen előfordulnak vektor- és mátrixkönyvtárakban, és máshol is, ahol a hatékonyság elsődleges. Alkalmazznak template metaprogramokat a C++ típusrendszer kiterjesztésére [24] és alkalmazás specifikus nyelvek (DSL) beágyazására is [12].

Az alábbi táblázatban összefoglaljuk a template metaprogramok tulajdonságait a `max` példa segítségével:

Template definíciók	Fordítási idő	Futási idő
Algoritmusok tervezése Megtervezzük a template kódot	Template példányosítás Paraméter kikövetkeztetés Metaprogram végrehajtás	Algoritmusok végrehajtása A program kiértékeli a kifejezéseket
A <code>max(T,S)</code> visszatérő értéke definiálása metaprogrammal	A Paraméter kikövetkeztetés meghatározza T és S típusát definiálódik <code>IF<T,S>::RET</code>	A nagyobb paraméter értéke kiválasztódik típusától függetlenül

1. táblázat. Programozás template metaprogramokkal

4. A Templight rendszer

Ahogy a programfejlesztés számos más pontján, úgy a metaprogramok készítésekor is követhetünk el hibákat. Ilyenkor a hiba felfedezése, helyének és okának meghatározása előzi meg a javítást. A C++ template metaprogramokban elkövetett hibák azonban sokkal nehezebben felfedezhetőek, behatárolhatóak és kijavíthatóak, mint futási idejű párjaik [15]. Ennek több oka is van:

- A C++ szintaxisát nem a metaprogramok írásához tervezték. A template szintaxis gyakran vezet rosszul olvasható, nehezen javítható és karbantartható metaprogramokhoz.

- A C++ nyelvet és a fordítóprogramokat a klasszikus programozási hibák és nem a metaprogramok hibáinak felderítésére és jelentésére dolgozták ki. A metaprogramokban elkövetett hibákat csak nagyon áttételes módon, gyakran nehezen értelmezhető módon közlik.
- A fordítóprogramokat nem a metaprogramok végrehajtására optimalizálták. Gyakran olyan belső adatszerkezeteket és algoritmusokat használnak, melyek kielégítően működnek klasszikus programok lefordításánál, de nem hatékonyak metaprogramok végrehajtásakor.
- A fordítóprogramok belső szerkezeti részletei ismeretlenek a legtöbb programozó számára.
- A programozók többségének nincsen tapasztalata a C++ template metaprogramok írásában.

Ehhez hozzájárul még a metaprogramozást támogató eszközök szinte teljes hiánya is. A futási idejű programok fejlesztését komoly eszköz-készlet támogatja: statikus programellenőrző eszközök, debuggerek, profilerek. Eközben a C++ template metaprogramok esetében hasonló eszközök csak csekély számban állnak rendelkezésre.

Ennek az eszközhianyának a pótlására dolgoztuk ki a *Templight* rendszert, amely egy fordítóprogramba beépülő debugger és profiler összetevőből és a kinyert adatokat vizualizáló kliens programokból álló programcsomag.

A *clang* fordító a népszerű nyílt forráskódú LLVM projekt C/C++ fordítóprogramja [26]. Más fordítóprogramoktól eltérően a *clang* moduláris szerkezetű, újrafelhasználható osztályokból épül fel objektumelvű módon, így ideálisan módosítható. A *Templight* technikailag egy patch, azaz forráskódú módosítás a *clang* forrásállományon. A módosítások részben a *clang* számára átadható parancssori paramétereket érintik, melyek bekapcsolják a *Templight*-specifikus működést, részben pedig a fordítóprogram egyes template-ekkel kapcsolatos egyes tevékenységét detektálják.

A *Templight* patch-elt *clang* fordítót pontosan úgy kell használni, mint a natív változatát. Ugyanakkor fordításkor további opcionális parancssori argumentumokat tudunk átadni.

```
$ clang++ -templight fib.cpp
```

```
$ ls
fib.cpp.trace.xml
$ wc fib.cpp.trace.xml
123 275 3838 fib.cpp.trace.xml
```

A `-templight` kapcsolóval bekapcsolhatjuk a rendszer fő funkcionalitását, a template akciók nyomkövetését. Innentől kezdve a clang fordító minden egyes template-ekkel kapcsolatos eseményéből (példányosítás kezdete, vége, memoization, stb.) egy-egy bejegyzés készül egy belső memóriatárolóba, kiegészítve egy időbélyeggel. A tároló a fordító indításakor foglalódik le, alapértelmezésben kellően nagy (500000 MB), de további parancssori paraméterekkel ez a méret megnövelhető. Mivel az egyes bejegyzések a fordító memóriáján belül kerülnek feljegyzésre és fordítás alatt se memóriafoglalás, se output művelet nem történik, ez a tevékenység gyakorlatilag nem jelent többletidőt a fordítás alatt. Vannak olyan template metaprogram hibák amelyek a fordítóprogram összeomlásához vezethetnek. Ilyen esetekben használhatjuk a `-safe` opciót, amely minden egyes template akciót azonnal kiír az outputra. Ilyenkor az összeomlás előtti utolsó akcióig rendelkezésre állnak a hibakeresési információk. (Természetesen ebben az esetben a profiling adatok erősen torzíthatnak).

A fordítás befejezése után (akár sikeres, akár hibás leállás következett be) az eseményeket kiírjuk egy output állományba. Jelenleg a YAML (default), szöveges és XML formátumok használatosak. Egy kiírt XML állomány eleje például ezt tartalmazhatja:

```
$ head fib.cpp.trace.xml
<?xml version="1.0" standalone="yes"?>
<Trace>
<TemplateBegin>
<Kind>TemplateInstantiation</Kind>
<Context context = "Fib<5>";"/>
<PointOfInstantiation>fib.cpp|22|14</PointOfInstantiation>
<TimeStamp time = "421998401.188854"/>
<MemoryUsage bytes = "0"/>
</TemplateBegin>
<TemplateBegin>
```

A Templight rendszer képes a használt memóriát is nyomon követni, de ez, mint a fenti példában látható, alapértelmezésként ki van kapcsolva.

Ennek a memórialekérő művelet magas költsége az oka, amely erősen torzítaná az egyéb profiling adatokat.

Az output állományt a Templight kliens oldali megjelenítő programjai, vagy egyéb külső programok használhatják fel. Az általunk megírt megjelenítők egyike, a *Templar* eszköz grafikus debuggerként működik: beolvassa a Templight által kibocsájtott nyomkövetési fájlt és azon interaktív tevékenységeket végez:

- Lépésről lépésre lejátszhatjuk a C++ template metaprogram végrehajtását, miközben egyaránt láthatjuk a forráskódot és a template-ek példányosítási fáját.
- Visszafelé is haladhatunk a metaprogram végrehajtáson.
- Megszakítási pontokat rakhatunk le egyes template-eknél, hogy ezekig a pontokig átléphessük az összes közbülső akciót.
- Szűrőket helyezhetünk el, hogy az általunk érdektelen template akciókat (például a standard könyvtár vagy a `boost::mpl` példányosításait) figyelmen kívül hagyhassuk.

A másik kliens programunkkal a példányosítási időket vizsgálhatjuk meg, akár névtér/template/osztálynév csoportosításban, akár a példányosítási fa alapján.

A Templight rendszer és a kliens programok nyílt forráskódú szoftverek, melyek letölthetőek a <http://plc.inf.elte.hu/templight> címről [27]. A clang fejlesztők erősen érdeklődnek a Templight funkcionalitás clang-ba történő állandó beépítése iránt.

5. Kapcsolódó munkák

A C++ template metaprogramok vizsgálatát először Veldhuizen végezte el cikkeiben [20–22]. Később Vandevorde és Josuttis vezették be a nyomkövető (tracer) template fogalmát [19]. Ez egy olyan osztály, melynek műveletei üzeneteket bocsájtanak ki. Ha egy ilyen osztályt adnak át egy template paraméterül, a kibocsájtott üzenetek megmutatják, hogy az egyes műveleteket milyen sorrendben és gyakorisággal hívták. Ennek az osztálynak egy változata azt ellenőrizte, hogy egy template nem használta-e fel paraméterének olyan tulajdonságait, melyekre nem számítottunk.

A sok template definíciót tartalmazó C++ programok fordítási idejének mérséklésére Veldhuizen dolgozott ki alternatív programozási modelleket, melyek mindegyike más szempontok szerint (fordítási idő, kód méret, vagy futási sebesség) volt optimális [23]. Javaslatai alátámasztásához a programok teljes fordítási idejét mérte. Hasonlóan járt el Gurtovoy és Abrahams is, amikor az egyes template metaprogram konstrukciók hatásait elemezték könyvükben [1]. A vizsgált konstrukciók programjaik hangsúlyos részét tették ki, majd a programok teljes fordítási idejéből következtettek ezek hatására. Természetesen a teljes programot mérve egyes részletek hatásai homályban maradtak. Abrahams és Coelho egy másik munkájában fordítóprogramokat és fordítási stratégiákat hasonlított össze és már figyelmeztet a template metaprogramok hatékonyságának olyan speciális összetevőire, mint az újrapéldányosítás (memoization) vagy az egyes szimbólumok mérete is [2].

Gurtovoy és Abrahams már említett könyvükben egy teljes fejezetet szentelnek a diagnosztikának, ahol módszereket mutatnak arra, hogyan lehet szöveges outputot kibocsájtani template metaprogramokból warning-ok segítségével. Implementálták is a korábban említett tracer osztály template metaprogram változatát (`mpl::print`) [1].

A szerző és társai egy korábbi munkája kódinstrumentálás segítségével oldotta meg C++ template metaprogramok nyomkövetését [11]. A megoldás fordítófüggetlen volt, amennyiben a C++ forráskódba olyan szabványos C+ template-eket injektált, amelyek példányosítása jól definiált warning-okat generált. Ezeket összegyűjtve és elemezve rekonstruálhatjuk a template metaprogram végrehajtását, az egyes osztályok példányosításának sorrendjét. A módszer hátránya, hogy minden beinjektált template csak egyszer példányosul egy adott template paraméter esetében, így a memoization-ök nem detektálhatók ezen megközelítés segítségével. További probléma, hogy a warning-ok keletkezésének idejét csak rossz pontossággal lehet megállapítani (a fordítóprogram által törölt kibocsájtásuk után), ezért ez a módszer profiling adatok gyűjtésére alkalmatlan.

Később Watanabe Python nyelven készített el hasonló kódinstrumentáló eszközt. A fentebb említett problémák miatt azonban ez az eszköz sem terjedt el.

6. Összefoglalás

A C++ template metaprogramok egyre szélesebb körben használhatók, különösen az újrafelhasználható, az adott környezethez alkalmazkodó ún. aktív könyvtárak fejlesztésekor. Mivel a template metaprogramokat a fordítóprogram hajtja végre fordítási időben, az esetlegesen elkövetett hibák nehezen felismerhetők, behatárolhatók és javíthatók. Sajnos ebben a tevékenységben jól kialakított támogató eszközrendszer sem létezik. A szerző munkája ilyen eszközrendszer kialakítására és elterjesztésére irányul.

A Templight rendszer a népszerű clang fordítót egészíti ki patch formájában. Miközben a fordító működik, a kiegészítések feljegyzéseket készítenek az egyes template-ekkel kapcsolatos akciókról, mint pl. példányosítás kezdete és vége, memoization, melyeket időbélyeggel is ellátnak. A fordítás befejeztével a feljegyzett akciók változatos formátumokban írhatók ki. Az így kinyert adatok pontos képet adnak a fordítás során végrehajtott template metaprogramokról.

A nyomkövetés eredményét kliens programok elemezhetik. Két kliens programot a Templight rendszer is felajánl. A Templar egy, a futási idejű programoknál megszokott interaktív debugger programhoz hasonlít. Megjeleníti a forrásfájlt és gráf formátumban a példányosítási fákat is. A felhasználó lépésről lépésre rekonstruálhatja a program template-ekkel kapcsolatos tevékenységét. Megszakítási pontok és szűrők egészítik ki a funkcionalitást. A másik program segítségével a fordítási idők tanulmányozhatóak változatos módokon.

Hivatkozások

- [1] D. Abrahams, A. Gurtovoy, *C++ template metaprogramming, Concepts, Tools, and Techniques from Boost and Beyond*, Addison-Wesley, 2004.
- [2] D. Abrahams, C. P. Coelho, Effects of Metaprogramming Style on Compilation Time, 2001,
http://users.rcn.com/abrahams/instantiation_speed
- [3] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley, 2001.

- [4] ANSI/ISO C++ Committee, Programming Languages – C++, ISO/IEC 14882:1998(E), American National Standards Institute, 1998.
- [5] K. Czarnecki, U. W. Eisenecker, *Generative Programming: Methods, Tools and Applications*, Addison-Wesley, 2000.
- [6] R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, J. Willcock, A Comparative Study of Language Support for Generic Programming, *Proceedings of the 18th ACM SIGPLAN OOPSLA*, 2003, pp. 115–134.
- [7] Z. Juhász, Á. Sipos, Z. Porkoláb, Implementation of a Finite State Machine with Active Libraries in C++, *Generative and Transformational Techniques in Software Engineering II, International Summer School*, Braga, Portugal, 2007, LNCS 5235, pp. 474–488.
- [8] B. Karlsson, *Beyond the C++ Standard Library, A Introduction to Boost*, Addison-Wesley, 2005.
- [9] D. R. Musser, A. A. Stepanov, Algorithm-oriented Generic Libraries, *Software-practice and experience*, 27(7) (1994), pp. 623–642.
- [10] M. Odersky, L. Spoon, B. Venners, *Programming in Scala: A Comprehensive Step-by-Step Guide*, Artima Press, 2010.
- [11] Z. Porkoláb, J. Mihalicza, Á. Sipos, Debugging C++ template metaprograms, *Generative Programming and Component Engineering, 5th International Conference*, Portland, Oregon, 2006, pp. 255–264.
- [12] Z. Porkoláb, Á. Sinkovics, Domain-specific language integration with compile-time parser generator library, *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, Eindhoven, 2010.
- [13] D. Gregor, J. Järvi, J. G. Siek, G. Dos Reis, B. Stroustrup, A. Lumsdaine, Concepts: Linguistic Support for Generic Programming in C++, *Proceedings of the 2006 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2006.

-
- [14] J. Siek, A. Lumsdaine, Essential Language Support for Generic Programming, *Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation*, New York, pp. 73–84.
 - [15] Á. Sipos, Effective development of C++ Template Metaprograms, PhD thesis, Eötvös Loránd University, Budapest, Hungary, 2009.
 - [16] B. Stroustrup, *The C++ Programming Language, Special Edition*, Addison-Wesley, 2000.
 - [17] M. Torgersen, C. P. Hansen, E. Ernst, P. Ahe, G. Bracha, N. Gafter, Adding Wildcards to the Java Programming Language, *Proceedings of the 2004 ACM Symposium on Applied Computing*, 2004, pp. 1289–1296.
 - [18] E. Unruh, Prime number computation, ANSI X3J16-94-0075/ISO WG21-462.
 - [19] D. Vandevoorde, N. M. Josuttis, *C++ Templates: The Complete Guide*, Addison-Wesley, 2003.
 - [20] T. Veldhuizen, D. Gannon, Active libraries: Rethinking the roles of compilers and libraries, *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, SIAM Press, 1998, pp. 21–23.
 - [21] T. Veldhuizen, Using C++ Template Metaprograms, *C++ Report*, 7(4) (1995), pp. 36–43.
 - [22] T. Veldhuizen, Expression Templates, *C++ Report*, 7(5) (1995), pp. 26–31.
 - [23] T. Veldhuizen, Five compilation models for C++ templates, *Proceedings of the First Workshop on C++ Template Programming*, Erfurt, Germany, 2000.
 - [24] I. Zólyomi, Z. Porkoláb, T. Kozsik, An extension to the subtype relationship in C++, *Generative Programming and Component Engineering, Second International Conference*, LNCS 2830 (2003), pp. 209–227.

- [25] Boost Metaprogramming library,
<http://www.boost.org/libs/mpl/doc/index.html>
- [26] The clang project homepage,
clang.llvm.org
- [27] The Templight template metaprogram debugger and profiler,
<http://plc.inf.elte.hu/templight>

Megerősítéssel tanulás

Szita István*

Eötvös József Collegium**

szityu@gmail.com

1. Mi a megerősítéssel tanulás?

A *megerősítéssel tanulás* az emberi és állati tanulás pszichológiája inspirálta. A fogalmat B. F. Skinner pszichológus hozta létre az 50-es években, az emberi és állati viselkedés leírására. Elméletének, a behaviorizmusnak, központi eleme volt a *megerősítés*: ha az alany végrehajt valamilyen cselekvést, majd utána jutalmat kap, akkor a jövőben ezt a cselekvést gyakrabban/erősebben fogja végrehajtani. Ezt aztán lehet a kísérleti alany viselkedését formálni.

Mára a megerősítéssel tanulás csak nevében emlékeztet pszichológiai gyökereire (hasonlóan egy sor természet-ihlette területéhez a mesterségesintelligencia-kutatásnak, mint a neuronhálózatok, genetikai algoritmusok, hangyakolónia-optimalizálás). A diszciplína alapelve az, hogy behaviorista módon tanítsuk a számítógépet: csak egy jutalomfüggvényt adunk meg és egy algoritmust, ami mindenféle akciót kipróbál, és aztán úgy módosítja a viselkedését, hogy a több jutalommal járó akciókat gyakrabban hajtja végre.

Gyakran használt illusztratív példa a szerencsejátékos problémája, aki egy sor félkarú rabló előtt áll a kaszinóban. Mindegyik félkarú rabló kicsit más nyerési esélyeket ad: az egyik 5% eséllyel ad 10 zsetont, a másik 2% eséllyel ad 15-öt, egy harmadik meg 1% eséllyel 20 zsetont fizet és 2% eséllyel 5-öt. Mi legyen a szerencsejátékos stratégiája, ha

*Google, Inc., Zürich, Svájc

**1997–2005

szeretne minél többet nyerni, de (egyelőre) nem ismeri a gépek kifizetési sémáját?

Egy (jóval) bonyolultabb feladata sakkjátékosé: az akciójáért járó jutalmat/büntetést nem kapja meg azonnal a játékos, csak a játék végén – de a kapott jutalmat az összes többi döntése és az ellenfél döntései is befolyásolják.

A megerősítéses tanulás azt ígéri, hogy a megfelelő tanulóalgoritmus birtokában erőfeszítés nélkül megtaníthatjuk sakkozni a számítógépet, csupán értesíteni kell, hogy éppen nyert vagy veszített, aztán hátradőlni és várni, hogy megtanulja azokat az akciókat választani, amik győzelemhez vezetnek. Aztán megtaníthatjuk Go-t játszani, és Starcraftot és Settlers of Catant... Egy ilyen tanulóalgoritmus maga lenne a mesterséges intelligencia, és talán mondani sem kell, nem ismert. De sok ötletet ismerünk, amikkel *elég jó* tanulóalgoritmusokat lehet készíteni, és elég jó eredményeket lehet elérni. A cikk hátralevő része egy rövid bevezetés a megerősítéses tanulásba, ami bemutat ezen ötletek közül néhányat.

1.1. A feladat specifikációja

A megerősítéses tanulás egy meglehetősen tág feladathalmaz, amelyeknek közös jellemzője, hogy

- egy ügynök megfigyeléseket végezhet és eddigi megfigyelései alapján akciókat hajthat végre,
- az ügynök minden lépése után jutalmat (vagy büntetést) kap, ami függhet az ügynök eddigi akcióitól és a környezettől
- az ügynök célja, hogy a lehető legnagyobb összjutalmat gyűjtse

A MT, mint tudományterület célja pedig az, hogy olyan tanulóalgoritmusokat találjunk, amik hatékonyan találnak jó ügynököket.

Ezen feladathalmaz meglehetősen nehéz, több okból kifolyólag:

- **Részleges visszajelzés:** Az ügynök csak az akciójáért járó jutalmat kapja meg, de azt nem tudja meg, hogy mi lett volna a jutalom a többi akcióért, amiket nem próbált ki, így azt sem tudja meg, mi lett volna a legjobb akció – a feladat tehát nehezebb, mint a felügyelt tanításban, ahol az ügynök minden tanítómintáról tudja, hogy mi a helyes válasz.

- **Bizonytalanság:** A kapott jutalom, illetve az, hogy milyen állapotba jut az ügynök, nem csak a saját döntéseitől függ, hanem más ügynökök akcióitól (pl. egy többszemélyes játékban) vagy a véletlentől (pl. kártyajátékok)
- **Késleltetett jutalom:** Az ügynök akcióinak következménye lehet, hogy csak több lépéssel később bukkan fel. Képzeljünk el egy gomoku-játékos programot, amely követett el ugyan hibákat a játék során, de ellenfele is hibázott, úgyhogy sikeresen behúzott négy X-et egy sorba, és egyik oldalán sincs lezáró O. Pár lépéssel később behúzza az ötödik X-et egy sorban, ezzel megnyeri a játékot, azaz jutalmat kap. Melyik lépését erősítsük meg? A 4. X behúzása nyilván fontos lépés volt, de valószínűleg a harmadiké is, és az azt felvezető lépéssorozaté is, ami kikényszerítette, hogy az ellenfél máshova csoportosítsa erőit – de az elkövetett hibákat nyilván nem akarjuk megerősíteni.
- **Részleges megfigyelhetőség:** vannak olyan feladatok, ahol az ügynök nem kap meg minden lényeges információt az állapotról. Ilyenkor nem csak azt kell megtanulnia, hogy az adott helyzetben mi a legjobb döntés, hanem azt is, hogy mire érdemes emlékezni a korábbi megfigyelésekből és mit lehet kikövetkeztetni a rejtett információból. Jó példa erre a póker, ahol a játék központi eleme, hogy az ellenfelek kártyáiról következtetéseket vonjunk le a licitjeik és eddigi játéktílusuk alapján.

A feladat teljes bonyolultságában elméletileg kezelhetetlen (bizonyítható, hogy nem adható hatékony általános megoldás). Szerencsére sok esetben a feladat visszavezethető valamelyik speciális esetre.

- Ha az ügynök ismeri a teljes környezetét (melyik akció melyik állapotba visz, hol kap jutalmat), a kimenetek nem függenek a véletlentől vagy más ügynököktől, akkor a feladat lényegében a (jutalomig vezető) legrövidebb út megtalálásával ekvivalens, amit Dijkstra algoritmusával vagy heurisztikus keresőalgoritmusokkal (pl. A*) könnyen megoldhatunk.
- Ha egy játékban sikerül kiválasztanunk egy reprezentatív állapothalmazt (pl. profi játékmák állássorozatait), és egy szakértő mindegyik állapothoz elárulja az optimális(nak vélt) akciót, akkor

a teljes stratégia meghatározása (ami bármelyik állapotban tud döntést javasolni) egy klasszikus felügyelt tanítási feladat, ami pl. egy neuronhálózattal megoldható.

- Egy harmadik eset a Markov döntési folyamatoké [1], amikor az ügynök számára nincsen rejtett információ, és az ügynök állapotát csak saját döntései és véletlen faktorok befolyásolhatják (de más ügynökök nem). Markov döntési folyamatokkal pl. tudjuk modellezni az összes egyszemélyes játékot, amiben nincsen rejtett információ, vagy akár többszemélyes játékokat, ha egy kivétellel az összes játékos stratégiájára extra megszorításokat teszünk.

A Markov döntési folyamatok tehát elég általánosak ahhoz, hogy érdekes feladatokat modellezzünk vele, ugyanakkor eléggé specifikusak ahhoz, hogy létezzenek hatékony megoldóalgoritmusai. Ráadásként ezen algoritmusok (és az elemzésük) elégáns ötleteket vonultatnak fel. Mindezek következtében a megerősítéstanulás-kutatás jókora része az MDF-ekre koncentrál, olyannyira, hogy az MT-t gyakran szinonimaként kezelik MDF-ek megoldásával. Noha ez messze nem igaz, a Markov döntési folyamatokban lehet a legegyszerűbben bemutatni a megerősítéstanulás alapvető problémáit és algoritmusait, ezért ez a rövid bevezető írás csak MDF-ekkel foglalkozik. A leírtak nagy részéről bővebb információ található [1–3] könyveiben.

2. Markov döntési folyamatok

Az MT feladat Markov döntési folyamatkénti modelljében a világot két részre osztjuk: egy tanulóügynökre és az őt körülvevő környezetre. Az ügynök diszkrét időpontokban döntéseket hoz, ezeket az időpontokat $t = 0, 1, 2, \dots$ címkével jelöljük (azonban ez nem jelenti azt, hogy a döntések között ugyanannyi valódi idő telik el, egyszerűen az 1., 2., stb. döntés idejét jelölik). Minden döntés előtt az ügynök megfigyelheti a környezet aktuális állapotát, x_t -t, amely egy $\mathcal{X}$ állapothalmazból származik, majd választ egy a_t akciót az $\mathcal{A}$ akcióhalmazból (feltesszük, hogy $\mathcal{X}$ és $\mathcal{A}$ véges). A döntés hatására a környezet egy új állapotba kerül, x_{t+1} -be, és $r_{t+1} \in \mathbb{R}$ jutalmat ad az ügynöknek. Az MT feladat célja, hogy olyan döntéshozó stratégiát találjunk az ügynök számára, ami a lehető legtöbb összjutalmat gyűjti. A környezet állapota nem

változhat tetszőlegesen, csak a jelenlegi állapottól és akciótól, és esetleg a véletlentől függ, azaz létezik olyan P valószínűségeloszlás, hogy $x_{t+1} \sim P(\cdot | x_t, a_t)$. Hasonlóképpen, létezik olyan R jutalomfüggvény, hogy $r_{t+1} = R(x_t, a_t)$. Ez a két tulajdonság (hogy x_{t+1} és r_{t+1} csak a t -beli állapottól és akciótól függ) a Markov-tulajdonság, amiről a modell a nevét kapta. Az ügynök az összzutalom várhatóértékét, $E(\sum_t r_t)$ -t szeretné maximalizálni. Hogy a feladat (és az összeg) jóldefiniált legyen, feltesszük, hogy a folyamat véges sok lépés után befejeződik.

Összefoglalva, egy Markov döntési folyamat leírható egy $M = (\mathcal{X}, \mathcal{A}, x_S, P, R)$ ötössel, ahol $\mathcal{X}$ az állapottér, $\mathcal{A}$ az akciótér, x_S az ügynök kezdőállapota, $P : \mathcal{X} \times \mathcal{A} \times \mathcal{X} \rightarrow [0, 1]$ az átmenetivalószínűség-függvény és $R : \mathcal{X} \times \mathcal{A} \rightarrow \mathbb{R}$ a jutalomfüggvény.

A Markov-tulajdonság azt jelenti, hogy az ügynök számára az összes lényeges információt tartalmazza az aktuális állapot (hiszen az, hogy a jövőben hova jut és mennyi jutalmat kap, csak attól és a saját döntéseitől függ), azaz nincsen (releváns) rejtett információ. Ezért aztán feltehetjük, hogy az ügynök a döntéseiben csak az aktuális állapotot veszi figyelembe, azaz stratégiája leírható egy $\pi : \mathcal{X} \times \mathcal{A} \rightarrow [0, 1]$ függvénnyel, ahol $\pi(x, a)$ annak a valószínűsége, hogy az ügynök az x állapotban az a akciót választja. Az összes ilyen stratégia halmazát Π -vel jelöljük.

Egyelőre feltesszük azt is, hogy az állapotok és akciók száma nemcsak véges, de kezelhető méretű is, pl. a P átmenetifüggvényt reprezentálhatjuk egy $|\mathcal{X}|^2 \cdot |\mathcal{A}|$ méretű táblázattal, ami elfér a számítógép memóriájában.

2.1. Értékelőfüggvények

A π stratégia értéke $v(\pi) := E(\sum_{t=1}^{\infty} r_t \mid M, \pi)$, ahol a feltételes várhatóérték jelöli a feltevést, hogy az ügynök x_S -ből indul, π szerint választja akcióit, a környezet pedig P alapján változtatja állapotát és R szerint osztja a jutalmat. A célunk nyilvánvalóan az, hogy megtaláljuk a legjobb stratégiát, $\pi^* \in \Pi$ -t, amire $v(\pi^*) \geq v(\pi)$, $\forall \pi \in \Pi$. Első pillantásra a feladat nem egyszerű: még ha csak a determinisztikus stratégiákat nézzük is, egy $|\mathcal{A}|^{|\mathcal{X}|}$ méretű halmaz elemeit kell végignézni, kiszámolni a várhatóértékeket és kiválasztani a legnagyobbat. Szerencsére az MDF-ek erős belső struktúrával rendelkeznek, amit kihasználva célzottabban kereshetjük az optimális stratégiát. Ehhez kicsit általánosítanunk kell a stratégia értékét: $v(\pi)$ azt mondja meg, mi az összzutalom

várhatóértéke, ha x_0 -ból indulunk és végig π -t követjük, míg

$$Q^\pi(x, a) := E \left(\sum_{t=1}^{\infty} r_t \mid x_0 = x, a_0 = a, M, \pi \right)$$

annak az értéke, hogy az ügynök x -ből indul, először a -t választja, majd végig π -t követi. Defináljuk $Q^* : \mathcal{X} \times \mathcal{A} \rightarrow \mathbb{R}$ -t, mint az értékelőfüggvények felső burkolóját:

$$Q^*(x, a) := \max_{\pi \in \Pi} Q^\pi(x, a).$$

Elvben előfordulhatna, hogy Q^* maga nem értékelőfüggvénye semmilyen stratégiának (pl. mert a fenti definícióban minden (x, a) -ra más π a legjobb), de szerencsére nem ez a helyzet. Megmutatható, hogy Q^* az optimális stratégia értékelőfüggvénye, és magát az optimális stratégiát is könnyen megkaphatjuk belőle:

$$\pi^*(x, a) = \begin{cases} 1, & \text{ha } a = \arg \max_{a'} Q^*(x, a'); \\ 0, & \text{egyébként.} \end{cases}$$

Az egyszerűség kedvéért feltettük, hogy az $\arg \max$ egyértelmű (amennyiben nem az, akkor tetszőlegesen választunk az ekvivalens akciók közül).

3. Bellman optimalitási elv és az értékitéráció

A Bellman optimalitási elv szerint, ha vesszük az optimális stratégiát egy (x, a) párból, és az áthalad egy (x', a') páron, akkor a stratégiának (x', a') -ből is optimálisnak kell lennie. Az elv igazsága könnyen végiggondolható, de következményei messzehatók: az (x, a) -beli optimális politika megkapható úgy, hogy végignézzük az összes (x', a') párt, ahova egy lépéssel juthatunk, és kiválasztjuk azt az utat, amelyik a maximális jutalmat adja:

$$Q^*(x, a) = E \left(\max_{a_1} (R(s, a) + Q^*(x_1, a_1)) \mid x_0 = x, M \right).$$

Kihasználva, hogy M egy MDF és x_1 eloszlását leírja az átmenetifüggvény, kifejezhetjük a várhatóértéket:

$$\begin{aligned} Q^*(x, a) &= \sum_{y \in \mathcal{X}} P(y|s, a) \left(\max_{a_1} (R(x, a) + Q^*(y, a_1)) \right) \\ &= R(x, a) + \sum_{y \in \mathcal{X}} P(y|s, a) \max_{a' \in \mathcal{A}} Q^*(y, a'). \end{aligned}$$

A fenti rekurzív egyenletrendszer ($|\mathcal{X}| \cdot |\mathcal{A}|$ nemlineáris egyenlettel) a Bellman-egyenlet. Megmutatható, hogy (az általunk tett feltételek mellett, úgymint véges állapot- és akciótér, véges epizódok) egyértelmű megoldása van. A Bellman-egyenlet megoldása egyben az MT-probléma megoldása is Markov döntési folyamatokra.

A Bellman-egyenlet rekurzív, azaz kifejtésében önmaga is szerepel – ami első pillantásra nem tűnik túl hasznosnak a megoldhatóság szempontjából. De ha az egyenlet jobboldalába egy közelítő megoldást helyettesítünk, akkor a baloldalon egy olyan közelítő megoldást kapunk, ami pontosabb. Ezt az ötletet használja az értékiteráció (0. algoritmus), amely egy tetszőleges kezdő Q -függvényből kiindulva iterálja a Bellman-egyenletet.

Algoritmus 4 Értékiteráció.

```

 $Q_0(x, a) := 0 \quad \forall x \in \mathcal{X}, a \in \mathcal{A}$ 
for  $k := 1, 2, \dots$  do
  for all  $x \in \mathcal{X}, a \in \mathcal{A}$  do
     $Q_{k+1}(x, a) := R(x, a) + \sum_{y \in \mathcal{X}} P(y|x, a) \max_{a' \in \mathcal{A}} Q_k(y, a')$ 

```

Ha az MDF-ben minden epizód legfeljebb T lépésen belül véget ér, akkor az értékiteráció is legfeljebb T lépésen belül Q^* -hoz konvergál (és a további iterációk hatására nem változik). Ha nem lehet ilyen T korlátot mondani (de minden epizód véges), akkor Q_t ugyan sohasem lesz pontosan Q^* , de konvergál.

Az értékiteráció tehát megtalálja az MDF optimális stratégiáját, de messze nem a leghatékonyabb módon. Az egyik első dolog, amit észrevehetünk, hogy az érték-újraszámolások jó része felesleges. Példaként képzeljünk el egy N állapotú MDF-et, aminek az állapotai láncot alkotnak. Minden állapotban 0 jutalmat kap az ügynök, kivéve a legutolsót

(a végállapotot), ahol $+1$ -et. Egyetlen akció van, az jobbra lép, ahol lehetséges. Ezen a példán az értékiteráció minden k -ra csak egyetlen állapot értékét változtatja meg, de az összeset újrászámolja (általános esetben egyetlen iteráció végrehajtása $O(|\mathcal{X}|^2|\mathcal{A}|^2)$ időt is igényelhet).

Nincs azonban szükség arra, hogy minden (x, a) párt egyenletes gyakorisággal számoljunk újra: ha aszinkron frissítést használunk, akkor mi adunk meg egy $\{(x_k, a_k) : k = 1, 2, 3, \dots\}$ sorozatot, és ebben a sorrendben frissítjük őket. A Q -függvények indexét elhagyjuk, mert a gyakorlatban úgyis felesleges a legfrissebb verzió kívül mást számon tartani. Az algoritmus így alakul: (0. algoritmus)

Algoritmus 5 Aszinkron értékiteráció.

```

 $Q(x, a) := 0 \quad \forall x \in \mathcal{X}, a \in \mathcal{A}$ 
 $k := 0$ 
while van_még_minta do
   $(x_k, a_k)$  minta beolvasása
   $Q(x_k, a_k) := R(x_k, a_k) + \sum_{y \in \mathcal{X}} P(y|x_k, a_k) \max_{a' \in \mathcal{A}} Q(y, a')$ 
   $k := k + 1$ 
  
```

Ha minden (s, a) pár végtelen sokszor szerepel a frissítendő párok sorozatában, akkor bebizonyítható, hogy Q Q^* -hoz konvergál. A konvergencia sebessége nyilván nagyban függ attól, hogy milyen sorrendet határoozunk meg. Egy gyakran használt heurisztika szerint ha egy $Q(x, a)$ érték nagyot változott, akkor érdemes mindazon párokat frissíteni, amelyekből nagy eséllyel lehet x -be jutni. A prioritizált bejárás algoritmus is ezt az elvet alkalmazza: a frissítendő állapot-akciópárokat egy prioritásos sorban tároljuk. Minden lépésben a legnagyobb prioritású (x, a) párt frissítjük, feljegyezzük a módosítás $\Delta_{x,a}$ nagyságát, majd minden olyan (z, a') párt a sorba illesztünk, amiből x -be lehet jutni, mégpedig $P(x|z, a') \cdot |\Delta_{x,a}|$ prioritással (ha már szerepelt, akkor ennyivel növeljük a prioritását) [4].

4. Tanulás generatív modellel

A következő eset, amire az értékiteráció alkalmazhatóságát kiterjesztjük, az ismeretlen környezet esete. Elég nagyot sumákkoltunk ugyanis, amikor azt állítottuk, hogy az értékiteráció megoldja az MT feladatot

(legalábbis MDF-ekben): képleteink felhasználják P -t és R -et, pedig az MT protokollban sehol nem esik szó arról, hogy ezt elárulja nekünk a környezet, csak lépésenként egyetlen mintát a $P(\cdot|x_t, a_t)$ valószínűséggel, illetve egyetlen állapot-akciópár jutalmát. Csomó esetben (pl. egyszerű játékok, ahol ismerjük a szabályokat) ez nem gond, de amikor pl. az MDF egy közelítő modellje egy bonyolult valósidejű stratégiai játéknak, akkor egyáltalán nem remélhetjük, hogy P -t és R -et meg tudjuk szerezni.

4.1. Kitérő: sztochasztikus átlagolás

Tegyük fel, hogy van egy valós értékű véletlen változónk, Z , $\bar{z}$ várhatóértékkel. $\bar{z}$ -t nem ismerjük, de szeretnénk minél pontosabban közelíteni. Ehhez mintákat kaphatunk Z -ből: $z_1, z_2, \dots \sim Z$. A feladat megoldása egyszerű, ha n mintánk van, az átlaguk,

$$\hat{z} = \frac{1}{n} \sum_{i=1}^n z_i$$

jó közelítés, ha n elég nagy (és Z teljesít bizonyos feltételeket, pl. véges a szórása). Ha minden egyes minta után szeretnénk új, pontosabb becslést kapni, az inkrementális frissítési szabály is könnyen felírható: legyen $\hat{z}_0 := 0$, aztán $k > 0$ -ra

$$\hat{z}_k := (1 - \alpha_k) \cdot \hat{z}_{k-1} + \alpha_k \cdot z_k, \quad (1)$$

ahol $\alpha_k = 1/k$.

4.2. Értékiteráció mintavételezéssel

A minták alapján – a kitérőben vázolt módon – közelítőleg kiszámolhatjuk az MDF paramétereit, majd megoldjuk a közelítő MDF-et. Tegyük fel, hogy egy (x, a) párhoz van $n(x, a)$ megfigyelésünk, hogy milyen következő állapotba jut a környezet, jelöljük ezeket $y_1, y_2, \dots y_{n(x,a)}$ -val. Ezekből kiszámolhatjuk az átmenetifüggvény egy $\hat{P}$ közelítését, amit aztán beledughatunk az értékiterációba (0. algoritmus).

Algoritmus 6 Értékiteráció mintavételezéssel.

```

bemenet:  $n(x, a)$ , a gyűjtendő minták száma  $\forall x \in \mathcal{X}, a \in \mathcal{A}$ -ra
 $n(x, a, y) := 0 \quad \forall x, y \in \mathcal{X}, a \in \mathcal{A}$ 
// Mintagyűjtés
for  $x \in \mathcal{X}, a \in \mathcal{A}$  do
    for  $i := 1$  to  $n(x, a)$  do
        mintagyűjtés:  $y_i \sim P(\cdot|x, a)$ 
         $n(x, a, y_i) := n(x, a, y_i) + 1$ 
         $\hat{P}(y|x, a) := n(x, a, y)/n(x, a)$ 
         $Q_k(x, a) := 0$ 
// Értékiteráció
for  $k := 1, 2, 3, \dots$  do
    for  $x \in \mathcal{X}, a \in \mathcal{A}$  do
         $Q_{k+1}(x, a) := R(x, a) + \sum_{y \in \mathcal{X}} \hat{P}(y|x, a) \max_{a' \in \mathcal{A}} Q_k(y, a')$ 

```

4.3. Inkrementális értékiteráció mintavételezéssel

Ha inkrementális módon szeretnénk a frissítést végezni, egy apró trükköt kell alkalmaznunk: nem közvetlenül a $P(y|x, a)$ értékeket becsüljük, hanem $Z = R(x, a) + \max_{a' \in \mathcal{A}} Q(y, a')$ helyettesítést használunk, ahol x, a és Q adott, és y valószínűségi változó P eloszlással. Ekkor $E(Z) = R(x, a) + \sum_{y \in \mathcal{X}} P(y|x, a) \max_{a' \in \mathcal{A}} Q(y, a')$, éppen a becsülni kívánt mennyiség. Ez adja a 0. algoritmust.

Algoritmus 7 Inkrementális értékiteráció mintavételezéssel.

```

 $n(x, a) := 0 \quad \forall x \in \mathcal{X}, a \in \mathcal{A}$ 
 $Q(x, a) := 0 \quad \forall x \in \mathcal{X}, a \in \mathcal{A}$ 
for  $k := 1, 2, 3, \dots$  do
    bemenet:  $x_k, a_k$ 
     $n(x_k, a_k) := n(x_k, a_k) + 1$ 
    mintagyűjtés:  $y_k \sim P(\cdot|x, a)$ 
     $Q(x_k, a_k) := (1 - \alpha_{n(x_k, a_k)}) \cdot Q(x_k, a_k)$ 
         $+ \alpha_{n(x_k, a_k)} \cdot (R(x, a) + \max_{a' \in \mathcal{A}} Q(y_k, a'))$ 

```

Az algoritmus működésével kapcsolatban néhány megjegyzést kell tennünk. Először is, $|\mathcal{X}| \cdot |\mathcal{A}|$ különböző becslés folyik párhuzamosan,

ezért mindegyikhez külön számon kell tartani az α tanulási rátát, illetve az ezt meghatározó $n(x, a)$ számlálót.

Másodszor, bőven túlléptük a kitérőben bemutatott egyszerű sztochasztikus átlagolás határait. A képletben szereplő $R(x, a) + \max_{a' \in \mathcal{A}} Q(y_k, a')$ -ban a Q -értékek folyamatosan változnak a párhuzamosan folyó becslések miatt, úgyhogy meglehetősen bonyolult bizonyítani, hogy az algoritmus továbbra is konvergál, ha minden x, a párt végtelen sokszor frissítünk – és a konvergencia sebességéről nem tudunk semmit bizonyítani.

Végül pedig érdemes észrevenni, hogy az $\alpha_n = 1/n$ tanulási ráta az összes mintát ugyanolyan súllyal veszi figyelembe, de esetünkben érdemes lenne az újabb mintáknak nagyobb súlyt adni, hiszen azok újabb (tehát pontosabb) Q értéket használnak. Érdekes módon a konvergenciát minden olyan tanulásiráta-sorozatra be lehet bizonyítani, amire $\sum_{n=1}^{\infty} \alpha_n = \infty$, de $\sum_{n=1}^{\infty} \alpha_n^2 < \infty$, tehát elég akár $\alpha_n = 1/n^{0.5+\epsilon}$ mértékben csökkenteni.

Gyakorlati alkalmazásokban az $\alpha_n = 1/n$ ráta elfogadhatatlanul lassú konvergenciát eredményez, és gyakran konstans tanulási rátát alkalmaznak (amire természetesen nem igaz az aszimptotikus konvergencia, de gyorsabban vezet eredményre).

5. Tanulás ismeretlen környezetben

Továbbra is feltesszük, hogy a környezet egy MDF, de semmilyen más ismeretet nem tételezünk fel. Az ügynök tehát nem ismeri a P átmenetifüggvényt, és nem tudja megkérdezni a környezetet, hogy tetszőleges x, a állapot-akció párból hova jutna. A környezettel kizárólag a MT protokoll szerint tud kölcsönhatni, azaz (1) megfigyelheti az aktuális állapotát, (2) végrehajthat egy akciót (aminek hatására valamilyen jutalmat kap és új állapotba kerül), végül (3) új epizódot kezdhet, ha a régi véget ért. Ekkor egy új, fogós problémával találjuk szembe magunkat: ahhoz, hogy az ügynök mintát tudjon gyűjteni egy (x, a) párról, oda kell jutnia, de ahhoz, hogy oda tudjon jutni, megbízható becslése kell hogy legyen P -ről (amihez minták kellenek)

Illusztrációként képzeljünk el egy 10×10 -es négyzetrácsot, amelyben az ügynök az $(1, 1)$ -es pozícióból indul, és a rácsvonalak mentén mozoghat – de a kiválasztott akciója helyett 50% eséllyel egy véletlen-szerű másik akció hajtodik végre. A $(10, 10)$ -es pozíció a cél, ahol +100

jutalmat kap az ügynök és véget ér az epizód. Minden más állapotban –1 „jutalmat” kap. Az ügynöknek kezdetben fogalma sincs, merre van a cél, valamilyen módon fel kell derítenie a környezetet. A gyakorlatban használt felfedező stratégiákat két nagy csoportba lehet osztani: véletlen bolyongás-jellegűek, illetve módszeres felderítés. Az utóbbi kategória nyilván sokkal hatékonyabb, ám bonyolultabb is megvalósítani – így a jelenlegi írás csak a legegyszerűbb képviselőjüket, az optimista inicializálást tárgyalja.

Definiáljunk egy $\pi_e : \mathcal{X} \times \mathcal{A} \rightarrow [0, 1]$ explorációs stratégiát, aminek a célja, hogy a lehető legtöbb állapotba eljuttassa az ügynököt. A legegyszerűbb választás a véletlen bolyongás, amikor minden akciót egyforma valószínűséggel választunk. Egy explorációs stratégia birtokában felírhatunk egy tanulóalgoritmust az általános MT feladatra – amit Q -tanulásnak neveznek (0. algoritmus).

Algoritmus 8 Q -tanulás általános explorációs stratégiával

```

bemenet:  $\pi_e$ 
 $n(x, a) := 0 \quad \forall x \in \mathcal{X}, a \in \mathcal{A}$ 
 $Q(x, a) := 0 \quad \forall x \in \mathcal{X}, a \in \mathcal{A}$ 
repeat
   $x := x_S$ 
  repeat
    akcióválasztás:  $a \sim \pi_e(x, \cdot)$ 
     $n(x, a) := n(x, a) + 1$ 
     $y :=$  következő állapot a környezettől
     $Q(x, a) := (1 - \alpha_{n(x, a)})Q(x, a) + \alpha_{n(x, a)} \cdot (R(x, a) +$ 
       $\max_{a' \in \mathcal{A}} Q(y, a'))$ 
     $x := y$ 
  until nem epizod_vege( $y$ )
until tanulas_tart

```

5.1. Egyszerű explorációs stratégiák

A legegyszerűbb explorációs stratégia, ha minden akciót ugyanolyan valószínűséggel választunk, $\pi_e(x, a) = 1/|\mathcal{A}|$. Ez a stratégia kilégíti az explorációval szemben támasztott alapvető követelményeket – minden akciósorozat (köztük az optimáli stratégia is) nemnulla valószínűséggel

hajtódik végre, és ha az MDF egy állapotába lehetséges eljutni, akkor a véletlen bolyongás előbb-utóbb el fog jutni oda. A Q-tanulás véletlen bolyongással konvergál az optimális értékelőfüggvényhez. Nem túl meglepő módon, a stratégia nem túl hasznos a gyakorlatban.

Képzeljünk el egy ügyességi játékot (pl. egy Super Mario-szerű platformert), ahol úgy lehet célba érni, ha az ügynök pontosan végrehajt egy 10-lépéses akciósorozatot (pl. akadályokon, szakadékokon ugrál át), és ha téveszt, kezdheti előlről. Ha négy lehetséges akció van, akkor az ügynök $1/4^{10}$ eséllyel éri el a célt, vagyis átlag egymillió próbálkozást kell végeznie, amíg egyetlen mintát szerez a célállapotról, és egyáltalán esélye lesz elkezdeni tanulni. A konvergencia a legegyszerűbb feladatokat kivételével elfogadhatatlanul lassú.

Másik végtelként választhatjuk azt a stratégiát, hogy egyáltalán nem explorálunk, hanem mindig az éppen legjobbnak tűnő akciót választjuk:

$$\pi_e(x, a) = \begin{cases} 1, & \text{ha } a = \arg \max_{a'} Q(x, a'); \\ 0, & \text{egyébként.} \end{cases}$$

Ezt mohó stratégiának hívják, és hasonlít az optimális stratégia definíciójához – csak az optimális értékelőfüggvény helyett az aktuális becslést követi „mohón”, így lehet, hogy kihagyja a hosszú távon legkifizetődőbb akciót. Példaként képzeljük el az ügynököt, amint félkarú rablókkal játszik, és mindegyik gépet kipróbálja egyszer. Ha az egyik gépen nyer valamennyit, a többin meg nem, akkor ezután kizárólag azon a gépen fog játszani, pedig lehet, hogy a többinek nagyobb lenne a hosszútávú nyeresége.

A mohó stratégia, azaz az exploráció hiánya, nem túl meglepő módon elég rossz explorációs stratégia – de érdemes számon tartani, mint a véletlen bolyongással szembeni másik végtetet. A két extrémum egy lehetséges kombinációja az ϵ -mohó stratégia, ami $0 \leq \epsilon \leq 1$ eséllyel egy véletlen akiót választ, $1 - \epsilon$ eséllyel pedig a mohó akciót. Az ϵ -mohó stratégia ideális esetben az állapottér érdekes részeire koncentrálja az explorációt. pl. a négyzetrácsos feladatban tegyük fel, hogy az ügynök hosszas bolyongás után rátalált egy útra, ami a célhoz vezet, de meglehetősen kacskaringós. Az ϵ -mohó stratégia nagyrészt a megtalált utat fogja követni, de néha megpróbál letérni róla, így az ügynök rátalálhat egy-egy kanyarlevágásra.

Az ϵ -mohó stratégia az egyik legegyszerűbb módszer, amivel az ügynök el tudja végezni a szükséges felderítést, és ennek hatására a

legnépszerűbb is. Ez azonban messze nem jelenti azt, hogy hatékony lenne, éppúgy képes arra, hogy lokális optimumokban megragadjon, mint a mohó stratégia.

5.2. Optimista kezdőértékek

Az összes eddig bemutatott algoritmusban az értékelőfüggvényeket 0-ra inicializáltuk. Természetesen bármilyen más kezdőértéket is használhattunk volna, ez nem befolyásolná a végeredményt, ahova konvergálunk. Megváltozik a helyzet, amikor az ügynök a Q függvényt a tanulás során is használja (hogy irányítsa, melyik állapotokról szerezzen információt), azaz amikor mohó vagy ϵ -mohó stratégiát használ. Például ha az optimális értékelőfüggvénnyel inicializálnánk, $Q(x, a) := Q^*(x, a)$, akkor egyből az optimális útvonal mentén kezdene el az ügynök tapasztalatot gyűjteni, lerövidítve a tanulási időt. Természetesen ha ismernénk Q^* -ot, az egész tanulási folyamat teljesen felesleges lenne.

Ha semmilyen előzetes információnk nincs, akkor is segíthetjük a tanulást megfelelő inicializálással. Ennek belátásához gondoljuk végig, milyen hatással van a kezdeti Q -érték az ügynök döntéseire. Ha $Q(x, a)$ kezdőértéke nagyon alacsony, akkor az első akció értéke, amit az ügynök egy állapotban kipróbál, biztosan nőni fog – így amikor legközelebb ugyanoda jut az ügynök, megint ugyanazt az akciót választja, újra és újra. Sokkal izgalmasabb viselkedéshez jutunk, ha $Q(x, a)$ kezdőértéke nagyon magas: az első kipróbált akció „csalódást” fog kelteni, azaz a kapott jutalom kisebb lesz, mint az elvárás. Az ügynök tehát újabb akciókat fogkipróbálni, és általában, minél kevesebbszer választott egy akciót, annál szívesebben választja újra (hiszen ezen akciók becsült értéke még nem sokszor csökkent). Ez éppen az elérni kívánt viselkedés. Mennyire legyen nagy a Q -függvény kezdőértéke? Ha sokkal nagyobb, mint a valódi állapotértékek, akkor sok felderítőakciót végez az ügynök, alaposan feltérképezi az állapotteret. Ugyanakkor tovább tart, amíg a $Q(x, a)$ értékek $Q^*(x, a)$ -hoz konvergálnak, egyszerűen azért, mert sokat kell változniuk.

A fent leírt trükköt optimista inicializációnak hívják, és annyira egyszerű, hogy sokszor nem is tekintik az exploráció részének, annak ellenére, hogy meglehetősen hatékony. Zárásképpen álljon itt a Q -tanulás optimista kezdőértékekkel és ϵ -mohó stratégiával (0. algoritmus).

Algoritmus 9 Q-tanulás optimista kezdőértékekkel és ϵ -mohó stratégiával.

bemenet: M , optimista kezdőértékbecslés
 bemenet: ϵ , véletlen bolyongás aránya
 $n(x, a) := 0 \quad \forall x \in \mathcal{X}, a \in \mathcal{A}$
 $Q(x, a) := M \quad \forall x \in \mathcal{X}, a \in \mathcal{A}$
repeat
 $x := x_S$
 repeat
 akcióválasztás: $a := \arg \max_{a'} Q(x, a')$
 $1 - \epsilon$ eséllyel $a :=$ uniform véletlen akció $\mathcal{A}$ -ból
 $n(x, a) := n(x, a) + 1$
 $y :=$ következő állapot a környezettől
 $Q(x, a) := (1 - \alpha_{n(x,a)})Q(x, a) + \alpha_{n(x,a)} \cdot (R(x, a) + \max_{a' \in \mathcal{A}} Q(y, a'))$
 $x := y$
until nem epizod_vege(y)
until tanulas_tart

6. Lezáró gondolatok

Ez a cikk egy rövid bevezető a megerősítéses tanulás elméletébe, amiben eljutottunk a probléma megfogalmazásától egy megoldóalgoritmusig, ami képes az általános MT feladat megoldására (abban a speciális esetben, ha a környezet egy Markov döntési folyamat). Ez azonban nem a lezárás, hanem csak a kezdet, ha úgy tetszik, kedvcsináló.

Lehet-e Q-tanulásnál hatékonyabban tanulni MDF-ekben? (Lehet, a kulcs az okos exploráció.) Mit lehet tenni, ha túl nagy az állapottér? Hogyan alakulnak át a tanulóalgoritmusaink, ha nem tároljuk egyenként minden (x, a) pár értékét, hanem csak approximáljuk egy kisebb paraméterhalmaz hangolásával? Mit lehet tenni, ha az információ részben rejtett az ügynök elől? Mit lehet tenni, ha a környezetben más ügynökök is tevékenykednek (pl. próbálják legyőzni a tanulóügynököt)? Muszáj-e értékelőfüggvényeket használni a megerősítéses tanulásban, vagy anélkül is lehet stratégiát optimalizálni? Ha valaki elének tesz egy feladatot, hogyan érdemes az MT keretei között modellezni (hogyan definiáljuk az állapotokat? a jutalmakat?) Vannak-e sikeres alkalmazásai

a megerősítéses tanulásnak? Rengeteg kérdés, amiknek jó része aktív kutatási terület. [2, 3] könyvei remek áttekintést adnak a további kutatáshoz.

És természetesen nem esett szó a cikkben szereplő állítások elméleti alátámasztásáról (mely algoritmusok konvergálnak, hogyan lehet ezt megmutatni), amely önmagában megtölt egy könyvet [1].

Hivatkozások

- [1] D. P. Bertsekas, J. N. Tsitsiklis, *Neuro-Dynamic Programming*, Athena Scientific, 1996.
- [2] R. S. Sutton, A. G. Barto, *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, 1998.
- [3] Cs. Szepesvári, *Algorithms for reinforcement learning*, Vol. 4, Morgan & Claypool Publishers, 2010.
- [4] M. A. Wiering, J. Schmidhuber, Efficient model-based exploration, *Proceedings of the Fifth International Conference on Simulation of Adaptive Behavior: From Animals to Animats 5*, 1998, pp. 223–228.

Egy pályaelhagyó vallomása

Végh Zoltán*

Eötvös József Collegium**

vozoskoda@gmail.com

2004-ben alig páran üldögéltünk az Eötvös Collegium 018-as termében, és vártuk Csörnyei Zoltánt, az újonnan alakuló Informatikai Műhely leendő műhelyvezető tanárát, hogy megtartsuk az első műhelygyűlést. Az ELTE Informatikai kara még nagyon fiatal volt, én 2002-ben még a TTK-n kezdtem meg programtervező matematikusi tanulmányaimat. Ebből kifolyólag elsőéves collegistaként a Természettudományi Műhely óráira kellett járnom, és nagyon vártam, hogy ez az állapot megváltozzon.

Az ELTE-n informatikát tanuló emberek halmaza akkor még egyet jelentett a mi szakunkkal, megtűrt zárvány voltunk a TTK-n belül. Az oktatáson is érezhető volt a természettudományos megközelítés: egész első évben nem volt olyan kötelező kurzusunk, ami teljesítéséhez le kellett volna ülni gép elé.

Azóta eltelt 10 év, két generáció töltötte collegista éveit az Infó-műhely tagjaként, az egyetemeken megváltozott az informatikai oktatás, több lehetőség van egyénre szabottabb képzést választani, több kurzus közül. Ettől függetlenül valószínűleg ma is kihullanék a sorból.

Az ok teljesen egyszerű, lehetőségem van egy olyan területen dolgozni, ami jobban lázba hoz, és kevesebb emberrel kell versenyezni. Ez a film-, vagy tágabb értelemben a mozgóképgyártás.

A gyakorló programozókéhez képest kevés, de az átlagoshoz képest magasabb informatikai képzettség végig segített, hogy feljebb lépkedjek a filmes ranglétrán. Ez is egyfajta interdiszciplinaritás... Most a TV2 egyik napi sorozatának (Magánnyomozók) vagyok a rendezője.

*Constantin Entertainment, TV2; Budapest

**2002–2008

Mivel az egyetemről való távozásom után informatikához köthető eredményt vagy kutatást nem produkáltam, nem is éreztem indokoltnak írni ebbe a kötetbe. Csörnyei tanár úr kitartó unszolása is közrejátszott abban, hogy újra átgondolva mégis úgy látom, hogy a műhely (időrendben) első titkáráként illik pár sort írnom arról, hogy milyen és mekkora gellert kapott a pályám. Jelzem, hogy írásom tudományos értéke várhatóan csekély.

Már gimnázium alatt forgattam amatőr filmeket, és ezt egyetem közben is folytattam. Ezeket saját erőforrásaim (leginkább a tanulásra való időm) felhasználásával készítettem el. Filmfesztiválokra neveztem a videóimat, ahol filmes iskolák vezetőivel is megismerkedtem.

Így másodéves egyetemistaként elvégeztem egy mozgóképgyártó OKJ-s képzést is, aminek a tandíját azzal váltottam ki, hogy rendszergazda és stúdióvezető lettem az iskolánál. Analízis és bevmat helyett Angyalföldön hallgattam dramaturgiát és kameratechnikát.

Különösen emlékezetes számomra a Bevezetés a programozásba II. tárgy, amiből negyedik nekifutásra sikerült átmennem. Ebben az időben a bölcsészkar filmszakos hallgatóinak vizsgafilmeiben voltam operatőr és vágó. Első alkalommal még csak egy műben, később szemeszterenként hat filmben működtem ilyen módon közre. Ezek a segítségek sajnálatos módon minden évben egybeestek a Bevprog II vizsgára való felkészülésre szánt időmmel. . .

Az egyetemre jutó egyre kevesebb időm mellett további akadály volt az informatika mélységeiben való elmerüléshez vezető úton, hogy a fővárosi élethez munkát kellett vállalnom, az ösztöndíj nem biztosította a megélhetésemet. Az egyetemi működéssel ellentétben a videós iskolában mintatanuló voltam. Ennek köszönhettem, hogy a Nap-Kelte nevű reggeli műsornál el tudtam helyezkedni, mint lótifuti technikus. Országos tévéműsornál dolgozni mámorító volt, testközelből látni a „profikat”. Akármennyire is kis feladataim voltak (leginkább kábelhúzógatás), próbáltam megfelelően ellátni őket.

Szerettem volna továbblépni, bár még nem volt tiszta, hogy merre. Eltelt pár év, amikor jött egy alkalom, hogy egyetlen vágó-kolléga sem ért rá egy feladatra. Azt fillentettem a gyártásvezetőnek, hogy tudom kezelni az ezeréves Beta vágópultot, így megkaptam a munkát. Egy éjszakám volt megtanulni a rendszer kezelését, hajnalban már élesben kellett dolgozni rajta. Ha nem debugolom annyi hibás kódot a Coliszobában, akkor sosem jövök rá, hogy melyik lekopott feliratú gomb

éppen mit csinál. Így viszont vágó lettem az m1-nél.

2009-ben megszűnt a Nap-Kelte, a hallgatói státuszommal együtt. A hirtelen nyert szabadidőmet blogolásra fordítottam. Pusztán a saját szórakoztatásomra politikai akcióba kezdtem, kis eufemizmussal úgy fogalmaznék, hogy az érvénytelen szavazásra buzdítottam az olvasóimat. A blog váratlanul sikeres lett. Ennek köszönhetően megkeresett egy reklámügynökség munkát ajánlva: legyek közösségi média szakértő.

Természetesen elvállaltam, bár fogalmam sem volt, hogy mit kellene csinálnom. Szerencsémre a főnökeimnek sem. Kis mellékszál: az ilyen munkahelyek nagy részénél mindenkinek Macbookja van, de a Chrome telepítés legtöbbjükön kifog. Így nem volt nehéz nélkülözhetetlenné válni.

A közösségi média-biznisz nem jött be a cégnek, viszont reklámfilmes megrendelések voltak bőven. Eddigre már volt annyi tapasztalatom a saját amatőr filmes projektjeimből, hogy rendezőasszisztensként tudjam folytatni náluk.

Körülbelül ebben az időben gondolkodtam el komolyan azon, hogy mit akarok kezdeni magammal. Tudtam, hogy programozó belátható időn belül nem leszek, ahhoz programoznom kellene. Filmesként viszont tapasztalt operatőr és vágó voltam, de mikrofonozást, világosítást vagy akár forgatókönyvírást is végeztem már. Egyik területen sem jutottam el a szakma csúcsára, viszont a feladatok egymáshoz és a nagy egészhez való viszonyát elég jól átláttam. Szerintem ehhez is szükséges az a fajta gondolkodás, ami egy megfelelően működő programkód létrehozásához és menedzseléséhez kell.

Ha az egyes részlegeket (hangosztály, világosítók, stb.) úgy képzeljük el, mint C++ osztályokat, a bennük dolgozó embereket pedig mint objektumokat, akkor a velük való munka és folyamatszervezés leegyszerűsödik. Tisztázni kell, melyek a belső változók, melyek érhetőek el kívülről, és milyen interfészen keresztül kommunikálhatunk velük. Minden filmes projekt más, ahhoz, hogy a feladathoz igazítsuk a „programot”, az osztályokat örököltetjük.

Természetesen nem hagyható figyelmen kívül, hogy ebben az esetben hús-vér emberekről van szó, a pszichológia is közrejátszik az együttműködésnél. A korábbi hasonlatot úgy lehetne folytatni, hogy a különböző objektumok példányosításkor bugosak lesznek, változik a paraméterezhetőség, vagy akár az objektumokon végrehajtható függvények. Ezek pedig jellemzően nem jól dokumentált változások,

felismerésük időbe telik.

A következő időszakban megfogalmaztam rendezői ambícióimat, és egyre több alkalmat kaptam ennek gyakorlására. Ez addig tartott, amíg „felül nem definiálták az operátoraimat”. Meghívtam a destruktoromat – kiléptem a cégtől.

Előlről kezdtem egy másik programban belépő szinten. A TV2 egyik reality műsorában lettem gyártási asszisztens. Eddigre már volt elég referenciám ahhoz, hogy pár hónap múlva kapjak egy próbalehetőséget megmutatni magamat rendezőként.

Az első forgatási napon összetört a kamera. A főszereplő egy kerekesszékekben játszott. Ahogy megérkezett a cserekamera, kitört a szék kereke. A stáb romlott ételt kapott ebédre, így másnap elég nehezen ment a munka. Az, hogy a buszunk sofőrje nekiment egy villanyoszlopnak, már apróság volt ezekhez képest. Viszont az epizód a legnézettebb műsor volt a vetítése estéjén, megverve a konkurenciát is. Maradhattam rendező, azóta is ezt csinálom.

Nagyon sokan kérdezték már tőlem, hogy nem sajnálom-e az „elvesztegetett” éveimet, amit programozónak tanulva töltöttem el. A válaszom egyértelműen nem. Ezt az időszakot magánéletileg is fontosnak és boldognak mondhatom, emellett analitikus és strukturált gondolkodású emberek között töltöttem el, amiért csak köszönetet mondhatok a sorsnak. Legalábbis én szeretem így hinni.

Névmutató

B

Balassi Márton 49
Bencs Ferenc 18
Bodó Zalán 61
Bozó István 79

C

Czigola Gábor 93

CS

Cséri Tamás 35
Csörnyei Zoltán 9

E, É

Englert Péter 18

G

Gévay Gábor 18
Gilián Zoltán 108
Góbi Attila 118
Grósz Tamás 139

H

Horváth Gábor 155
Horváth László 5

K

Kaposi Ambrus 166
Kovács Györgyi 197
Kovács Lehel István 208
Kovács Máté 222
Kovács Péter 235
Kozár Gábor 155
Kozma László 248
Kozsik Tamás 118
Köllő Hanna 277

L

Lócsi Levente 24, 286
Lővei Péter 305

M

Manninger Mátyás 310

N

Nádor István 320
Novák Ádám 330

P

Páli Gábor János 353
Porkoláb Zoltán 373

SZ

Szita István 395
Szűgyi Zalán 155

T

Tóth László 139
Tóth Melinda 79

V

Végh Zoltán 411
Vezér Boglárka 118