DR. GREGORICS TIBOR

HABILITÁCIÓS TÉZISEK

2019.

Tartalom

1. Bevezetés

Ebben a dolgozatban a PhD fokozatszerzésem (1994) óta publikált eredményeimet foglaltam össze három, egymástól elkülönülő csoportban.

Az első témakör a mesterséges intelligencia heurisztikus gráfkereső algoritmusaihoz kapcsolódik, amelyik a doktori dolgozatom utóéletének is felfogható. Ez a kutatási terület a kilencvenes évek elejére ugyan lezárult, de mivel ezt követően találkoztam néhány olyan szoftverfejlesztési feladattal, ahol alkalmazni tudtam ezeket az ismereteket, továbbá maradtak még számomra érdekes nyitott kérdések, valamint a témakört azóta is folyamatosan oktatom, így az elmúlt évek során született néhány publikációm ebben a témában. Ezek egyrészt összefoglaló értekezések (két könyvnek fejezetei [1] [2], oktatási segédanyagok [3] [4]), másrészt folyóiratcikkek [5] [6] [7] [8], illetve konferencia kiadványok [9] [10]. Ezek eredményeit foglalom össze alább, a Heurisztikus gráfkeresés c. fejezetben.

A második témakör ahhoz a sajátos szemléletű programozási modellhez kapcsolódik, amely Hoare [11], Dijkstra [12], Gries [13] nyomán Fóthi Ákos [14] irányításával az ELTE programozáselméleti kutatások védjegyévé vált. Programozó matematikus hallgatóként az első olyan évfolyamba jártam, ahol ezt a modellt már oktatták, és tanúja voltam annak a kísérletezésnek, amely során a modell ki- és átalakult. Ez a modell nagy hatással volt rám, később ennek az oktatásába és kutatásába is bekapcsolódtam. Habár engem elsősorban a modellre épülő programozási módszertan érdekelt, de amikor a 2010-es években a Programozáselmélet (korábbi nevei: Szintézis és verifikáció) tárgy felelőse lettem, néhány észrevételt tettem közzé a modellel kapcsolatban is [15] [16]. Ezeket foglalom össze az Egységes programozási modell c. fejezetben.

A harmadik témakör a már említett programozási módszertan, azon belül a feladatoknak programozási tételekre történő visszavezetésével történő megoldása. Ezt is az oktatói munkám indukálta [17] [18], hiszen azt vizsgáltam, hogyan lehet garantáltan helyes programok előállítására tanítani a hallgatókat. Kidolgoztam a programozási tételeknek egy általánosabb családját [19], és vizsgáltam ennek felhasználását a programtervezésben [20] [21], valamint ezeknek a terveknek a konkrét programozási nyelveken történő alkalmazását [22]. Ezeket az eredményeket mutatom be a Visszavezetés felsorolókkal c. fejezetben.

Nem tudtam beilleszteni minden eredményemet a fenti témakörökbe. Ezek közül talán a legjelentősebb annak az előre csatolt mesterséges neuronhálónak a tervezése, amely segített a mediterrán fás szárú növények elterjedési és telepíthetőségi területeinek vizsgálatában, és amely egy többszerzős folyóiratcikk [23] részeként jelent meg.

2. Heurisztikus gráfkeresések

A heurisztikus gráfkeresések speciális útkereső algoritmusok. Segítségükkel azokat a feladatokat lehet megoldani, amelyek a modellezésük során leképezhetők ***útkeresési problémává***. Ez azt jelenti, hogy a feladat problématerét egy irányított gráf adott csúcsából (a startcsúcsból) kiinduló irányított útjai alkotják, és ezek között keresünk adott csúcsok (az ún. célcsúcsok) valamelyikébe vezető utat, gyakran egy optimális (költségű) ilyen utat. A problématér modellezéséhez használt ún. ***reprezentációs gráf*** egy véges kifokú, élsúlyozott irányított gráf, ahol az élsúlyok közös pozitív alsó korláttal rendelkeznek [3] [24]. Ezt hívja Nilsson δ-gráfnak [24]. Az említett korlátozások azt ellensúlyozzák, hogy nagyon nagy, akár végtelen nagy méretű gráfokban is tudjunk sikerrel optimális megoldási utat találni. Vannak olyan feladatok is, amelyek problématerének modellezéséhez egy ún. ÉS/VAGY gráf kell, amelyben irányított élek helyett irányított hiper-élek vezetnek egy csúcsból egyszerre több (véges számú) csúcsba. Egy ÉS/VAGY gráffal modellezett feladat megoldása azonban visszavezethető egy közönséges irányított gráffal modellezett feladat megoldására [10].

Többféle útkereső algoritmust ismerünk. Didaktikai szempontból ezeket az ún. kereső rendszer sémából lehet származtatni, és elsősorban az alkalmazott vezérlési stratégiájuk alapján különböztethetjük meg őket [24] [25]. Ezek között a ***heurisztikus gráfkereső algoritmusok*** olyan nem-módosítható vezérlési stratégiát használnak, amelyben elsődleges szerep jut a megoldandó feladattól származó ismeretnek, a heurisztikának.

Egy gráfkereső algoritmus a reprezentációs gráf startcsúcsából kiinduló utakat fokozatosan és szimultán módon deríti fel, és ezt az egyre bővülő részgráfot (***kereső gráf***) eltárolja. Minden lépésben azon már felfedezett csúcsok (***nyílt csúcs*ok**) közül, amelyeknek a gyermekeit még nem (vagy nem eléggé) ismeri, kiválaszt egyet, hogy a gyerekeihez vezető éleket megvizsgálja. A kiválasztás egy f:*NYÍLT*$\longrightarrow\mathbb{R}$ ***kiértékelő függvény*** segítségével történik (*NYÍLT* az adott pillanatban érvényes nyílt csúcsok halmazát jelöli), és mindig a legkisebb értékű nyílt csúcs kiválasztására kerül sor. Ha az f értékének kiszámolásához felhasználunk olyan h:$N\longrightarrow\mathbb{R}$ ismeretet, ún. ***heurisztikus függvényt*** (N a reprezentációs gráf csúcsainak halmazát jelöli), amely a modellezett feladat ismereteire támaszkodva adja meg egy adott csúcsra az abból valamelyik célcsúcsba vezető optimális út költségének becslését, akkor ***heurisztikus gráfkeresésről*** beszélünk.

A feladatok útkeresési problémaként történő modellezésével, az útkereső algoritmusok vizsgálatával és rendszerezésével, az általános gráfkereső algoritmus tulajdonságaival, a nevezetes heurisztikus gráfkereső algoritmusokkal több publikációmban is foglalkoztam. A Fekete Istvánnal és Nagy Sárával közösen készített Bevezetés a mesterséges intelligenciába c. könyv első kiadása, amelynek a keresésekkel kapcsolatos részét írtam [26], még a doktori fokozatszerzésem előtt jelent meg, de azóta két újabb kiadást is megélt. Öt évvel a fokozatszerzésem után jelent meg hazai mesterséges intelligencia kutatókkal közösen készített Mesterséges intelligencia c. könyvünk [1], amelynek több fejezetét én készítettem (1, 2, 3, 4.1, 4.2 fejezetek). A heurisztikus gráfkereső algoritmusok összefoglalása olvasható [2]-ben is, de több online oktatási anyagot is megjelentettem ebben a témában [3] [4].

A továbbiakban a heurisztikus gráfkereső algoritmusokkal kapcsolatos doktori fokozatom megszerzése óta született kutatási eredményeimet mutatom be négy tézis köré csoportosítva.

2.1. Mohó *A* algoritmus

Az *A algoritmus* olyan heurisztikus gráfkeresés, amelynek *f* kiértékelő függvénye *g*+*h* alakú, ahol *g* az algoritmus által számolt ún. költségfüggvény (*g*(*n*) a startcsúcsból az *n* csúcsba vezető út egyik már felfedezett út költségét mutatja), *h* pedig egy nem negatív heurisztikus függvény (*h*(*n*) a hátralevő optimális megoldási út költségét becsüli). Az *A algoritmus* minden olyan feladatra, amelynek van megoldása, talál megoldást [24]. Ismert eredmény az is, hogy ha a *h* ***megengedhető*** is (*h*(*n*) minden *n* csúcsra alulról becsüli a hátralevő optimális megoldási út költségét), akkor az algoritmus garantálja az optimális megoldás megtalálását, amennyiben létezik megoldás [27]. (Ilyenkor az *A algoritmust A* algoritmusnak* szokás nevezni.) Ennek egyik fontos feltétele az, hogy az algoritmus csak akkor terminálhat egy célcsúcs megtalálásával, amikor a célcsúcsot éppen ki akarja terjeszteni, noha lehet, hogy ez a célcsúcs már jó ideje ott van a memóriájában a nyílt csúcsok között [28].

Egy fiatal kutatók számára kiírt és elnyert általam vezetett OTKA (F4188) pályázat keretében vizsgáltuk az *A algoritmus* működését, és kipróbáltuk egy olyan változatát is, amelyik mohó módon azonnal terminál, ha célcsúcsot talál, és nem vár addig, amíg sor kerül ennek a célcsúcsnak a kiterjesztésére. Ez a mohó változat minden olyan feladatra talál megoldást, amelyikre az eredeti változat is megoldást ad ([6] 3.4. és 4.1.3. tétel). Nyilvánvaló az is, hogy a mohó változat futási ideje lényegesen rövidebb. Elveszíti viszont azt a képességet, hogy megengedhető heurisztika esetén optimális megoldással terminál̈jon, ha van megoldás. Felfigyeltünk ugyanakkor arra, hogy bizonyos feladatoknál (például a *15*-ös kirakó játék esetében) a mohó változat mégis mindig optimális megoldást ad, amennyiben létezik megoldás.

Mohó A algoritmusnak tehát azt a gráfkereső algoritmust nevezzük, amelyik kiértékelő függvénye *g*+*h* alakú, ahol *g* az algoritmus által számolt költségfüggvény, *h* pedig a nem negatív heurisztikus függvény, az algoritmus pedig sikeresen terminál, ha egy csúcs kiterjesztésekor annak valamelyik gyereke célcsúcs. ***Mohó A* algoritmusról*** akkor beszélünk, ha a heurisztikus függvény megengedhető.

2.1. Tézis: Az A* algoritmus mohó (és emiatt hatékonyabb) változata talál adott korlátnál nem rosszabb megoldást – amennyiben van egy feladatnak megoldása –, sőt, bizonyos feltételek esetén garantálja az optimális megoldás megtalálását is.

Egy *mohó A* algoritmus h* heurisztikus függvénye megengedhető, azaz minden *n* csúcsra az abból célba vezető legolcsóbb út költségét alulról becsüli. Speciálisan, minden *t* célcsúcsot közvetlenül megelőző *n* csúcsra a *h*(*n*) érték kisebb vagy egyenlő a célcsúcsba vezető él súlyánál (költségénél) (azaz $h(n) \leq c(n,t)$). Megmutatható, hogy ha megadható olyan nem-negatív *C* valós szám, hogy arra minden *t* célcsúcs és minden (*n*,*t*) él mellett teljesül a $h(n)+C \geq c(n,t)$ kritérium, akkor a *mohó A* algoritmus* által talált megoldás költsége legfeljebb *C*-vel lehet rosszabb (nagyobb) az optimális megoldási költségnél ([6] 4.2.3. tétel).

Ennek következménye, hogy ha a fenti *C* speciálisan 0, azaz minden *t* célcsúcsra és minden (*n*,*t*) élre teljesül a $h(n) = c(n,t)$ kritérium, akkor a *mohó A* algoritmus* biztosan optimális megoldási utat talál, ha van megoldás ([6] 5.1.2. tétel). Mivel a *h* heurisztikus függvény feladata, hogy bármelyik csúcsban alulról becsülje az abból célba vezető optimális út költségét, a fenti kritérium lényegében azt fogalmazza meg, hogy a *h* heurisztikus függvénynek a célcsúcsok szülőcsúcsainál a pontos hátralevő költséget kell mutatnia. Ez valósul meg például a *15*-ös kirakó játékon, ha a nevezetes *W* (hány lapocska nincs még a helyén), illetve *P* (mennyi a nem helyén levő lapocskák célbeli helyükre történő mozgatások lépésszámainak az összege) heurisztikák mellett a *mohó A* algoritmus* is minden esetben optimális megoldást talált, hiszen ezek a heurisztikák 1 lépéssel a cél előtt éppen az 1 értéket mutatják.

Nevezetes *A* algoritmus* az, amelyik *h* heurisztikus függvénye monoton megszorításos (következetes) is, azaz minden (*n*,*m*) élre fenn áll, hogy $h(n)-h(m) \leq c(n,m)$. Ilyenkor az *A* algoritmust* A^c *algoritmusnak* nevezzük. (A már említett heurisztikák a 15-ös játékra ilyenek.) Erről az algoritmusról Nilsson megmutatta, hogy egy csúcsot legfeljebb egyszer terjeszt ki, és a kiterejsztéskor már ismeri az algoritmus a startcsúcsból ezen csúcsba vezető optimális költségű utat [24]. Ennek tükrében természetesen beszélhetünk *mohó* A^c *algoritmusról* is. ([6] 4.3.1. definíció és 4.3.3. tétel). Belátható, hogy amennyiben van olyan *C* nem-negatív valós szám, amelyre minden *t* célcsúcs és minden (*n*,*t*) él mellett teljesül a $h(n)+C = c(n,t)$ kritérium, akkor a *mohó* A^c a*lgoritmus* által talált megoldás garantáltan optimális ([6] 5.2.2. tétel). Azt is sikerült megmutatni, hogy a monoton megszorítás és a $h(n)+C = c(n,t)$ kritérium együtt nemcsak elégséges, de szükséges feltételei is az optimális megoldás megtalálásának.

Összefoglalás:

A 2.1. tézist az alábbi eredmények támasztják alá:

1. Az általános gráfkereső algoritmus, az *A algoritmus*, az *A* algoritmus* és az A^c *algoritmus* működési eredményére vonatkozó tulajdonságai a mohó változatokra is érvényesek, kivéve, hogy az *A* algoritmus* és az A^c *algoritmus* mohó változatai általában nem garantálják az optimális megoldás megtalálását.
2. Becslés adható a célcsúcsokba vezető élek súlyai alapján arra, hogy a *mohó A* algoritmus* által talált megoldási út költsége mennyivel lesz nagyobb az optimális költségnél.
3. Létezik elégséges feltétel arra, hogy a *mohó A* algoritmus*, illetve a *mohó* A^c *algoritmus* garantáltan optimális megoldást találjon.
4. Szükséges és elégséges kritérium adható a heurisztikus függvényre ahhoz, hogy a mohó *A** algoritmus garantáltan optimális megoldást találjon.

A fenti eredményeket Ásványi Tiborral közösen publikáltam 1998-ban az „A More Effective Version of algorithm A” (Annales Univ. Sci. Budapest., Sect. Comp. 17 (1998) 33-48.) közleményben [6]. A tézis igazolásához bevezetett kritériumokra, és azok következményeire Ásványi Tibor figyelt fel, ezek tételszerű megfogalmazása és bizonyítása a saját eredményem.

2.2. Egy best-first gráfkeresés

A best-first gráfkereső algoritmus szigorúbb értelmezés szerint az, amelyik *f* kiértékelő függvénye azonos a hátralevő optimális költséget becslő *h* heurisztikus függvénnyel: egy ígéretes nyílt csúcs kiválasztásánál az algoritmus csak arra figyel, hogy várhatóan milyen költséggel lehet majd onnan eljutni a célba. Ez az előre tekintő stratégia nem veszi számításba, hogy eddig mennyit jöttünk a startcsúcsból. Ez egy olyan mohó stratégia, amelyik nem garantálja az optimális megoldási út megtalálását, sőt végtelen gráfokban a megoldás megtalálását sem.

2010-ben vezettem egy kutatás-fejlesztési projektet, amely a Telekom számára tervezett egy postai címeket feldolgozó alkalmazást, amelynek elkészítettük egy prototípusát is. A feladat lényege az volt, hogy szöveges formában kapott, gyakran pontatlanul és hiányosan megadott postai címeket kellett hivatalos (szabványos és valóságos) címmé alakítani, azaz megtisztítani. Feladatunk volt az is, hogy ha az inputot többféleképpen is lehetett értelmezni, akkor rangsoroljuk a szóba jövő változatokat hihetőségük szerint.

2.2. Tézis: Hatékony szoftvert lehet készíteni egy módosított best-first gráfkereső algoritmusra támaszkodva nagy mennyiségű pontatlan és hiányos postai cím értelmezésére, tisztítására.

A megoldáshoz többféle mesterséges intelligencia technológiát is alkalmaztunk. Egy szabályalapú rendszert terveztünk [7], ahol egy szabály az inputként megadott cím valamelyik részletét azonosította. Például ilyen szabály volt az, amelyik a cím egy különálló szavát megpróbálta településnévként értelmezni, és megkereste a hivatalos településjegyzékben a szóhoz legközelebb álló településnevet a Levenshtein távolság segítségével. Minél kevésbé illeszkedett a vizsgált szóhoz illesztett településnév, annál kevésbé volt hihető az adott szabály által előállított címváltozat. Az egymás után alkalmazott szabályok vezettek el a kezdeti nyers szövegtől a kitisztított címig, amelynek a hihetősége az egyes szabályalkalmazások hihetősége határozta meg.

A közbenső (félig megtisztított) címekhez, és természetesen a végső hivatalos címekhez is egy hihetőségi mértéket (0 és 1 közötti számot) rendeltünk, amely alapján aztán a címeket (a félig tisztítottakat is és a véglegeseket is) rangsorolni lehetett. Amikor egy szabályt alkalmaztunk egy félig megtisztított címre, akkor a lépés eredményeként csökkent a tovább tisztított cím hihetősége, attól függően, hogy maga a szabály mennyire volt releváns (hasznos), illetve a szabály alkalmazásával kapott eredmény mennyire volt hihető. Formailag a szabály alkalmazása előtti cím hihetőségét megszoroztuk a szabály hasznosságával (0 és 1 közötti szám), valamint a szabály alkalmazhatóságának hihetőségével (0 és 1 közötti szám). Ez egy tipikus heurisztikus bizonytalanság kezelési technika, amelyet a szabályalapú következtetéseknél lehet alkalmazni [1].

A szabályok hasznossági mértékére (melyik szabály mennyire fontos, mennyire gyakran eredményes, azaz mennyire hasznos) mi adtunk kezdeti becslést, majd megerősítéses tanulással finomítottuk annak értékét. A szabályok alkalmazhatóságának hihetőségét változatos módon számoltuk, de például a településnevek, illetve a közterületnevek azonosításánál erre 0 és 1 közé normált Levenshtein távolságot használtuk.

Mivel egy félig megtisztított címre egyszerre több szabályt is lehet alkalmazni, ráadásul többféleképpen, nagyon sok potenciális megoldás (kitisztított cím) adódik egy inputra. A problémateret egy olyan irányított fával lehet modellezni, amelynek gyökere a kezdeti szöveges formájú tisztítatlan cím (input), ágai pedig egy-egy szabály-sorozat alkalmazását szimbolizálják. Egy

ágon a levelek felé haladva egyre tisztább címeket képviselő csúcsokat találunk. A fa levelei a teljesen megtisztított cím-változatok. Egy ág költsége nem egy szokásos útköltség, hanem az ág végén levő levél hihetőségi mértéke. (A fa gyökerének mértéke: 1.) Minden él (szabály-alkalmazás) a szülő csúcs hihetőségét csökkentheti, de biztosan nem növeli. A feladat tehát egy olyan útkeresési probléma, ahol a cél annak az ágnak (vagy ágaknak) a felderítése, amelyek levélcsúcsának a legnagyobb a hihetősége.

Ennek a fának a mérete a viszonylag kevés szabály ellenére is hatalmas. Annak érdekében, hogy elkerüljük a kombinatorikus robbanást az alábbi két ötlet megvalósítása bizonyult jelentősnek.

Egyrészt csoportosítottuk a szabályokat (település nevet felismerő szabályok, irányítószámot felismerő szabályok stb.), és rögzítettük, hogy melyik csoporthoz tartozó szabályt hányadik lépésben lehet alkalmazni. Ezzel a trükkel karcsúbbra szabtuk a problémateret reprezentáló fát, ám az még mindig elég nagy volt ahhoz, hogy egy „brute-force" maximum kiválasztással megkereshessük a leghihetőbb célcsúcsot.

A legjobb megoldás megkereséséhez egy best-first gráfkeresést alkalmaztunk három módosítással. Egyrészt a csúcsokhoz rendelt *g* értékeket itt nem a megszokott módon kellett számolni. Egy csúcs *g* értéke a csúcshoz tartozó félig tisztított cím hihetőségi mértéke, amelyet a szülőcsúcs hihetőségi mértékének, az alkalmazott szabály hasznosságának és a szabályalkalmazás hasznosságának szorzataként állítunk elő. Másrészt nem az optimális költségű megoldási utat keressük, hanem az optimális hihetőségű levélcsúcsot. A keresés mohó módon mindig a legnagyobb hihetőségű nyílt csúcsot választja (itt érvényesül a best-first stratégia), és ennek kiterjesztése a kiválasztott csúcs által képviselt címet tisztítja tovább az összes lehetséges módon. Harmadrészt, amikor találunk egy célcsúcsot (levélcsúcsot), akkor az algoritmus nem áll le: tovább keresi a célcsúcsokat, és ha jobbat talál, akkor azt tárolja majd el. (Lényegében egy fajta maximum keresésbe megy át.) A hatékonyság érdekében azonban töröljük a *NYÍLT* halmazból az összes olyan csúcsot, amelyiknek hihetőségi mértéke rosszabb az eddig talált legjobb célcsúcs hihetőségénél. Ezzel biztosan nem veszítünk el az eddig talált megoldásnál jobbakat, hiszen a problémateret modellező fa minden ágában a hihetőségi mértékek lefele haladva nem csökkennek. Könnyű látni, hogy ez a módszer ebben a speciális problématérben garantálja az optimális hihetőségű célcsúcs megtalálását. Egyrészt talál megoldást, mert véges gráfokban (ilyen a véges fánk is) minden gráfkeresés talál megoldást, ha van megoldás. Másrészt optimális megoldást talál, hiszen egy maximum keresést alkalmazunk. A módszer nem vezetett kombinatorikus robbanáshoz a szabályfajták meghatározott sorrendben történő alkalmazásának és a vágásoknak köszönhetően. Ez utóbbit futtatási eredmények igazolták [7].

Összefoglalás:

A 2.2. tézist megvalósító szoftverben alkalmazott ötletek közül az alábbiak származnak tőlem:

1. Az alkalmazott szabályok csoportosítása és ezen csoportoknak a problémateret adó fa szintjeihez történő rendelése.
2. A hihetőségi mértékek számításának kalkulusa.
3. Módosított best-first algoritmus bevezetése és annak vizsgálata.

Az eredményeket először egy belső használatra készített dokumentációban írtuk le, majd folyóiratban is publikáltuk: Giachetta, R., Gregorics, T., Istenes, Z., Sike, S. „Address Standardization" Annales Univ. Sci.Budapest., Sect. Comp. 37 (2012) 5-18.

2.3. *A** algoritmus*

Egy gráfkereső algoritmus minden általa felfedezett *k* csúcsnál két értéket tart nyilván: $\pi(k)$-t, a *k* csúcs valamelyik már felfedezett szülőjét, és $g(k)$-t, az *s* startcsúcsból *k* csúcsba vezető egyik már felfedezett útnak a költségét. Ennek következményeként a $\pi:N\longrightarrow N$ ***visszamutató pointerek*** minden már felfedezett *n* csúcshoz egyértelműen kijelölnek egy már felfedezett startcsúcsból *n* csúcsba vezető $s\longrightarrow^{\pi} n$ utat. Megjegyezzük azonban, hogy $g(n)$ nem feltétlenül ennek az útnak a költségét mutatja.

Az *A** *algoritmus* egy nevezetes heurisztikus gráfkeresés, amelyet Detcher és Pearl vezetett be azon vizsgálataik során, amikor az *A** *algoritmus* memória bonyolultságát elemezték. Az *A** *algoritmus* kiértékelő függvénye egy *n* nyílt csúcs megítélésénél nemcsak annak $g(n)$ költségét és $h(n)$ heurisztikus értékét veszi figyelembe, mint az *A** *algoritmus*, hanem az algoritmus által nyilvántartott startcsúcsból az *n* csúcsba vezető út minden *k* csúcsának $g(k)$ költségét és $h(k)$ heurisztikus értékét is. Az *A** *algoritmus* kiértékelő függvényének számítási képlete $f(n) = \max_{k\in s\longrightarrow\pi n} g(k)+h(k)$, azzal a kiegészítéssel, hogy ha a *NYÍLT* halmaznak több azonosan legkisebb kiértékelő függvényértékű csúcsa is van, és ezek között van célcsúcs, akkor a célcsúcs kiválasztása élvezzen előnyt. Az *A** *algoritmus* néhány jelentős tulajdonságát Detcher és Pearl bizonyította be [29].

Az *A** *algoritmus* kiértékelő függvényének kiszámítása időigényes. Jelentős időt lehet megspórolni azzal, ha a nyílt csúcsok kiértékelő függvény értékét nem a kiválasztás során számoljuk ki újra és újra, hanem akkor, amikor egy kiterjesztés következtében a *NYÍLT* halmazba kerülnek, vagy ha már ott voltak, de egy eddiginél olcsóbb utat találunk hozzájuk. Az algoritmus egy *n* csúcs kiterjesztésekor úgyis megvizsgálja az *n* csúcs összes gyerekét, és ekkor egy először felfedezett, vagy egy olcsóbb úton újra felfedezett *m* gyerekének kiértékelő függvényértéket az $f(m) = \max(g(m)+h(m), f(n))$ képlettel is számolhatja. Kezdetben $f(start)=h(start)$. Ezt a gráfkeresést Russel és Norviq javasolta [30], de nem az *A** *algoritmus* másik definíciójaként, hanem az *A** *algoritmusnak* a javításaként abból a célból, hogy az esetleges nem monoton tulajdonságú heurisztikus függvények alkalmazásából adódó hátrányt kiküszöböljék [30]. Ugyanennek a problémának a megoldására alkotta Martelli a *B algoritmust* [31]. Én arra mutattam rá, hogy Russel és Norviq javaslata valójában az *A** *algoritmusnak* egy alternatív és hatékonyabb definíciója.

2.3. Tézis: Az A** algoritmusnak egy alternatív és hatékonyabb változata a Russel és Norviq által bevezetett gráfkeresés. A két változat azonos, így eredményük, memória igényük, és lépésszámuk is megegyezik. Az A** algoritmusnak a lépésszáma kevesebb, mint a nevezetes B algoritmusé.

A Russel és Norviq által bevezetett gráfkeresés és a Detcher és Pearl által definiált *A** *algoritmus* azonosságát az bizonyítja, hogy (bármelyik feladat megoldása során) ezek a gráfkeresések minden lépésben ugyanazokat a nyílt csúcsokat ugyanazon kiértékelő függvényértékkel tartják nyilván (feltételezve, hogy nemdeterminisztikus működésüket ugyanazon „tie-breaking-rule” segítségével számoljuk fel) ([8] 1. tétel). Ennek az állításnak a bizonyításából kiderül, hogy az *A** *algoritmus* e két változatának működése között mindössze annyi a különbség, hogy lehetnek a reprezentációs gráfnak olyan már felfedezett, de nem nyílt csúcsai, amelyeknél a két verzió eltérő kiértékelő függvényértéket számol. A kiértékelő függvénynek azonban csak a nyílt csúcsok közül történő kiválasztásnál van szerepe. A bizonyításnak talán az a legérdekesebb része, amely megmutatja, hogy még azon nyílt csúcs kiértékelő függvényértékei is megegyeznek a két változatnál, amelyhez úgy talált az algoritmus olcsóbb utat, hogy nem közvetlenül a szülő csúcsát terjeszti ki.

Az állításból közvetlenül adódik, hogy mindkét változat minden lépésben ugyanazt a csúcsot választja kiterjesztésre. Ennek egyik következménye, hogy a két változat ugyanakkor és ugyanolyan költségű megoldási út megtalálásával terminál (feltéve, hogy van a feladatnak megoldása), és mivel az eredeti

változatról bebizonyítható ([29], valamint [8] 2. és 3. lemma), hogy garantáltan optimális megoldást talál, ha van megoldás, ezért ez az alternatív változatra is fenn áll. A másik következmény a memória igények azonossága, azaz a zárt (kiterjesztett) csúcsok számának – sőt a zárt csúcsok halmazának – azonossága, de nemcsak termináláskor, hanem minden lépésben, ugyanis a két változat által a nyilvántartott kereső gráf azonos, ha minden lépésében ugyanazt a csúcsot terjesztik ki. (Említettük, hogy eltérés csak a kereső gráf nem nyílt csúcsainál feltüntetett kiértékelő függvényértékben lehet.) A harmadik következmény a kiterjesztések, azaz a lépések számának azonossága, amely a futási idő fontos komponense.

Érdekes tulajdonsága továbbá az *A** algoritmusnak*, hogy az egymás után kiterjesztett csúcsok kiértékelő függvényértékei monoton növekedő sorozatot alkotnak ([8] 4. tétel), akárcsak a monoton heurisztikájú *A* algoritmusnak*.

A memória bonyolultság vizsgálatainál nemcsak különböző gráfkereső algoritmusokat szokták összevetni azonos feladat osztályokon, hanem egy adott gráfkeresést is vizsgálhatunk ugyanazon feladat különböző heurisztikái mellett. Megmutattam, hogy a különböző heurisztikát használó *A** algoritmusok* esetén ugyanúgy használható a jobban informáltság fogalma, mint azt Nilsson az *A* algoritmusokra* bevezette [24], és ennek a memória igényre gyakorolt hatása is ugyanaz ([8] 5. tétel).

Detcher és Pearl vette észre, hogy az *A** algoritmus* bizonyos szempontból uralja az összes megengedhető heurisztikus gráfkereső algoritmust a megengedhető heurisztikájú feladatokon. Az ezt megfogalmazó állítás úgy szólt, hogy nincs olyan megengedhető heurisztikájú feladat, ahol az *A** algoritmus* olyan csúcsot terjesztene ki, amelyet egy hasonlóan optimálist megoldást találó gráfkereső algoritmus nem terjeszt ki ([29] 10. tétel). Sajnálatos módon ennek bizonyítása tartalmaz egy kis hibát: a szerzők konstruálnak egy megengedhető problémát, de annak a reprezentációs gráfja nem δ-gráf, mert az egyik éle lehet nulla költségű is. Nekem sikerült ezt a hibát kijavítanom, és Detcher és Pearl állítására bizonyítást találnom. Ráadásul igyekeztem a bizonyításuk szerkezetét, gondolatmenetét is megőrizni ([8] 6. tétel).

Végezetül az *A** és az *A** algoritmusok* futási idejének összehasonlítását is elvégeztem, és kimutattam, hogy hogy az *A** algoritmus* gyorsabb nemcsak az *A* algoritmusnál*, hanem annak Martelli féle változatánál [31], az ún. *B algoritmusnál* is ([8] 7. tétel és következményei).

Összefoglalás:

A 2.3.tézis állításaival kapcsolatban [8] -ban az alábbi eredményeket mutattam be:

1. Rámutattam, hogy Detcher és Pearl *A** algoritmusának* alternatív definíciója a Russell és Norviq által bevezetett gráfkereső algoritmus egy elhanyagolható tünetet leszámítva azonos.
2. Beláttam, hogy az *A** algoritmus* egymás után kiterjesztett csúcsainak kiértékelő függvényértékei monoton növekedő sorozatot alkotnak.
3. Kiterjesztettem a jobban informáltság fogalmát az *A** algoritmusokra*.
4. Észrevettem egy hibát Detcher és Pearl az *A** algoritmus* memória bonyolultságával kapcsolatos egyik állításának bizonyításában, és egy másik bizonyítást adtam rá.
5. Összevetettem az *A** algoritmus* futási idejét az *A* algoritmusnak*, és annak hatékonyabb változatának, a *B algoritmusnak* futási idejével, és kimutattam, hogy az *A** algoritmusé* jobb.

2.4. UML diagrammok kirajzolása

2013-tól veszek részt az ELTE IK-n a végrehajtható UML modellezéssel foglalkozó kutatás-fejlesztési labor munkájában. Ennek keretében készítettünk egy szöveges alapú végrehajtható UML eszközt, amelynek egyik modulja a szöveges leírással megadott osztály- és állapotgép diagrammokat jeleníti meg grafikusan. Az osztályokat, illetve az állapotokat dobozokkal (szöveget is tartalmazó téglalapokkal), az asszociációkat, a származtatást, illetve az állapot-átmeneteket a dobozok közé rajzolt tört vonallal lehet ábrázolni.

A kirajzolási probléma megoldása két fő lépésből áll: először el kell helyezni a diagramm dobozait a rajzoló felületen, és ezek közül némelyeket össze kell kötni tört vonalakkal.

A kirajzolást nem terveztük automatikusra: lehetőséget adtunk arra, hogy feltételeket lehessen megfogalmazni az elhelyezéshez. Ilyen például az, hogy egy doboz legyen egy másik felett, de vízszintes irányban tetszőlegesen elcsúsztatva, vagy pontosan felette, vagy négy doboz alkosson gyémánt alakzatot, vagy egy összekötő vonal egy doboz felső részéről induljon ki stb.)

2.4. Tézis: Útkereső algoritmusok alkalmasak egy dobozokból és azokat összekötő vonalakból álló rajz elkészítésére részleges korlátozások betartása mellett.

Az elrendezést leíró nyelv a dobozokat pontszerű alakzatoknak tekinti, és ezek koordinátáira állít fel korlátokat. Például, ha az *A* doboznak a *B* doboz felett kell lenni *(north*(*A*,*B*)), akkor ez a két doboz *y* koordinátáira írja elő az $A.y - B.y \geqq 1$ feltételt. Az *above*(*A*,*B*) (legyen A egy egységgel a B felett) két egyenlőtlenséget is generál: $A.y - B.y = 1$ és $A.x - B.x = 0$. Látható, hogy az elrendezés leírása egy speciális lineáris programozási problémát ($A \cdot x \leqq b$) határoz meg, amely mátrixának sorai két darab egy abszolút értékű elemen kívül csupa nulla értéket tartalmaznak. Ezeket az egyenlőtlenség rendszereket különbségi korlátrendszereknek hívják, és hatékonyan megoldhatók Bellman-Ford algoritmusával [32].

A dobozok koordinátáinak meghatározása után elhelyezzük a dobozokat egy rácsszerkezeten úgy, hogy ekkor a dobozoknak már kiterjedése is legyen. Ez a dobozba írandó szöveg méretétől és a dobozhoz kapcsolódó vonalak számától függ. Egy kezdetben pontszerű doboz úgy kap kiterjedést, hogy a doboz pontján keresztül menő egy függőleges és egy vízszintes vonal helyébe több, egymástól egységtávolságra eső, függőleges és vízszintes vonalakat rajzolunk. Így egy pont az új rácsvonalakra illeszkedő téglalap lesz.

Az összekötő vonalakat a dobozok közé (reflexív kapcsolat esetén egy doboz köré) egymás után rajzoljuk be. Két eltérő doboz összekötése esetén megkülönböztetjük a kezdő dobozt és a céldobozt. A kezdő doboz szélétől indulva kell a céldoboz széléhez húzni egy olyan törtvonalat, amely a rácsszerkezet rácsvonalaira illeszkedik, és kikerüli a többi dobozt. Megállapodás szerint két összekötő vonal nem osztozhat egy rácsvonal ugyanazon szakaszán, sőt egy rácsponton is csak akkor, ha ott egymásra merőlegesen, kanyarodás nélkül haladnak át, azaz keresztezik egymást. Egy összekötő vonal fontos tulajdonsága a hossza (hány egység hosszú), a töréspontjainak (ahol a vonal elkanyarodik) száma, valamint más összekötő vonalakkal történő kereszteződéseinek száma. Ezen tényezők súlyozott összegét tekintjük a *p* vonal költségének: $w(p) = c_1 \cdot length(p) + c_2 \cdot turns(p) + c_3 \cdot crossings(p)$. Célunk a legolcsóbb ilyen vonal megtalálása.

Ahhoz, hogy egy optimális megoldást garantáló heurisztikus gráfkereséssel (*A* algoritmussal*) oldjuk meg a feladatot, azt útkeresési problémaként kell megfogalmaznunk. A feladat elsődleges modellje felfogható ugyan egy olyan irányított gráfnak, amelyben a rácspontok a csúcsok, a rácspontokat összekötő rácsvonal-szakaszok az oda-vissza irányuló élek, de hamar be lehet látni, hogy egy összekötő

vonalat szimbolizáló irányított út költsége nem az általa érintett irányított élek költségének összege. Ezért ez az elsődleges modell nem illeszkedik az *A* algoritmus* által használt gráfmodellekhez.

Elkészíthető viszont a feladat duális modellje. Ebben a csúcsok a szabad (a dobozok között található) egységnyi rácsvonal-szakaszok, kiegészítve a keresés startcsúcsával, amely a kezdő doboz középpontját szimbolizálja (ennek nem kell rácspontnak lenni), valamint célcsúcsokkal, amelyek a céldoboz oldalaira (de nem a sarkaira) merőlegesen illeszkedő, a céldoboztól kifelé induló rácsvonal-szakaszok. Ennek a duális gráfnak élei az egy közös rácspontban egymáshoz kapcsolódó rácsvonal-szakaszokat kötik össze oda-vissza irányított módon. Egy szabad (dobozok közötti) rácspont 12 ilyen élt generál, mert ennyiféleképpen lehet áthaladni rajta a rácsponthoz kapcsolódó egyik szakaszról egy másikra. Ha egy ilyen él azonos állású rácsvonal-szakaszokat köt össze (azaz nincs a rácspontban kanyar), akkor az él költsége legyen c_1, amennyiben elkanyarodik, akkor $c_1 + c_2$, ha egy korábban már berajzolt törtvonalat keresztez, akkor a költség legyen $c_1 + c_3$. Szükség van még ezeken kívül a duális gráf startcsúcsából kiinduló élekre is, amelyek a kezdő doboz közepéből a kezdő doboz oldalaihoz merőlegesen kapcsolódó rácsvonalakhoz vezetnek. Ezeknek a költsége legyen $2 \cdot c_1$. Be lehet látni, hogy az elsődleges gráf egy összekötő vonalának költsége azonos a duális gráfban neki megfeleltetett megoldási út költségével, azaz a duális gráfban egy megengedhető heurisztikus függvénnyel az *A* algoritmus* optimális megoldási utat fog találni. Ilyen heurisztikus függvény lehet a következő. Becsüljük alulról egyrészt a hátralevő lépések minimális számát, a Manhattan távolságnak is nevezett Hamilton távolságot, másrészt a cél eléréséhez szükséges minimális törések (fordulók) számát: $h(n) = c_1 \cdot Manhattan(n) + c_2 \cdot minimalturns(n)$ a duális modell egy tetszőleges n csúcsára. Belátható, hogy a duális modellben ez a célba vezető hátra levő optimális költséget alulról becslő heurisztika lesz.

Összefoglalás:

A 2.4. tézist alátámasztó állítások:

1. A diagramok dobozainak elhelyezését leíró speciális egyenlőtlenség rendszer Bellman-Ford algoritmusával oldható meg.
2. A dobozok közötti összekötő vonalak megtalálására alkalmas egy megfelelő duális modellben megvalósított *A* algoritmus*.

Az alfejezetben ismertetett módszer egy konferencián előadott és annak kiadványában megjelent: Gregorics, B., Gregorics, T., Kovács, G. F., Dobreff, A., Dévai, G.: Textual Diagram Layout Language and Visualization Algorithm [9]. A végrehajtható UML modellek annotált Java nyelven történő szöveges leírásának ötlete és Java programként való végrehajtása Dévai Gergelytől származik. A Java annotációk kidolgozásának oroszlán részét Kovács Gábor végezte. A részben felügyelt elrendezés leíró nyelvének megalkotását, az elrendezés algoritmusainak implementálását, az egész folyamatnak a txtuml eszközbe integrálását Gregorics Balázs végezte, a kiszámolt elrendezésnek a PAPYRUS segítségével történő grafikus megjelenítése Dobreff András munkája volt. A diagramok dobozainak elhelyezését leíró speciális egyenlőtlenség rendszer Bellman-Ford algoritmussal történő megoldásának ötletet én vetettem fel a [32]-ben olvasott különbségi korlátrendszerekkel kapcsolatos alfejezet alapján. A dobozok közötti összekötő vonalak megtalálására én javasoltam az *A* algoritmust*, definiáltam az algoritmus modelljét adó ún. duális modellt, és bebizonyítottam, hogy az elrendezéshez talált heurisztika megengedhető, azaz az algoritmus által talált megoldás garantáltan optimális.

3. Egységes programozási modell

Az ELTE informatika oktatásában kezdettől fogva jelen volt a programozás elméleti hátterének bemutatása, 1982-től pedig ennek egy sajátos, relációs modellre épülő megközelítése. Ennek gyökerei Hoare [11], Mills [33], Dijkstra [12] és Gries [13] munkáiból erednek, egységbe foglalásának alapgondolata, kidolgozásának sok ötlete Fóthi Ákosé [14], ugyanakkor számos részletet mások találtak ki. Hadd említsem meg ezek közül Kozics Sándort, aki akárcsak Fóthi Ákos, tanárom volt. Ma már igen nehéz azt pontosan meghatározni, hogy egy-egy részlet kitől származik: egyrészt azért, mert bizonyos ötletek megszületése és kidolgozása nem mindig ugyanahhoz a személyhez fűződtek, másrészt pedig a szakfolyóiratokban nem közölt elemek szerzőinek neve sehol nincs dokumentálva. A legteljesebb névsort azokról, akik ennek a csapatmunkának részvevői voltak, valószínűleg a [34] publikációban találjuk.

A relációs programozási modell lényege az, hogy mind a megoldandó feladatot, mind a megoldó programot közös állapottéren felírt relációval ábrázoljuk, és a megoldás tényét, ezen relációk összehasonlításával állapítjuk meg. Az így megadott alapdefiníciókhoz illeszthetjük a programok helyességbizonyításának megszokott eszköztárát (feladat specifikációja, programkonstrukciók, helyességi tételek), és ezáltal a modell beágyazódik a formális verifikáció nemzetközileg is elfogadott szemléletébe.

A továbbiakban azokat az eredményeimet mutatom be, amelyekkel módosítottam vagy kiegészítettem ezt a relációs modellt. Ezek bevezetése minden esetben egy szembeállítást igényel, amelyben elmondom a bírálatomat a modell egy-egy eleméről és alternatívát fogalmazok meg arra. Mindezzel együtt úgy gondolom, hogy nem a relációs modell életképességét kérdőjelezem meg, hanem ellenkezőleg, erősítem azt.

3.1. Program fogalma

A programozás elmélet relációs modelljében a program is egy reláció, amely azt mutatja meg, hogy egy adott állapotból indulva milyen végrehajtások következhetnek be annak működése során. Korábban a program végrehajtásait olyan állapotsorozatokkal adtuk meg, amelyek minden állapota ugyanannak az állapottérnek elemei voltak: azaz a program állapottere állandó volt. Egy valódi program esetén azonban ez nincs így: időnként újabb komponensek (új változók) keletkeznek, majd később megszűnnek, azaz az állapottér dinamikusan változik.

A program általam adott definíciója visszacsempészi az alapvetően statikus szemléletű relációs modellbe a program dinamikus természetét, miközben a megoldás vizsgálatához továbbra is a megszokott statikus szemléletet alkalmazhatjuk.

3.1. Tézis: A program matematikai fogalma átalakítható úgy, hogy megjelenjen benne a program azon természete, hogy az állapottere változik egy-egy végrehajtása során. Mindemellett a relációs modell keretében bevezetett helyesség vizsgálati módszerek továbbra is érvényesek maradnak.

Új, a modellben korábban nem szereplő, fogalom az ***alap-állapottér***, amely az absztrakt program interfésze. A program ennek változóin keresztül kommunikál a környezetével: kezdőállapotát a környezettől kapja, végállapotát – ha leáll – a környezetének adja át. Egy program végrehajtása során az alap-állapottér komponensein kívül új komponensek jöhetnek létre, és szűnhetnek meg. Ennél fogva a program egy végrehajtása során dinamikusan változik az állapottér, de az éppen aktuális állapottérnek mindig komponensei lesznek az alap-állapottér komponensei. A program változóit tehát két csoportba sorolhatjuk: a mindvégig létező alap-állapottérbeli ***alapváltozókra*** és a futás közben létrejövő, majd megszűnő ***segédváltozókra***.

A ***program*** matematikai fogalma továbbra is egy reláció, amely megmutatja, hogy az alap-állapottér egy-egy állapotából (kiinduló állapotból) indulva a program milyen végrehajtásokat (állapot-sorozatokat) produkálhat. Ugyanahhoz a kiinduló állapothoz több végrehajtási sorozat is tartozhat. Egy végrehajtás lehet véges, illetve végtelen hosszú; leállhat hibás állapotban, illetve speciálisan holtpontban. A hibás működést egy speciális (*fail*) végállapotú sorozat jelzi. A végrehajtások első állapota az alap-állapottérbeli kiinduló állapot, további állapotai kiegészülhetnek segédváltozókkal, amelyek legkésőbb a működés leállása előtt (ha ez bekövetkezik) megszűnnek. Egy végrehajtás tehát olyan állapot-sorozattal írható le, amelynek állapotai olyan állapotterekhez tartoznak, amelyek altérként tartalmazzák a program alapállapotterét (illetve speciális esetben a *fail* hibás állapot). Egy végrehajtási sorozat utolsó állapota, amennyiben az nem a *fail*, megállapodás szerint alap-állapottérbeli állapot, azaz a segédváltozók – amennyiben egy végrehajtás véges és nem a *fail* állapotban végződik – legkésőbb az utolsó lépésben megszűnnek.

A program alap-állapottere valójában egy önkényesen kijelölt állapottér, amely a programváltozók átnevezésével, törlésével, illetve bővítésével változtatható, és ha kell, hozzáigazítható a környezethez úgy, hogy ettől a program természete ne változzon meg. Nyilvánvalóan nem változik meg egy program működése akkor, ha az alap-állapotterének változóit átnevezzük, feltéve, hogy az új változónevek továbbra is egyediek maradnak. Egy program új változóval történő bővítése ugyancsak hatástalan a program működésére. Egy program működése attól sem változik meg, ha a változóinak státuszán módosítunk: néhányat alapváltozóból segédváltozóvá, néhányat segédváltozóból alapváltozóvá minősítünk át. Egy alapváltozó ugyanis könnyedén átminősíthető segédváltozóvá úgy, hogy az csak a

program elindulásakor jön létre, és a program leállásakor megszűnik. Egy segédváltozóból is lehet alapváltozó, ha már eleve létezőnek tekintjük, ezért nem kell sem létrehozni sem megszüntetni, élettartama mindegyik végrehajtás teljes hosszára kiterjed.

A program fogalmának fenti megváltoztatása miatt a programok konstrukcióival kapcsolatban két megjegyzést kell tennünk.

Az egyik az, hogy az ismert elemi programokat (üres program, hibás program, értékadás) két újabbal kell kiegészíteni: segédváltozó létrehozása, illetve segédváltozó megszűntetése. Ezek az elemi programok összevonhatók más elemi programokkal: egy lépésben be lehet vezetni új komponenseket, miközben másokat meg lehet szüntetni; egy értékadás összevonható a baloldalán szereplő segédváltozó létrehozásával, vagy a jobboldalán levő segédváltozó megszűntetésével.

A másik megjegyzés a három strukturált szekvenciális programszerkezettel kapcsolatos. A szekvencia, az elágazás, és a ciklus szakirodalomban található definíciói mindig közös állapottéren megfogalmazott programkomponensekből és ugyanezen állapottéren értelmezett logikai függvényekből (feltételekből) konstruálnak új programot. De miért ne térhetne el egymástól a komponensek állapottere, a mi esetünkben alap-állapottere. Ezért a konstrukciók során először meg kell állapodni a közös alap-állapottérben, erre átalakítani a konstrukció komponenseit. Hangsúlyozni kell, hogy ez a kérdés független a program most bevezetett új definíciójától, viszont az alap-állapottér előbb tárgyalt átalakításai teszik lehetővé azt, hogy kielégítően tisztázzuk ezt a kérdést.

Térjünk rá végül a helyesség vizsgálatra. A program itt módosított fogalmához ugyanúgy adható meg a program hatását leíró programfüggvény, mint korábban, hiszen a nem hibásan leálló végrehajtások most is ugyanazon állapottérből indulnak, mint amelyikben leállnak. Ennél fogva az azonos alap-állapottéren felírt feladat és program esetében a megoldás matematikai fogalma változatlan marad.

Találkozhatunk olyan esetekkel, amikor a megoldandó feladat állapottere és a választott program alap-állapottere eltér egymástól. A program alap-állapotterének rugalmasságából adódik, hogy ilyenkor a program alap-állapottere (az alap-változók átnevezésével, törlésével, bővítésével) hozzáigazítható a feladat állapotteréhez. A helyességvizsgálathoz elég egyetlen olyan hozzáigazítás találni, amelyre a program megoldja a feladatot.

Összefoglalás:

A 3.1. tézist megvalósító eredményeket a [15] és [17] -ben publikáltam:

1. Módosítottam és formálisan is definiáltam a program fogalmát: kiegészült a dinamikusan változó állapottérrel.
2. Új fogalomként vezettem be az alap-állapottér fogalmát.
3. Pontosítottam a programszerkezetek definícióit. Két új elemi programot is be kellett vezetni: új változó létrehozása, illetve megszűntetése.
4. A program fogalmának módosítása nem változtatta meg a program függvény fogalmát, így a korábbi helyesség vizsgálati módszerek érvényesek maradtak.
5. Lényeges egyszerűbb lett annak eldöntése, ha egy feladatot egy olyan programmal akarunk megoldani, amelyik állapottere nem azonos a feladatéval, bővebb vagy szűkebb annál, esetleg más változóneveket használ. Megmutattam, hogy egy program alap-állapottere a program természetének megváltoztatása nélkül hozzáigazítható (változó átnevezéssel, bővítéssel, szűkítéssel) a feladat állapotteréhez. A megoldás fogalmát ezekután elegendő azonos alap-állapotterű feladatra és programra vizsgálni.

3.2. Alprogramok az absztrakt programleírásban

A program fogalmának 3.1. alfejezetben bevezetett módosítása lehetőséget ad arra, hogy a relációs modellre támaszkodó absztrakt programleírásokban (például a struktogrammokban) az alprogramok fogalmát a konkrét programozási nyelvekben megszokott módon bevezessük.

3.2. Tézis: Az algoritmusok absztrakt leírásaiban definiálhatjuk az alprogramok és azok hívásának fogalmait.

Egy program-leírás bármelyik ***részprogramja*** kiemelhető az őt tartalmazó programból. Ez a továbbiakban nemcsak részprogramként kezelhető, hanem például ún. ***makróként*** is, amelynek bizonyos lexikális elemeit (változó nevek, kifejezések) a tartalmazó programba történő visszahelyettesítésekor más lexikális elemekkel helyettesíthetők, vagy akár függvény- vagy eljárásszerűen meghívható ***alprogramként*** is. Az alábbiakban csak az alprogramokról esik szó.

Amikor az alprogramot elkülönítjük az azt tartalmazó programtól, akkor arra valahogy hivatkozni kell. Mivel minden alprogram (ahogy maga a program is) egy-egy feladatot old meg, és minden feladat leírható egy (általában nem megengedett, azaz közvetlenül nem implementálható) értékadással, ezért az alprogramok ezekkel az értékadásokkal címkézhetők. Ezt az értékadást egyszerre használhatjuk a alprogram azonosítására – ez a ***alprogram feje*** –, valamint a alprogram helyének – ***hivatkozásának*** vagy ***hívásának*** – jelölésére is. Eljárásszerű hívás esetén ez az értékadás lesz az eljárás ún. ***hívó utasítása***, függvényszerű hívásakor ehhez elég az értékadás jobboldali kifejezését, az ún. ***hívó kifejezést*** használni.

Célszerű kikötni, hogy az alprogram kizárólag az ott bevezetett lokális változókat használhatja (esetleg engedhetünk ezen a megkötésen azzal, hogy megengedjük az alprogramot hívó program alapváltozóinak olvasását). Az alprogram segédváltozói közül néhány szerepel az alprogram azonosítására (hivatkozására) szolgáló értékadásban is: ezek közül az értékadás bal oldalán állók az ún. ***bemenő paraméterváltozók***, jobboldalán állók az ***eredmény paraméterváltozók***. (Ugyanaz a változó lehet egyszerre mindkettő.) A paraméterváltozók ugyanazt a szerepet töltik be, mint a konkrét programozási nyelvekben. Az alprogram hívásakor a bemenő paraméterváltozók megkapják kezdőértékként a hívó utasításban a helyükön álló velük azonos típusú kifejezések értékét. Az alprogram leállásakor az eredmény paraméterváltozók értékei eljárásszerű hívás esetén átmásolódnak a hívó utasításban a helyükön álló változókba, függvényszerű hívás esetén a hívó kifejezés értékeként adódnak vissza.

Összefoglalás:

A 3.2. tézis megvalósítását a [17] részletes tárgyalja, és számos példát is mutat a struktogrammok használatánál az alprogramok jelölésére.

1. A nem-megengedett értékadást címkeként használjuk egy program részprogramjának jelölésére, és ezt bármelyik olyan absztrakt program leírásnál alkalmazhatjuk, amelyik használ értékadásokat. Ez egyértelműen megmutatja, hogy milyen inputból milyen outputot állít elő a részprogram.
2. A részprogram címkéjét használjuk a beágyazó programban a részprogram helyének jelölésére is, amely jelölheti a részprogramnak alprogramként történő meghívását is.

3.3. Programszerkezetek szemantikájának pontosítása

A strukturált programok formális helyeségbizonyítására Hoare egy deduktív módszert dolgozott ki [11], amelynek alapját az elemi programok és a programkonstrukciók ún. levezetési szabályai képezik. Hoare ezt a módszert determinisztikus szekvenciális strukturált programokra fogalmazta meg, és ezt általánosította Dijkstra nemdeterminisztikus programokra [12], Gries párhozamos programokra [13].

A programszerkezetek közül mind az elágazás, mind a ciklus, mind a várakozó utasítás programállapotokon értelmezett logikai függvényeket (feltételeket) használ. A szakirodalomban leírt elméletek ezeket a függvényeket teljesnek, mindenhol értelmezettnek tekintik. [16] Csakhogy a valóságban könnyen előfordulhat az, hogy egy program elágazás-, ciklus-, vagy várakozási feltétele nem minden esetben értelmezett, azaz a feltételt adó logikai függvény parciális, és ekkor a program hibásan működik. A formális helyességbizonyításnak kezelnie kell ezt problémát: nem mondhatjuk egy programra, hogy helyes, ha előfordulhat a működése során, hogy egy feltételt nem tud kiértékelni.

3.3. Tézis: A formális helyességbizonyítás levezetési szabályait úgy kell módosítani, hogy azok számoljanak azzal a lehetőséggel, hogy a vizsgált program feltételeit adó logikai függvények parciálisak.

Legyen $\pi \in A \to \mathbb{L}$ egy parciális logikai függvény (A az állapottere a logikai függvényt tartalmazó programnak). Ez azt jenlenti, hogy egy tetszőleges $a \in A$ állapotra a $\pi(a)$ lehet igaz, lehet hamis, de lehet, hogy nincs értelmezve. Azt, hogy egy adott $a \in A$ állapotban π értelmezve van úgy fogjuk jelölni, hogy $\pi(a) \vee \neg\pi(a)$, vagy röviden $\pi \vee \neg\pi$.

Ennek fényében egyrészt újra kell a ciklus, az elágazás, és a várakozó utasítás szemantikáját definiálni: ezek azon állapotokból indulva hibás működést eredményezzenek, amely állapotokban a programszerkezet valamelyik feltétele értelmetlen.

Másrészt e szerkezetek levezetési szabályait is módosítani kell. A $\pi \in A \to \mathbb{L}$ ciklusfeltételű (**while** π **do** S_0 **od**) ciklus levezetési szabályában az előfeltételeket ki kell egészíteni az $I \Rightarrow \pi \vee \neg\pi$ állítással, azaz ha egy állapot kielégíti a ciklus $I{:}A \to \mathbb{L}$ invariáns állítását, akkor arra a ciklusfeltétel legyen értelmezett. A $\pi_1, \ldots, \pi_n \in A \to \mathbb{L}$ feltételekkel megadott többágú nemdeterminisztikus (**if** $\pi_1 \to S_1 \,\square\, \ldots \,\square\, \pi_n \to S_n$ **fi**) elágazás levezetési szabályában az előfeltételeket ki kell egészíteni a $Q \Rightarrow (\pi_1 \vee \neg\pi_1) \wedge \ldots \wedge (\pi_n \vee \neg\pi_n)$ állítással, azaz ha egy állapotra teljesül az elágazás $Q{:}A \to \mathbb{L}$ előfeltétele, akkor arra az állapotra az elágazás minden feltétele értelmes kell legyen. A $\pi \in A \to \mathbb{L}$ őrfeltételű (**await** π **then** S_0 **ta**) várakozó utasítás levezetési szabályában új feltételként jelenik meg a $Q \Rightarrow \pi \vee \neg\pi$ állítás, azaz ha egy állapot kielégíti a várakozó utasítás $Q{:}A \to \mathbb{L}$ előfeltételét, akkor arra az őrfeltétel értelmezett kell legyen. A módosított levezetési szabályok helyességének bizonyítása a fenti módosítások hatására csak kicsit változik.

Összefoglalás:

A 3.3. tézis megvalósítását a T. Gregorics és Zs. Borsi, *An unified approach of program verification*, Acta Universitatis Sapientiae Informatica, 9 (1), pp. 65-82, 2017. [16] folyóiratban Borsi Zsolttal közösen publikáltam. Ebben saját eredményeim:

1. Parciális logikai feltételek alkalmazása a program konstrukciókban, és ennek felhasználásával a ciklus, az elágazás, és a várakozó utasítás szemantikájának újra definiálása.
2. Kimondtam a megváltozott szemantikájú ciklus, elágazás, és várakozó utasítás levezetési szabályait, és beláttam ezek helyességét.
3. Pontosítottam a párhuzamos programszerkezet szemantikájának leírását.

4. Visszavezetés felsorolókkal

Az analóg módon történő programozás a programkészítés általánosan ismert és alkalmazott technikája. Ennek során egy feladatot úgy kell megoldani, ahogyan egy hasonló feladatot már korábban megoldottunk, azaz a megoldó program előállításánál a hasonló feladat megoldó programját mintaként használjuk fel. (Ez az ötlet nem kuriózum, az informatikán kívül is széleskörben ismert.) Az ***analóg programozást*** lehet ösztönösen és tudatosan is alkalmazni, elvégezhetjük elnagyoltan vagy precízen, követhetjük laza ajánlásként vagy pontos technológiai szabványként. Az ELTE IK programozás oktatásának egyik sajátos vonása az a ***visszavezetésnek*** nevezett analóg programozási módszer, amely mind a kitűzött feladatnak, mind az arra hasonlító korábban megoldott mintafeladatnak a precíz leírásából, formális specifikációjából indul ki, mert ezek alkalmasak arra, hogy a két feladat közötti analógiát pontosan felfedjük: felismerjük a hasonlóságot, de világosan lássuk az eltéréseket. A visszavezetés során a mintafeladat megoldó programját (a mintaprogramot) alakítjuk át úgy, hogy figyelembe vesszük a kitűzött- és a mintafeladat specifikációkban felfedett eltéréseket, és ezeket az eltéréseket átírjuk a mintaprogramban, ezáltal közvetlenül megkapjuk a kitűzött feladatot megoldó programot. Ilyenkor tehát nem a mintaprogram előállítási folyamatát, az előállításának lépéseit másoljuk le, ismételjük meg, hanem magát a mintaprogramot adaptáljuk a kitűzött feladatra. [17]

A visszavezetés során mintaként használt feladat-program párokat (ahol a program megoldja a feladatot) ***programozási tételeknek*** hívjuk. Az elnevezés onnan származik, hogy egy mintafeladat-program párt egy matematikai tételhez hasonlóan alkalmazunk: ha egy kitűzött feladat hasonlít a mintafeladatra, akkor a mintaprogram lényegében, kis módosításokkal, megoldja a kitűzött feladatot is.

A visszavezetés sikere azon múlik, hogy rendelkezünk-e kellő számú, változatos és eléggé általános programozási tétellel ahhoz, hogy egy új feladatot ezek valamelyikével megoldhassuk. A szakirodalomban már régen kialakultak a nevezetes programozási tételek: összegzés, számlálás, maximum kiválasztás, lineáris (szekvenciális) keresés, kiválasztás. A számlálás ugyan egy speciális összegzés, mégis önálló tételként jelenik meg. Vannak olyan iskolák, ahol ehhez hasonlóan külön tételt képez az összegzés néhány más speciális esete: feltételes összegzés, másolás, ki- illetve szétválogatás, vagy megjelenik az ún. feltételes maximum keresés, esetleg a bináris (logaritmikus) keresés. Ezeket a tételeket egydimenziós tömbök feldolgozására, vagy – általánosabban – egész számok intervallumán értelmezett függvények értékeinek feldolgozására vezették be, ennek megfelelően beszélhetünk tömbös-, illetve intervallumos programozási tételekről. (Például tömbös számlálás az, ha egy egész számokat tartalmazó tömb páros elemeinek számát kell megadnunk, intervallumos számlálás pedig az, amikor egy egész számok intervalluman értelmezett egész értékű függvény páros értékeinek számát keressük.) Az én ötletem az volt, hogy az ismert tételeket ún. felsorolásokkal megfogalmazott feladatokra általánosítsuk abból a célból, hogy ezzel a programozási tételek felhasználási körét szélesítsük.

Az alábbi három alfejezetben erre az ötletre épülő eredményeimet mutatom be.

4.1. Felsoroló fogalma

Sok feladat megoldásánál valamilyen gyűjtemény elemeit kell feldolgozni. Egy gyűjtemény lehet egy összetett adatszerkezetű objektum, amely elemi értékek sokaságát tartalmazza (pl. halmaz, tömb, szekvenciális inputfájl), de lehet virtuális is, például amikor egy egész szám valódi osztóit, vagy két egész szám által behatárolt zárt intervallum egész számait kell feldolgoznunk.

A gyűjtemények elemeinek feldolgozásához fel kell sorolni a gyűjteményben tárolt elemeket, de nem szerencsés, ha a felsorolást végző műveletek egyben a gyűjtemény műveletei is, mert ez nem teszi lehetővé például, hogy egy gyűjteményt egyszerre több felsorláshoz is felhasználjunk. Célszerű a felsorolást végző műveleteket a gyűjteménytől elkülönített objektumhoz kötni. Ezt nevezzük felsoroló (bejáró) objektumnak (enumerator, iterator).

4.1. Tézis: Bevezethető a felsoroló objektum absztrakt fogalma úgy, hogy azzal a programozási feladatok megoldásának tervezésénél is kényelmesen dolgozhassunk, és illeszkedjen a programozási nyelvekben megtalálható iterátorok jelentéséhez.

Egy ***felsoroló objektum*** véges sok elemi érték felsorolását teszi lehetővé azáltal, hogy rendelkezik a felsorolást végző műveletekkel: rá tud állni a felsorolandó értékek közül az elsőre (*first()*) vagy a soron következőre (*next()*), meg tudja mutatni, hogy tart-e még a felsorolás (*end()*) és vissza tudja adni a felsorolás aktuális értékét (*current()*).

Minden felsoroló objektumra gondolhatunk úgy, mint egy véges elemű sorozatra. A felsoroló reprezentációjában ennek a sorozatnak természetesen nem kell megjelenni, sőt (a bejáró tervezési minta ajánlása alapján [19]) annak a gyűjteménynek is csak a hivatkozására van szükség, amelynek elemeit a felsorolás bejárja. Néhány nevezetes felsoroló definícióját megtaláljuk [17] tankönyvben.

Egy felsoroló objektumnak három állapotát különböztethetjük meg. Beszélhetünk az „*indulásra kész*" állapotról, amelyet a *first()* művelet szüntet meg. Általában nem-definiált, hogy ebben az állapotban a másik három művelet hogyan viselkedjen, ezek a műveletek ilyenkor hibát okozhatnak, ezért nem használhatók. A „*felsorolás alatt*" állapotban viszont éppen fordítva, a *first()* nem használható, a másik három művelet igen. Ebben az állapotban az *end()* művelet hamis eredményt ad. A felsorolás utolsó elemének elérése után egy újabb *next()* művelettel a felsoroló „*befejezett*" állapotba kerül. Ezt követően még megengedett a *next()* és az *end()* művelet használata, de a *first()* és a *current()* nem. Nem kell viszont bajlódnunk ezzel a meglehetősen bonyolult feltételrendszerrel, ha a ***gyűjtemények feldolgozásánál*** mindig az alábbi algoritmus szerkezetet használjuk (*t*:*enor*(*E*) a felsoroló objektum).

```
t.first()
while not t.end() loop
    feldolgoz(t.current())
    t.next())
endloop
```

A modern imperatív programozási nyelvekben ennek kódolására használhatjuk az ún. „foreach" ciklusokat.

Összefoglalás:

A 4.1. tézishez kapcsolódó eredményeimet a [17] és a [19] publikációk tartalmazzák.

1. Definiáltam általánosan a felsoroló objektum fogalmát.
2. Konkrét példákat mutattam jól ismert struktúrák elemeinek felsorolását végző felsorolókra.
3. Javaslatot tettem a felsorolásos feldolgozással a felsorló műveletek biztonságos használatára.

4.2. Programozási tételek felsorolókra

Egy felsoroló feldolgozása igen változatos tevékenység lehet: összegezhetjük a felsorolt értékeket, megszámolhatjuk az adott tulajdonságúakat, kiválaszthatjuk a legnagyobbat. Ezek éppen azok az alapfeladatok, amelyek megoldására programozási tételeket szoktak bevezetni, csakhogy ezeket a tételeket tömbökre, esetleg egész számok intervallumán értelmezett függvényekre, néha közvetlenül sorozatokra, vagy szekvenciális inputfájlokra szokták megfogalmazni. Most az ezeket mind magába foglaló, a felsorolóra kimondott változataikat vezetjük be.

4.2. Tézis: Programozási tételek felsorolókra történő általánosítása jelentős támogatást ad a programtervezés visszavezetéses módszerének. Ezek az új tételek általánosabbak a szakirodalomból ismert programozási tételeknél, hiszen speciális esetként tartalmazzák azokat.

A felsorolós programozási tételek alapvető tulajdonsága, hogy egy felsoroló objektum szolgáltatja számukra a feldolgozandó elemeket. Ezért mindegyik programozási tétel algoritmusa a gyűjtemények felsorolóval történő feldolgozásának algoritmus sémájára épül (lásd előző alfejezet).

Az új tételek specifikációiban fontos elem a *t*:*enor*(*E*) felsoroló. Amikor specifikáció utófeltételében a felsorolás elemeire hivatkozunk, akkor az megtehető indexeléssel (hiszen a felsorolás egy sorozattal írható le), de ennek csak speciális esetben, egy tömb vagy egy konkrét sorozat felsorolásakor van jelentősége, hiszen egy halmaz, vagy egy szekvenciális inputfájl elemeinek felsorolásakor már nincs értelme az elemek indexéről beszélni. Ezért a felsorolós programozási tételek specifikációjában az $e \in t$ kifejezéssel fejezzük ki a felsorolás tényét (tehát az $\in$ nem az ismert „eleme” halmazműveletet jelöli – hiszen *t* nem egy halmaz), az *e* segítségével pedig a *t* felsorolás egymás után bejárt elemeire tudunk hivatkozni. Az algoritmusban ezt az elemet a *t.current()* segítségével kérdezhetjük le.

A felsorolós programozási tételek nem közvetlenül a felsorolt elemeket dolgozzák fel (adják össze, számolják meg, stb.), hanem az elemekhez hozzárendelt értékeket. Ezeket az értékeket bizonyos tételeknél (pl. összegzés, maximum kiválasztás) egy $f:E \to H$ függvény (ez lehet identitás is), másoknál (pl. számlálás, keresés) egy $felt:E \to \mathbb{L}$ logikai függvény állítja elő. Ennek következtében a feldolgozás során általában nem közvetlenül a *t.current()* által visszaadott értékeket, hanem az *f*(*t.current()*) vagy *felt*(*t.current()*) értékeket kell megvizsgálni.

A maximum kiválasztás, illetve feltételes maximumkeresés esetén a feldolgozás eredményei között szerepel mind a megtalált maximális érték, mind pedig az elem, amelyhez a maximális érték tartozik. Mivel a maximális értékre a ciklusmag elágazásának feltételében szükség van, csak a maximális elem nyilvántartása nélkülözhető, ha a konkrét feladat azt nem kéri. Speciális eset az, amikor a *f* függvény identitás, azaz egy elem és annak értéke megegyezik.

A lineáris keresésnél és a kiválasztásnál az eredmények között szerepel maga a felsoroló is. Ennek az oka az, hogy ennél a két tételnél korábban is leállhat a feldolgozás, mint hogy a felsorolás véget érne, és ekkor maradhatnak még fel nem sorolt (fel nem dolgozott) elemek. Ezeket az elemeket további feldolgozásnak lehet alávetni, ha a felsorolót tovább használjuk. (Felhívjuk azonban a figyelmet arra, hogy ha egy már korábban használt felsorolóval dolgozunk tovább, akkor nem szabad majd a *first()* művelettel újraindítani a felsorolást.) Kiválasztásnál nem kell a felsoroló által szolgáltatott értéksorozatnak végesnek lennie, hiszen ez a tétel más módon garantálja a feldolgozás véges lépésben történő leállását. A lineáris keresésnek itt is ismert az ún. „optimista” változata, amely azt vizsgálja, hogy vajon a felsorlás minden elemére teljesül-e a megadott feltétel.

A programozási tételeknek jól ismertek azok a változatai, amelyek egy tömb elemeit, vagy egy egész számok intervallumán értelmezett függvény értékeit dolgozzák fel. Az intervallumon értelmezett függvényekre megfogalmazott változatok valójában általánosításai a tömbökre megfogalmazott

változatoknak, hiszen egy tömb felfogható egy intervallumon (ami a tömb indextartománya) értelmezett függvénynek. Egy intervallum klasszikus felsorolása egyesével halad alulról felfelé: erre épül a tömbös és az intervallumos programozási tételek működése is. Egy intervallumot azonban másképpen is fel lehet sorolni, nemcsak a klasszikus módon, ettől a feldolgozás módja (legyen az számlálás vagy maximumkiválasztás) nem változik. Ez világít rá arra, hogy a felsorolós programozási tételek általánosításai a tömbös és az intervallumon értelmezett függvényekre épülő programozási tételeknek. A megoldó algoritmusokban az intervallumot befutó egész típusú indexváltozó helyett a *current()* kifejezést használjuk, az indexváltozónak kezdeti értékadását a *first()*, növelését a *next()* helyettesíti, az *end()* művelet pedig annak az általánosítása, hogy az indexváltozó elérte a számára megengedett felső határt. Mindezt nem jelenti azt, hogy a konkrétabb (tömbös, egész intervallumos) programozási tételek feleslegessé válnának: a megoldandó feladat dönti el, hogy a tételek melyik változatát alkalmazzuk.

Összefoglalás:

A felsorolós programozási tételekkel kapcsolatban megfogalmazott 4.2. tézist az alábbi publikációim támasztják alá:

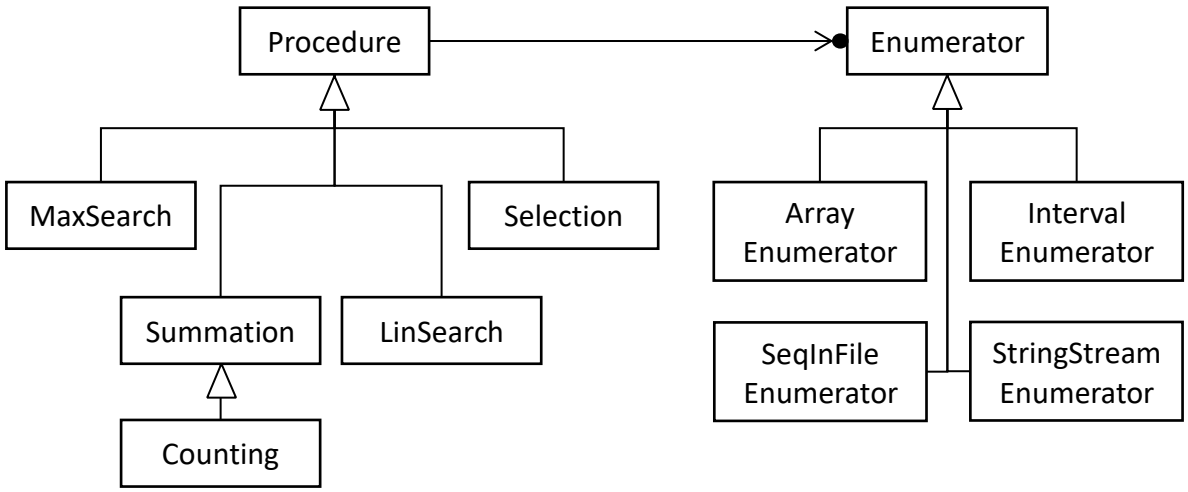
1. A [17], illetve a [19] publikációkban megadtam felsorolókra az összegzésnek, számlálásnak, maximum kiválasztásnak, kiválasztásnak, lineáris keresés két változatának, és a feltételes maximum keresésnek a programozási tételeit.
2. Megmutattam az új programozási tételek robosztusságát: az ELTE IK BSc PTI részére készült egyetemi jegyzetben [17] számos feladat megoldásával találkozhatunk; a [35] a visszalépéses keresés algoritmusainak objektumelvű modellezésével foglalkozik, és ezeket a felsorolós lineáris keresés segítségével implementálja.
3. Elemeztem a programozási tételek különböző absztrakciós szinten bevezetett változatait (egy dimenziós tömb elemeire, egyész intervallum elemeire, felsorolóra): a [20] egyetlen feladatot old meg többféleképpen, a maximum kiválasztás programozási tételének különböző absztrakciós szinten bevezetett változataival; a [21] az összegzés programozási tétele felhasználásának változatosságáról szól, és ezek között a felsorolós változat is megjelenik.
4. Bevezettem egy speciális jelölésrendszert a nevezetes (egész intervallum, vektor, mátrix, halmaz, szekvenciális inputfájl) gyűjteményekre alkalmazott felsorolós programozási tételeknek a feladatok specifikációjában történő leírására. [17]

4.3. Osztály-sablon könyvtár

A felsorolós programozási tétel család hasznosságának igazolására egy C++ nyelven írt osztály-sablon könyvtárat készítettem, amelyre támaszkodva olyan nehézségű feladatokat lehet megoldani, amelyekkel a felsőoktatás különböző informatikus képzésein a hallgatók a képzésük első két félévében szoktak találkozni.

A könyvtár és annak használata elsősorban objektum-orientált technológiára épül, de sablon (template) paramétereket is használ. A könyvtár osztály-sablonjaiból saját osztályokat származtathatunk úgy, hogy azok virtuális metódusait felüldefiniáljuk (sablon függvény tervezési minta [36]) egyedi viselkedést tervezünk, majd az így kapott osztályokból példányosított tevékenység-objektumokba speciális objektumokat építünk (függőség befecskendezés). Ennek eredményeként minden olyan programrész, amelyet programozási tételre vezethetünk vissza egy-egy tevékenység-objektumként jelenik meg, amelyet aztán végrehajthatunk.

4.3. Tézis: Megadható olyan program könyvtár, amely támogatást ad mind a programozási feladatok visszavezetéses módszertannal történő megoldásának megértéséhez, mind az objektum elvű programtervezés elsajátításához.



1. ábra. Programozási tételek osztály-sablon kód-könyvtára

A könyvtárban három féle osztályt találunk. A felsorolós programozási tételeket általánosan leíró osztály-sablonokat és ezek ősosztályát (*Procedure*); néhány nevezetes felsorolónak az osztály-sablonját és ezek ősosztályát (*Enumerator*), valamint két speciális segédosztályt, amelyet a maximum keresés tételének maximumot, illetve minimumot kereső változatainak példányosításához használunk.

Nemcsak a könyvtár felhasználása igényel objektum-orientált technológiát, de maga a könyvtár is ennek szellemében készült. A *Procedure* osztály-sablon az összes programozási tétel őse, amely azt az általános feldolgozási stratégiát fogalmazza meg, amely során egy felsoroló által előállított elemeket egyesével dolgozzuk fel (a felsorolóra az *enor* pointer hivatkozik).

```
init();
for (enor->first(); !enor->end(); enor->next()) {
  body(enor->current());
}
```

A feldolgozást a *Procedure* ősosztály *run()* metódusa tartalmazza, amely a származtatás során már nem írható felül (ezt C++ nyelvben a „final” kulcsszó jelzi). Ezt a metódust hajtja végre mindegyik nevezetes programozási tétel úgy, hogy egyedi módon implementálják (felülírják) az *init()* és a *body()* részeket. Mindeközben újabb, a konkrét tételre jellemző virtuális metódusokat vezetnek be, amelyeket a konkrét alkalmazásoknál kell felüldefiniálni: a konkrét feladat megoldását leíró osztályban. Ilyen például egy lineáris keresés vagy egy számlálás feltételét megadó logikai értékű függvény, amelynek bemenete az éppen felsorolt elem, vagy a maximum kereséseknél a felsorolt elemekhez értéket rendelő függvény, amely értékeket már össze lehet hasonlítani abból a célból, hogy a legnagyobbat ki tudjuk választani. Természetesen a konkrét feladat megoldását leíró osztályban, amely valamelyik programozási tételt leíró osztályból származik, már nem lehet felülírni a programozási tétel *init()* és *body()* metódusait (ezek tehát „final” tulajdonságúak).

A felsorolókat leíró osztályok őse egy interfész (*Enumerator*), egy teljesen absztrakt osztály-sablon, amely bevezeti, de nem definiálja a felsoroló műveleteket. Ebből származnak a konkrét felsorolók osztályai, amelyek már reprezentációt és implementációt is tartalmaznak.

Az osztály-sablon könyvtár természetesen nem ipari alkalmazások készítéséhez készült. Nem hiszem, hogy a gyakorlatban való felhasználása célszerű lenne. Egy programozási tétel (lényegében egy ciklus) implementálása ugyanis önmagában nem túl nehéz feladat, ezért sokkal könnyebb közvetlenül kódolni, mint egy összetett osztály-sablon könyvtárból származtatni. Ennek a könyvtárnak a használata a programozás egyetemi tanításában alkalmazható. Az elmúlt néhány évben, amikor az oktatásban bevezettük, több képzési cél elérését is támogatta. Egyrészt kiemelte az elemzés és tervezés fontosságát már az egyszerű programozási feladatok megoldásánál is, miközben az implementálást és a tesztelést háttérbe szorította. A könyvtár segítségével minden olyan feladatot meg lehet oldani ciklusok kódolása nélkül (azaz a ciklusok írása során elkövetett kitesztelendő hibákkal nem kell foglalkozni), amelyet – akár egymásba ágyazottan – valamilyen felsorolót használó programozási tételre lehet visszavezetni. Másrészt elmélyítette a programozási tételekre történő visszavezetés módszerének ismeretét. Harmadrészt, mivel a könyvtár osztály-hierarchiára épül, jól megértethetők voltak vele az objektum-orientált programozás fogalmai: az egységbezárás, az elrejtés, az öröklődés, a futási idejű polimorfizmus, de jó példákat szolgáltatott a tervezési mintákra is. A könyvtár használatával igen szép megoldásokat tudunk előállítani. Programozói szemmel például nagyon érdekes, hogy az előállított megoldásokban mindössze egyetlen ciklus lesz, mégpedig a programozási tételek ősosztály-sablonjának előbb említett *run()* metódusában, az alkalmazás során hozzáadott kódban pedig egyáltalán nem kell majd ciklust írni.

Összefoglalás:

A 4.3. tézist alátámasztó eredményeim a [22], és a [18] publikációkban találhatóak:

1. Elkészítettem egy osztály-sablon könyvtárat C++ nyelven, amellyel megoldhatók a felsorolós programozási tételekre visszavezethető feladatok.
2. A könyvtár használata támogatja a visszavezetéses programtervezési paradigma megértését, valamint az objektum-orientált programozási paradigma elsajátítását.
3. A könyvtár használatával a feladatok megoldása strukturális szempontból igen egyszerű lett: a programozónak nem kell ciklust írnia.

5. Irodalomjegyzék

[1] I. Futó, Mesterséges intelligencia, I. Futó, Szerk., Budapest: Aula, 1999.

[2] T. Gregorics, „Diszkrét módszerek: Heurisztikus gráfkereső algoritmusok,” in *Informatikai Algoritmusok 3.*., A. Iványi, Szerk., Budapest, ELTE Eötvös Kiadó, 2013, pp. 1597-1636.

[3] T. Gregorics, Heurisztikus útkereső algoritmusok (online), Budapest: ELTE IK, 2014.

[4] T. Gregorics, „Artificial intelligence, elektronikus jegyzet,” ELTE IK Digitális Könyvtár, http://www.inf.elte.hu/karunkrol/digitkonyv/Lapok/2014-esjegyzettámogatásipályázatotnyertoktatókésjegyzeteik.aspx, 2015.

[5] T. Gregorics, „Which of Graphsearch Versions is the Best?,” *Annales Univ. Sci. Budapest., Sect. Comp.,* %1. kötet15, pp. 93-108, 1995.

[6] T. Ásványi, és T. Gregorics,, „A More Effective Version of Algorithm A,” *Annales Univ. Sci. Sect. Comp.,* %1. kötet17, pp. 33-48, 1998.

[7] R. Giachetta, T. Gregorics, Z. Istenes és S. Sike, „Address Standardization,” *Annales Univ. Sci. Sect. Comp.,* %1. kötet37, pp. 5-18, 2012.

[8] T. Gregorics, „Remarks on the algorithm A**,” *Acta Universitatis Sapientiae, Informatica,* %1. kötet6 (2), pp. 190-205, 2014.

[9] B. Gregorics, T. Gregorics, G. F. Kovács, A. Dobreff és G. Dévai, „Textual Diagram Layout Language and Visualization Algorithm,” in *ACM/IEEE*, Ottawa, ON, Canada, 2015.

[10] S. Nagy és T. Gregorics, „Searching in AND/OR graphs”.

[11] C. A. Hoare, „Proof of correctness of data representations,” *Acta informatica,* %1. kötet1, p. 271–281, 1972.

[12] E. W. Dijkstra, A Discipline of Programming., Prentice-Hall, 1976.

[13] D. Gries, The Science of Programming, Springer Verlag, 1981.

[14] Á. Fóthi, Bevezetés a programozáshoz, Budapest: ELTE Eötvös Kiadó, 2005.

[15] T. Gregorics, „Concept of abstract program,” *Acta Universitatis Sapientiae Informatica,* %1. kötet4 (1), pp. 7-16, 2012.

[16] T. Gregorics és Z. Borsi, „An unified approach of program verification,” *Acta Universitatis Sapientiae Informatica,* %1. kötet9 (1), pp. 65-82, 2017.

[17] T. Gregorics, Programozás - Tervezés, Budapest: ELTE Eötvös Kiadó, 2013.

[18] T. Gregorics, Programozás - Megvalósítás, Budapest: ELTE Eötvös Kiadó, 2013.

[19] T. Gregorics, „Programming theorems on enumerator,” *Teaching Mathematics and Computer Science,* %1. kötet8 (1), pp. 89-108, 2010.

[20] T. Gregorics, „Abstract levels of programming theorems,” *Acta Universitatis Sapientiae Informatica,* %1. kötet4 (2), pp. 247-259, 2012.

[21] T. Gregorics, „Force of Summation,” *Teaching Mathematics and Computer Scinece,* %1. kötet6 (2), pp. 185-199, 2014.

[22] T. Gregorics, „Analogous programming with a template class library,” *Teaching Mathematics and Computer Scinece,* %1. kötet10 (1), pp. 135-152, 2012.

[23] Á. Bede-Fazekas, L. Horváth, A. Trájer és T. Gregorics, „An ArcGIS Tool for Modeling the Climate Envelope with Feed-Forward ANN.,” *Applied Artificial Intelligence,* %1. kötet29 (3), pp. 233-242, 2015.

[24] N. J. Nilsson, Principles of Artificial Intelligence, Springer-Verlag, 1982.

[25] T. Gregorics, „Kereső rendszerek a mesterséges inteligenciában,” in *Futó, I (Ed.): Mesterséges intelligencia, , 1999.*, I. Futó, Szerk., Budapest, Aula, 1999, pp. 26-46.

[26] I. Fekete, T. Gregorics és S. Nagy, Bevezetés a mesterséges intelligenciába, Budapest: LSI, 1990, 1999.

[27] P. Hart, N. J. Nilsson és B. Raphael, „A Formal Basis for the Heuristic Determination of Minimum Cost Paths,” *IEEE Trans. System, Man and Cybernet,* %1. kötet, összesen: %24, (2), pp. 100-107, 1968.

[28] I. Fekete, T. Gregorics és . L. Z. Varga, „Corrections to Graph-Search Algorithms,” in *ELTE*, Budapest, 1988.

[29] R. Detcher és J. Pearl, „Generalized Best-first Search Strategies and Optimality A*,” *Journal of the Association for Computing Machinery,* %1. kötet32(3), pp. 505-536, 1985.

[30] S. J. Russell és P. Norvig, Mesterséges intelligencia. Modern megközelítésben, Budapest: Panem-Prentice Hall Inc, 2000.

[31] A. Martelli, „On the Complexity of Admissible Search Algorithms,” *Artificial Intelligence,* %1. kötet8(1), pp. 1-13, 1977.

[32] T. H. Cormen, C. E. Leiserson, R. L. Rivest és C. Stein, Új algoritmusok.

[33] H. D. Mills, Mathematical functions for structured programming,, Gaithersburg : SC 72-6012, IBM, 1972.

[34] Á. Fóthi and co;, „Workgroup on Relation Models of Programming: Some Concepts of a Relational Model of Programming,” in *ELTE*, Visegrád, 1995.

[35] T. Gregorics, „Object-oriented backtracking,” *Acta Universitatis Sapientiae, Informatica,* %1. kötet9, %1. szám2, pp. 144-161, 2017.

[36] E. Gamma, R. Helm, R. Johnson és J. Vlissides, Programtervezési minták, Budapest: Kiskapu Kft., 2004.