



Eötvös Loránd Tudományegyetem

Informatikai Kar

Információs Rendszerek Tanszék

AXML lekérdezések

Témavezető:

dr. Kiss Attila
tanszékvezető, docens

Szerző:

Pinczel Máté
programtervező informatikus
MSc, 1. évfolyam

Budapest, 2012

A dolgozat elkészítését az ELTE IK, Információs Rendszerek Tanszék
KMS projektlaborja támogatta.



Tartalomjegyzék

BEVEZETÉS.....	3
ALAPFOGALMAK	5
AXML	5
LEKÉRDEZÉSEK	6
MINTÁK	7
MINTA ILLESZTÉSE	9
SZABÁLYOK	10
GAXML.....	11
QAXML.....	12
DQAXML izolált függvényekkel	13
NQAXML izolált függvényekkel	13
DQAXML sűrű beágyazással.....	13
NQAXML sűrű beágyazással.....	14
MEGVALÓSÍTÁS	15
BEHELYETTESÍTHETŐ FELTÉTEL.....	15
MINTA ILLESZTÉSE UTANKÉNT	17
NEGATÍV CSÚCSOK KEZELÉSE	18
UTAK ILLESZTÉSE	18
REDUKÁLTSÁG MEGTARTÁSA.....	21
FÁK TÁROLÁSA	22
DETERMINISZTIKUS FÜGGVÉNYHÍVÁSOK	23
PROGRAM HASZNÁLATA.....	24
Program szolgáltatásai	24
AXML Program szöveges reprezentációja	25
ELEMZÉS	28
ILLESZTÉS MŰVELETIGÉNYE	28
UTAK ÖSSZEKAPCSOLÁSA.....	28
MÉRÉSI EREDMÉNYEK.....	29
KITEKINTÉS	34
KÖSZÖNETNYILVÁNÍTÁS.....	35
IRODALOMJEGYZÉK.....	36

Bevezetés

Vállalati környezetben – és egyéb helyzetekben is – hasznos lehet a gyorsan változó dokumentumokat Active XML (AXML) formájában tárolni. Ezek a dokumentumok nagyon rugalmasan tudnak viselkedni, köszönhetően annak, hogy tartalmazhatnak távoli hívásokat/hivatkozásokat, így elosztottan tudjuk tárolni és szerkeszteni őket. De biztosítani kell azt is, hogy tárolt adatokon lekérdezéseket tudjunk végrehajtani. Erre több más módszer is létezik jelenleg, melyeknek különböző előnyök és hátrányai vannak. Ilyenek például a *XQuery* ([5]) és az *XPath* ([4]), melyek utakat definiálnak a dokumentumfán, és e segítségével nyernek ki információkat a dokumentumból. Egy más fajta megoldást nyújtanak erre a problémára a transzformáló nyelvek, mint például az XSLT (EXtensible Stylesheet Language Transformations, lásd [3]), melyet előszeretettel használnak HTML dokumentumok előállítására XML adatokból. A fent említett nyelvek és módszerek mind kipróbált és jól bevált technológiákon alapulnak, és találhatunk hozzájuk jól működő megvalósításokat. A lekérdezőnyelvek megalkotásánál a fő mozgatórugó eleinte a hatékony kiértékelés, a könnyű kezelhetőség, illetve vizsgálat volt. A jelenlegi kutatások már azt is előtérbe helyezik, hogy megfogalmazható-e az összes kiszámítható lekérdezés, tehát a lekérdező nyelvek kifejezőerejét/teljességét vizsgálják. Olyan nyelv megalkotását tűzték ki célul, amellyel minden kiszámítható lekérdezést meg lehet fogalmazni. Így nyilván kevésbé jó tulajdonságokkal fognak rendelkezni, például hatékonyság szempontjából.

Az egyik ilyen lekérdezőnyelv a QAXML ([1]). Ez az AXML fák egy speciális változatának fogható fel. E nyelvnek több különböző változata létezik, melyek között van olyan, ami eleget tesz az elvárásoknak, azaz a teljességnek, de olyan is akad, mely csak egy jól definiált részhalmazát tudja kifejezni a kiszámítható lekérdezéseknek. Ezek a változatok több jellemzőben térhetnek el, melyek például a következők lehetnek: a nyelv alap építőköveinek kifejezőereje, determinisztikus vagy nemdeterminisztikus megközelítés, illetve a megengedett fastruktúrák. Ennek ellenére a részleges megoldások is hasznosnak bizonyulhatnak a megfelelő környezetben, ahol nincs

szükség teljességre, hiszen korlátozott eszköztárral bíró kifejezéseket hatékonyabban lehet kiértékelni, így a nyelvhez hatékonyabb algoritmust adhatunk. Jelen pillanatban ezekhez a nyelvekhez nem található elérhető implementáció, habár mind hasznosnak bizonyulhatnak, tekintve, hogy a széles körben használható AXML fákból tudnak lekérdezni, amely már bizonyította hasznosságát.

Dolgozatomban bemutatom az AXML-t, és a hozzá tartozó fogalmakat. Áttekintem a QAXML formáit, és az ide tartozó fogalmakat, eredményeket. Kiemelten a QAXML determinisztikus változatával foglalkozom, melyhez egy programot is készítek, ami képes az AXML gráfok kezelésére, és a QAXML nyelven megfogalmazott lekérdezések egy osztályának kiértékelésére.

Alapfogalmak

Az itt felsorolt alapfogalmak (és még több) megtalálhatóak az [1] forrásban.

AXML

Definíció: (*Adatfa*)

Az adatfa egy címkézett, rang nélküli, rendezetlen fa. Legyenek adottak a következő diszjunkt végtelen halmazok:

- $N = \{r, n, m, \dots\}$ a csúcsok halmaza
- $\Sigma = \{a, b, c, \dots\}$ a címkék halmaza
- $D = \{\alpha, \beta, \dots\}$ az adatok halmaza

Továbbá egy véges függvény (*parent*) N felett, a szokott módon, és egy címkéző függvény *label*: $N \rightarrow \Sigma \cup D$ melyre igaz, hogy ha $label(n) \in D$, akkor n levél.

Az adatfákról feltételezzük, hogy *redukáltak*, tehát nincs két olyan testvér részfa, amelyek izomorfak a címkék és adatok megőrzésével. Ez tulajdonképpen egyfajta halmaz megközelítésnek felel meg. Az I fa adatait $dom(I)$ -vel jelöljük.

Definíció: (*AXML fa*)

Egy speciális adatfa, ahol a levél csúcsok, és csakis azok, kaphatnak címkeként függvényszimbólumot is. Legyen $\mathcal{F}$ a függvénynevek egy végtelen halmaza, ekkor $\mathcal{F}^! = \{!f | f \in \mathcal{F}\}$ és $\mathcal{F}^? = \{?f | f \in \mathcal{F}\}$ a függvényszimbólumok halmaza.

Itt a $!f$ egy függvényhívást, $?f$ pedig egy futó függvény visszatérésének helyét jelöli. Működés közben minden $!f$ -t előbb-utóbb felvált egy $?f$, amelyet pedig az eredményt reprezentáló részfa.

Ezeknél a fáknál újra kell értelmeznünk a redukáltság fogalmát, hiszen két részfat, amelyet az adatfa definíciója szerint izomorfnek tekintünk, itt nem tekinthetünk azonosnak, ha azok tartalmaznak egy $?f$ címkéjű csúcsot, azaz futó függvényt. Ez természetes kikötés, hiszen a már meghívott függvények visszatérése különböző lehet

(pl. különböző bemenet miatt, ha nem egyszerre lettek meghívva, ami triviálisan teljesül).

Lekérdezések

Definíció: (*C-genericitás*)

Legyen $C \subseteq D$ véges, ekkor egy adatfák feletti $\mathcal{R}$ reláció C -generikus, ha

$\forall \rho: N \cup D \cup \Sigma \rightarrow N \cup D \cup \Sigma$ -ra, ahol

- ρ injektív,
- $\rho(N) \subseteq N, \rho(D) \subseteq D$ és
- $\rho|_{\Sigma \cup C} = id$,

akkor $(I, J) \in \mathcal{R} \Leftrightarrow (\rho(I), \rho(J)) \in \mathcal{R}$.

Ez tulajdonképpen azt fejezi ki, hogy a C -én kívüli adatoktól és csúcsoktól független az $\mathcal{R}$.

Definíció: (*Kiszámíthatóság*)

$\mathcal{R}$ reláció kiszámítható, ha létezik nemdeterminisztikus $\mathcal{M}_{\mathcal{R}}$ Turing-gép, hogy ha adott egy $\leq$ rendezés az adatokon és $enc_{\leq}(I)$ kódolása I fának, akkor $\mathcal{M}_{\mathcal{R}}$ az $enc_{\leq}(I)$ bemenetre $enc_{\leq}(J)$ -vel a szalagján terminál $\Leftrightarrow (I, J) \in \mathcal{R}$.

Definíció: (*DTD – Document Type Declaration*)

A DTD egy megszorítás az adatfa szerkezetére, amely megadja, hogy egy adott címkéjű csúcsnak milyen címkéjű gyermekei lehetnek, és hány darab. Ezt $|b| \geq k$ formájú kifejezések logikai kombinációjaként írja le, ahol $b \in \Sigma \cup \mathcal{F}^! \cup \mathcal{F}^? \cup \{dom\}, k \in \mathbb{N}$.

Megkülönböztethetünk statikus, illetve aktív DTD-ket a szerint, hogy megenged-e függvény-szimbólumokat, vagy sem.

Definíció: (*Lekérdezés*)

Egy $\mathcal{R}$ reláció lekérdezés, ha a Σ feletti Δ DTD-nek megfelelő adatfákat tekintve $\mathcal{R}$ C -generikus, kiszámítható és $\forall (I, J) \in \mathcal{R}$ -re:

- $dom(J) \subseteq dom(I) \cup C$

- I és J csúcsai diszjunktak

Az utolsó kitétel kicsit összezavaró lehet a determinisztikus lekérdezések értelmezése szempontjából, hiszen egy lekérdezéssel új csúcsokat hozunk létre, amire nem akarunk megkötéseket tenni. Ezt úgy oldjuk meg, hogy a csúcsok neveitől eltekintünk.

Definíció: (*Lekérdezés teljesség*)

Egy nyelv lekérdezés teljes, ha kifejezi az összes kiszámítható, fenti definíció szerint adott lekérdezést.

Ezt a fogalmat értelmezhetjük a determinisztikus és a nemdeterminisztikus lekérdezések felett is.

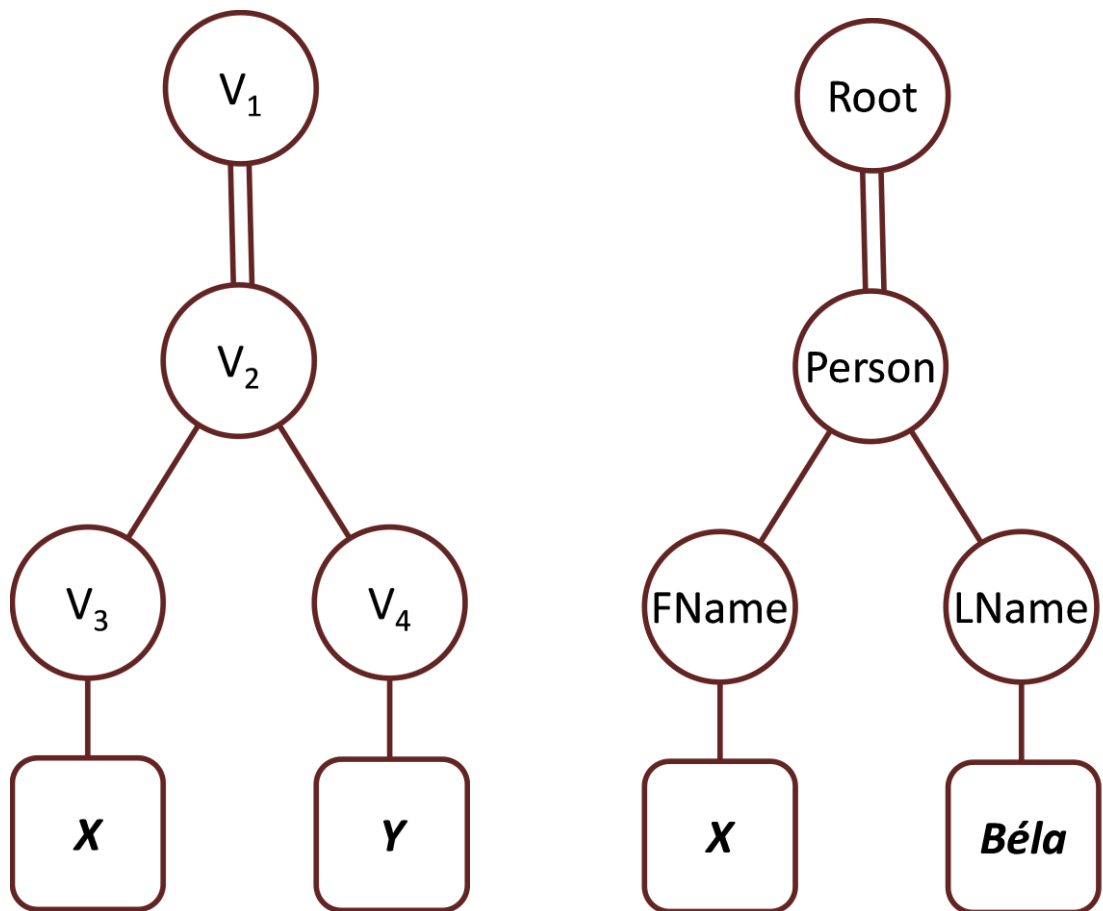
Minták

A lekérdezőnyelvek alap építőköve a mintaillesztés. A P minta egy $(T, cond)$ rendezett pár, ahol T egy faminta, és $cond$ egy feltétel, ami a minta adataira és címkéire ad megkötést.

Pontosabban a T minta egy címkézett fa, aminek a csúcsai változókkal, vagy konstansokkal vannak címkézve, és kétféle élt tartalmazhat: a $|$ a *gyermek* kapcsolatot jelöli, és a $||$ a *leszármazott* kapcsolatot. A minta szemantikájáról bővebben később, előre elég annyi, hogy a leszármazott reláció a gyermek reláció tranzitív-reflexív lezártját jelenti. Továbbá a csúcsok lehetnek pozitív vagy negatív ($\sim$) előjelűek, de a gyökér mindig pozitív. A minta egy csúcsát *határ csúcsnak* nevezzük, ha előjele negatív, vagy ő maga a gyökér. (Például az ábrán a gyökér az egyetlen határ csúcs.)

Egy tetszőleges pozitív csúcsban gyökerező S részfához S^+ legyen az a részfa, amelyet a negatív csúcsokban gyökerező részfák eltávolításával kapunk. Továbbá egy tetszőleges határ csúcsához legyen $var(b)$ a b -ben gyökerező T részfából kapható T^+ fa változóinak és a b felett elhelyezkedő n határ csúcsokhoz tartozó $var(n)$ halmazok uniója. A gyökérhez tartozó $var(r)$ halmaz különösen fontos lesz számunkra, hiszen ezek azok a változók, amelyek értékeit felhasználhatjuk majd a mintaillesztés eredményként.

A *cond* feltétel minden b határcsúcsához rendel egy feltételt, melyet a $var(b)$ változóiból képzett egyenlőségek logikai kombinációjaként adunk meg. Két féle változót különböztetünk meg, az *adatváltozót* és a *strukturális* változót. Az előbbit $X = Y$ vagy $X = d$ formájában írjuk be a feltételbe, ahol X és Y adatváltozókat, d pedig egy adatot jelöl. Az utóbbi pedig $V = t$ alakban jelenik meg, ahol V egy változót, t pedig egy címkét vagy függvényszimbólumot jelent.



$$V_1 = \text{Root} \wedge V_2 = \text{Person} \wedge V_3 = \text{FName} \\ \wedge V_4 = \text{LName} \wedge Y = \text{Béla}$$

1. ábra

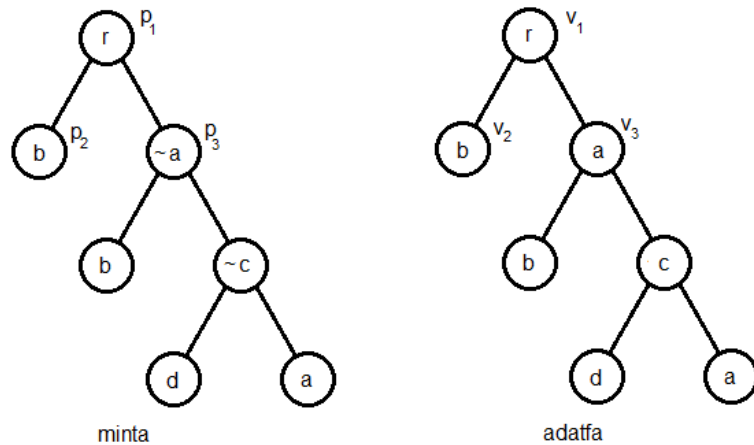
Az 1. ábra bal oldalán egy ilyen T faminta és a hozzá tartozó feltétel látható, míg a jobboldalon ugyanez összevonva. Az összevonást a feltétel(ek) speciális formája miatt tehetjük meg, pontosabban azért, mert azok egyenlőségek konjunkciói.

Minta illesztése

Az imént definiált minták alkalmasak adatok kinyerésére egy fából. Az illesztés eredménye egy $v: var(T^+) \rightarrow N$ leképezés, amely – pongyolán fogalmazva – egyfajta leképezés az adatfa és a minta között, a szerkezet és a *cond* feltételek „megtartásával”. Pontosabban a következő tulajdonságokkal rendelkezik:

1. A leképezés megőrzi a gyermek és leszármazott relációkat, azaz, ha a mintában $N_1 = parent(N_2)$, akkor az adatfában $v(N_1) = parent(v(N_2))$. Illetve $N_1 = ancestor(N_2)$ esetén $\exists n \in \mathbb{N}_0, M_1 \dots M_n$ úgy, hogy
 - $v(N_1) = parent(v(M_1))$,
 - $v(M_1) = parent(v(M_2)), \dots v(M_n) = parent(v(N_2))$ vagy $N_1 = N_2$.
2. $\forall X$ -re, ahol X egy (konstans) adat, $v(X)$ is egy adat.
3. Eleget tesz *cond*(r)-nek, azaz
 - V strukturális változó esetén a $V = t$ -nek eleget tesz, ha a $v(V)$ címkéje t és
 - X adatváltozó esetén $X = Y$ -nek eleget tesz akkor, ha $v(X)$ és $v(Y)$ címkéje megegyezik; és $X = d$ -nek akkor, ha $v(X)$ -hez d adat tartozik.
4. Ha egy tetszőleges negatív n csúcsnak, amely a gyökérhez tartozó T^+ részfa egy levelén gyökerezik, megváltoztatjuk az előjelét pozitívvá, v akkor sem terjeszthető ki erre az új fára az itt felsorolt követelmények betartásával (itt *cond*(n)-nek is teljesülnie kell).

Az első három kitétel természetes és szemléletes. Az utolsó feltétel azt fejezi ki, hogy a maximális negatív gyökerű részmintára illeszkedő részfa nem fordulhat elő az illesztett fában a megadott helyen. De negatív gyökér alatt is előfordulhat újabb negatív gyökér. Ekkor ezt természetes módon értelmezhetjük úgy, mint a megszokott dupla negációt, azaz $A \wedge \neg(B \wedge \neg C) = A \wedge (\neg B \vee C)$ esetén $A = igaz, B = igaz$ csak akkor utasítandó el, ha $C = hamis$, de $C = igaz$ esetén elfogadjuk. Így a 2. ábra mintája illeszkedik a mellette látható adatfára. Hiszen annak ellenére, hogy 'r' alatt megjelenik egy $a(b,c)$ részfa, 'c' alatt találunk egy $c(a,d)$ részfát, ami miatt az $a(b,c)$ részfa nem tud illeszkedni a negatív 'a' gyökerű részmintára, így semmi sem tiltja a $v := \{(v_1, p_1), (v_2, p_2), (v_3, p_3)\}$ választást.

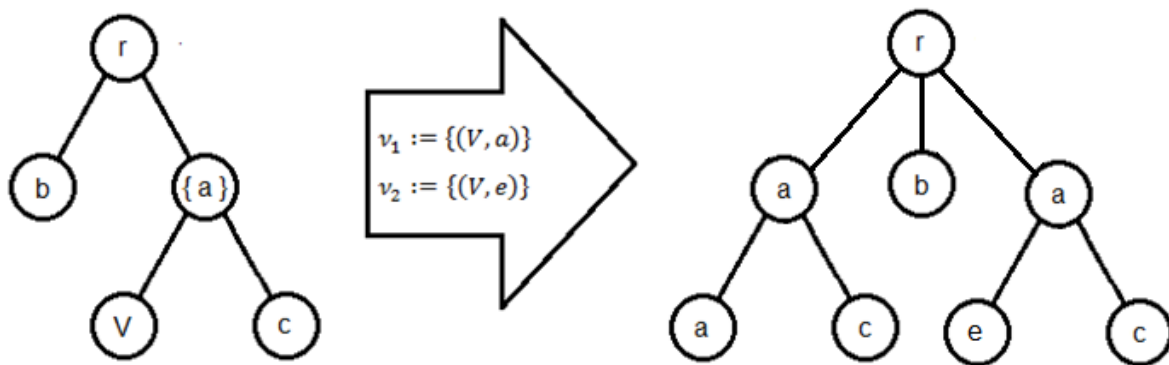


2. ábra

A mintákban előfordulhat egy speciális változó, a *self*, amely egy adott csúcsra tud illeszkedni. Ez tulajdonképpen a egy paraméteres mintaillesztést jelent. Az ilyen mintákat relatív mintáknak nevezzük, és a megadott csúcshoz képest értékeljük ki.

Szabályok

A minta illesztése önmagában nem sok lehetőséget ad a kinyert információ felhasználásra. Ezért szabályok formájában használjuk fel őket. Ezek $Body \rightarrow Head$ formában vannak megfogalmazva (hasonlóan a *Datalog* szabályokhoz), ahol a *Body* egy minta, *Head* pedig egy $Body^+$ változóit is tartalmazó AXML fa. Ebbe a fába fogjuk a kinyert információt behelyettesíteni, azaz a változókat a leképezés szerint behelyettesítjük. Ám a mintaillesztés általában több lehetséges leképezést is ad eredményül. Ezt úgy kezeljük, hogy a $\{ \}$ zárójelek közé írt *konstruktorcsúcsot*



3. ábra

sokszorosítjuk a megfelelő számban, így kapva egy fát, vagy erdőt. (Ha erdőt kapunk, azt befűzhetjük ideiglenesen egy csúcs alá.)

A 3. ábra egy ilyen példát mutat be. Itt két leképezést kaptunk a minta illesztéséből (a mintát nem tüntettem fel), ezek (itt) rendre egy-egy értéket tartalmaznak V -hez, amelyekkel kitöltjük a fát úgy, hogy a konstruktorcsúcsot a saját szintjén sokszorosítjuk. Így kapjuk a jobb oldalon látható fát.

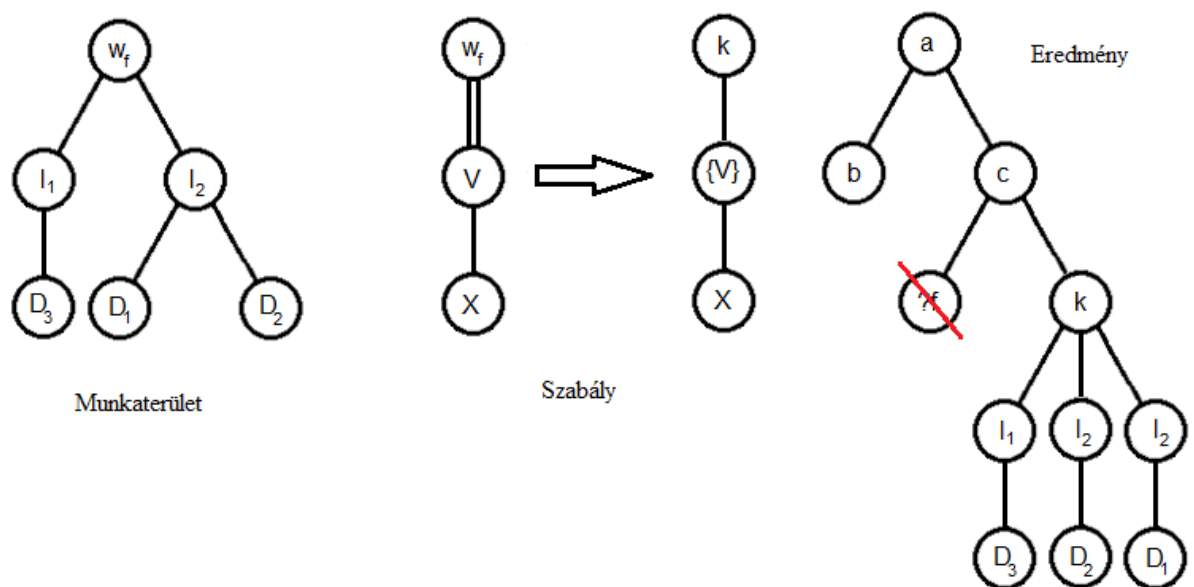
GAXML

A Guarded AXML az AXML egy speciális változata, amely tulajdonképpen egy végrehajtható programot reprezentál, ahol az adatok és a függvények mind az adatfában helyezkednek el. Jelöljük a függvényeket a fent definiált módon, azaz egy szimbólumhalmazból véve, pl. f -et! Ekkor a $!f$ címkéjű csúcs jelöli azt a pontját az adatfának, ahol az f függvényt meghívhatjuk, és $?f$ a már futó függvény visszatérési pontját. Ez a kétféle címke egymás után fordul elő, ebben a sorrendben, egymást váltva.

A függvények rendelkeznek egy belépési őrfeltétellel, amely faminták egy logikai kombinációja. Ezeknél csak az illeszkedés voltára vagyunk kíváncsiak, nem pedig a konkrét leképezésre. Ha ez a feltétel teljesül, akkor létrejön a függvény munkaterülete, ahova kinyeri a neki szükséges adatokat. Ezt *relatív szabályok* (*self* csúcs) egy halmazával teszi meg (*input query*), azaz minden egyes szabály segítségével kibont egy-egy fát az adatfából (lásd a minta illesztésénél). Ezeket a fákat egy w_f csúcs alá gyűjti, így kapva meg a saját munkaterületét, amely szintén egy AXML fa. A munkaterület természetesen tartalmazhat további függvényhívásokat is, melyek itt fognak lefutni, újabb munkaterületeket hozva létre.

A függvény visszatérése hasonlóan zajlik a híváshoz. Egy visszatérési őrfeltételhez van kötve, amely hasonlóan definiálható a belépési őrfeltételhez. Ha ez teljesül, és ha nincsen a munkaterület fájában futó függvényhívás, akkor visszatérhetünk az eredménnyel. Ezt szabályok egy másik halmazával tesszük meg, amelyeket a w_f -ben gyökerező fára illesztünk, és eredményét a $?f$ címkéjű csúcs testvéreiként helyezzük el az adatfába, végül eltávolítjuk $?f$ -et.

Egy program *terminált*, ha nem tartalmaz futó vagy hívható függvényekkel címkézett csúcsokat.



4. ábra

Ezt a folyamatot mutatja be a 4. ábra, ahol az adatfa kezdetben az a, b, c és $?f$ csúcsokból állt. Az input query-t itt sem írtam fel, az most lényegtelen.

QAXML

A QAXML egy GAXML program, mely egy lekérdezést hajt végre. Itt a lekérdezés eredménye az eredeti fába kerül egy *output* címkéjű új csúcs alá, amelyet a gyökér gyermekeként hozunk létre.

Több változata létezik, melyek mind függvényhívásokat használnak, csupán a megengedett „konstrukciókban” és a végrehajtás módjában különböznek. A végrehajtás módja szerint két kategóriába sorolhatjuk az egyes változatokat, ezek a

determinisztikus és a nemdeterminisztikus lekérdezőnyelvek. A *determinisztikus* végrehajtás folyamán az összes függvényt szimultán hívjuk meg, amelyek belépési őrfeltétele teljesül, és hasonlóan minden függvény egyszerre tér vissza, melyeknek visszatérési őrfeltétele igaz. A *nemdeterminisztikus* esetben választunk egy függvényt, amely meghívható vagy visszatérhet, és csak ezt dolgozzuk fel.

Az egyes QAXML nyelvek rokonságot mutatnak a *while*-nyelvcsalád tagjaival. Ezek fontosak a kifejezőerejük vizsgálatához, amit az [1] cikk részletez. Itt csak a fontosabb eredményeket tekintem át, melyek szükségesek a QAXML létjogosultságának belátásához.

DQAXML izolált függvényekkel

Determinisztikus végrehajtású QAXML lekérdezőnyelv, mely a függvényeknek csak triviális beágyazását engedi meg. Ez azt jelenti, hogy a kezdeti fában csakis a gyökér gyermekeként fordulhat elő, pontosan egy függvényt tartalmazó csúcs. A szabályok fejrészeiben is csak a konstruktor csúcs felett fordulhatnak elő függvények. Ez azt jelenti, hogy nem áraszthatjuk el a fát hívásokkal, mert minden lépésben pontosan annyi függvény kerülhet bele a fába, ahány a szabályokban előfordul, függetlenül az illeszkedések számától.

NQAXML izolált függvényekkel

A DAXML izolált függvényekkel használt változatával megegyezik, kivéve abban, hogy nemdeterminisztikus a végrehajtása.

Tétel: A DQAXML izolált függvényekkel pontosan a NQAXML izolált függvényekkel determinisztikus részét lehet kifejezni.

Tétel: Sem a DQAXML sem a NQAXML izolált függvényekkel *nem* lekérdezés teljes.

DQAXML sűrű beágyazással

Determinisztikus végrehajtású QAXML lekérdezések, melyek a függvények sűrű beágyazását is megengedik. Ez azt jelenti, hogy akár minden Σ elemével címkézett csúcs alatt szerepelhet függvényhívás.

NQAXML sűrű beágyazással

A DQAXML sűrű beágyazással használt változatával egyezik meg, de végrehajtása nemdeterminisztikus.

Tétel: A NQAXML sűrű beágyazással lekérdezés teljes.

Tétel: A DQAXML sűrű beágyazással *nem* lekérdezés teljes.

Mindegyik fent említett lekérdező nyelvhez létezik olyan változat, mely olyan változókat használ, amik egész részfákat tudnak megragadni. Ezek szintén definiálva vannak az [1]-ben.

Megvalósítás

A mellékelt programban a DQAXML lekérdezések egy részének végrehajtását valósítottam meg. Tehát a megfogalmazott programok függvényhívásai determinisztikus végrehajtás mellett értékelődnek ki, és sűrűn lehet őket beágyazni. Természetesen ez tartalmazza az izolált függvények esetét is.

A program .Net környezetben íródott C# nyelven. A megoldás osztályait és típusait egy osztálykönyvtár tartalmazza.

A megoldás folyamán a minták leképezéseit relációként kezelem. Ehhez feltehetjük, hogy minden csúcshoz tartozik egy egyedi azonosító (pl. a neve), amely alapján el lehet tárolni a hozzá tartozó adatcsúcsot az adott relációban. Így a reláció sémája a minta csúcsainak neveiből fog állni. Lentebb kiderül, hogy ez nem lesz elég a feladat megoldásához, így újabb oszlopok kerülnek majd be a táblákba.

Behelyettesíthető feltétel

Hívjuk behelyettesíthető feltételnek azokat a formulákat, amelyek csak egyenlőségek konjunkcióiból állnak. Ezek nem tartalmazhatnak olyan feltételeket, mint $V \neq t$ vagy $V \neq W$. Ennek az a haszna, hogy be tudjuk őket írni a fába. Továbbá az illesztés bármelyik pontján, ha egy változót tartalmazó csúcsot illesztünk, akkor el tudjuk dönteni, a változó értéke alapján, hogy megengedett-e az a pozíció. Ha pedig a változónak nincs értéke, akkor egyértelmű értéket kap.

Fentebb láthattunk példát arra, hogy hogyan néz ki egy behelyettesített faminta (1. ábra), és láthattuk a definíciót. Feltűnhet, hogy a definíció jóval általánosabb, hiszen ott a csúcsok változókkal vannak címkézve, és ezekre adunk logikai feltételeket, amelyeknek (többek között) teljesülnie kell (vagy éppen nem szabad teljesülnie) ahhoz, hogy a leképezést illeszkedésnek mondhassuk. Ezek a feltételek egyenlőségek logikai kombinációi, amelyekben tagadás, és implikáció is szerepelhet. Át tudjuk őket alakítani diszjunktív normálformára, azaz

$$\bigvee_i \bigwedge_j [\neg] Eq_{ij}$$

alakra, ahol Eq_{ij} -k a már fent említett formában megfogalmazott egyenlőségeket takarnak. Tehát vagy az illesztett részfa csúcsainak címkéire vonatkoznak, vagy két változó adott interpretációban felvett értéke közötti egyenlőségeket jelentenek. Az interpretációt az aktuálisan kiválasztott hasonló szerkezetű (lásd Minta illesztése részben az 1-2 feltételek) részfa, azaz a leképezés adja.

A megfelelő leképezéseket nyilván csak azok az interpretációk adhatják, amelyekkel $\bigwedge_j [\neg]Eq_{*j}$ teljesül, legalább egy $*$ -ra ahol $*$ tetszőleges értéket jelöl. Továbbá az illesztés 4. feltétele alapján, a „rákövetkező” határcsúcsokhoz tartozó feltételek diszjunktív normálformájából egyetlen tagnak sem szabad teljesülnie, hiszen ez pont azt jelentené, hogy kiterjeszthető. És ha mélyebben is helyezkednek el határcsúcsok, akkor azok feltételeit is meg kell vizsgálnunk. (Hiszen a feltétel is rekurzív módon van megadva.) Láthatjuk, hogy így egyetlen minta illesztését több illesztésre vezettük vissza, amelyektől azt várjuk, hogy egyszerűbben lehet őket kiértékelni.

Ha megvannak az így kapott minták eredményei, akkor relációs algebrai műveletekkel meg tudjuk kapni a tényleges végeredményt. De ezekben a konjunktív feltételekben az egyenlőségek nem csak pozitív formában szerepelhetnek, hanem negálva lehetnek. Ez azért jelenthet problémát, mert az egy csúcsra vonatkozó atomok igazságértékének eldöntéséhez nem elég az adott változó értékét ismerni. Mondhatnánk, hogy ez fennáll a nem negált atomokra is, de azokat el tudjuk majd tüntetni, ahogy nemsokára kiderül.

Tehát az általános feltétel nem tűnik túl alkalmasnak a kiértékeléshez. Ezen okok miatt csak a behelyettesíthető feltételeket engedem meg. Itt egyenlőségeket és változókat is eliminálhatunk, mégpedig az $X = Y, V = W$ formájú atomok mentén, melyek változók közötti egyenlőséget írnak elő. Ilyen esetben az egyik változó összes előfordulását helyettesíthetjük a másik változóval, és elhagyhatjuk az egyenlőtlenséget. Így olyan formulát kapunk, amely csak konstanssal való egyenlőséget ír elő. Ez alkalmas arra, hogy egy mintacsúcsról rögtön eldöntsük, illeszkedhet-e egy adatcsúcsra, ha ismerjük az eddig illesztett csúcsok pozícióit.

A programban jelen pillanatban az ilyen típusú feltételek megengedettek. Ezt az erős korlátozást a probléma számításelméleti komplexitása indokolja, ugyanis [2] megmutatja, hogy a minták illesztése általános esetben NP-teljes probléma.

Minta illesztése utanként

A minta illesztése folyamán egyfajta leképezést keresünk a minta csúcsairól az adatfa csúcsaira. Ez egy bonyolult feladatnak tűnik, ezért dekomponáljuk. Illesszük a minta leveleitől a gyökeréig terjedő utakat az adatfára, mégpedig annak hasonló módon kapható útjaira. Így jóval több, kisebb, de egyszerűbb feladatokat kapunk. Ezeknek az illesztéseknek az eredményeit relációkban tároljuk el, tehát a minta minden leveléhez egy tábla fog tartozni, amelynek oszlopai az adott úton található csúcsok a mintában, értékei pedig az egyes csúcsok adatfában lévő pozíciói. Ekkor a leképezések lehetséges töredékeit kapjuk meg, azaz a keresendő eredmény reláció vetítéseit a minta útjának csúcsaira. Ezek a töredékek tartalmazhatnak olyan sorokat, amelyeket nem fogunk tudni felhasználni, de nem veszünk el egy hasznosat sem.

Ezek kombinálásával, azaz természetes összekapcsolásával, megkapjuk az összes lehetséges leképezést, hiszen ezzel gyakorlatilag az utakat kapcsoljuk össze egy fává. Tekintsük az $n_1, n_2, \dots, n_k, m_1, \dots, m_i$ és $n_1, n_2, \dots, n_k, o_1, \dots, o_j$ utakhoz adott relációkat. Itt az első k csúcs közös, de a többi diszjunkt, tekintve, hogy fáról van szó. Így a közös attribútumok pontosan $n_1, n_2, \dots, n_k$ lesznek, így ezeken fogjuk összekapcsolni őket. Tehát az eredmény relációban azon sorok összeillesztései fognak megjelenni, amelyekben az $n_1, n_2, \dots, n_k$ csúcsok értékei azonosak, vagyis ugyanoda képződtek le az adatfában. Könnyen látható, hogy ez pont azt jelenti, hogy megtartják a fa szerkezetét.

Ez akkor sem fog sérülni, ha minden relációt összekapcsolunk. Ekkor az attribútumok pontosan a fa csúcsai lesznek, hiszen minden csúcsot elérünk egy úton, és nyilván csak a fa csúcsai lesznek az utakon. A leszármazott relációkat is megőrzi az összekapcsolások, ha ezt az utakon értelmezett leképezések teljesítették.

Hamar kiderül, hogy ez így még nem elég, hiszen a szerkezet megőrzésén kívül a változók értékeinek is meg kell egyeznie. Ezeket pedig a változót tartalmazó csúcsok adatfabeli pozíciója határozza meg. Hiába teljesül az adott utakon, hogy az ugyanazon

változót tartalmazó csúcsok ugyanazt az értéket rendelik a változóhoz, ez az összekapcsolás folyamán nem fog teljesülni, hiszen ezeket lokálisan határoztuk meg, és nem figyeltünk az összekapcsolásnál ezek egyenlőségére. Ennek kijavítására az egyes utak relációiban eltároljuk az előforduló változókat is, és reláció soraiban a csúcsok helyzete által meghatározott értéket. Az összekapcsolásnál ezen értékeken is egyezniük kell a soroknak. Így az eredmény tábla sorai eleget fognak tenni a feltételeknek. (ábra!)

Negatív csúcsok kezelése

A negatív csúcsok azt akarják kifejezni, hogy az alattuk gyökerező részére nem szabad kiterjeszthetőnek lenni a reláció egy sorának sem. Azaz egy leképezés csak akkor legális, ha nincs olyan kiterjesztése, ami egy legfelül elhelyezkedő negatív csúcs megváltoztatásával teljesíti a feltételeket. Ezért az illeszkedést a következő módon kell kiszámítani:

Jelölje $Neg(r)$ az r alatt található azon negatív csúcsokat, melyektől r felé haladva nem ütközünk másik negatív csúcsba.

1. Vegyük S^+ -t, ahol S az r -ben gyökerező fa, tehát az eredeti minta!
2. Számítsuk ki az S^+ -hoz tartozó $R(v_1, v_2, \dots, v_n)$ relációt!
3. Számítsuk ki $Neg(r)$ -t!
4. $\forall n \in Neg(r)$ -re, ha van ilyen
 - 4.1. Változtassuk meg n előjelét pozitívrá
 - 4.2. Számítsuk ki az így kapott fához tartozó Q relációt ezzel az algoritmussal
 - 4.3. $R := R \setminus \pi_{v_1, v_2, \dots, v_n}(Q)$

Utak illesztése

Az utak illesztése a levéltől a gyökér felé haladva történik. Ezt több tényező is indokoltá teszi:

- A haladási irány felfelé egyértelmű a fában.
- Az adatfa definíciójából következik, hogy ha a minta aljában adatváltozót, vagy konstans adatot találunk, akkor ez a csúcs csak az adatfa leveléhez illeszkedhet. Ha nem illeszkednek, akkor nem kell folytatnunk az illesztést.

Az illesztés folyamán több objektum dolgozik össze. Az *egyikük* (**FitManager**) felügyeli, az illesztések legalitását, azaz a változók értékeit. Ezt két leképezés segítségével teszi. Az első számon tartja azon változókat, melyek rendelkeznek értékkel és a hozzájuk tartozó értékeket. A másiból megtudhatjuk, hogy egy változót a minta mely csúcsai kötöttek le a jelenlegi értékével. Ez az objektum három fő szolgáltatást nyújt:

1. Képes eldönteni a minta egy csúcsáról, hogy az illeszkedhet-e egy adatcsúcsához. Ezt úgy teszi, hogy ellenőrzi, vajon a változónak van-e már értéke, és ha van, az egyezik-e az adatcsúcs értékével.
2. Képes regisztrálni ezeket az illeszkedéseket. Azaz bejegyzi a minta csúcsát, leköti a változót, és ha ő az első aki leköti, akkor bejegyzi az értékét is.
3. Képes törölni a regisztrációkat. Ezt a visszalépésnél kell használni, hogy más illeszkedést találhassunk.

Egy *másik* egyed (**StepManager**) fogja megvalósítani a léptetést. Ez tartja számon a csúcsok pozícióit, és azok állapotát. Ez a tevékenység több szabály alapján lett optimalizálva:

1. Bontsuk a minta útját olyan szakaszokra, amelyek szülő relációval kapcsolódnak egymáshoz. Ha e szakaszok végpontját lerögzítjük, az minden pontjának egyértelműen meghatározza a helyzetét. Így elég csak e szakaszok kezdőpontjait számon tartanunk, hiszen nem tudjuk őket külön léptetni.
2. Az adatfában minden csúcsához ismerjük a gyökértől mért pontos távolságát. A mintában pedig adhatunk minden csúcsához egy minimális távolságot a gyökértől. Ez utóbbi legyen a gyökérig vezető úton található szülő kapcsolatok száma. Ekkor használhatjuk a $mintacsúcs.mintáv \leq adatcsúcs.táv$ egyenlőtlenséget vágási feltételként, ugyanis ha ez nem teljesül, akkor már nem fér el a minta az úton.
3. A szétvágott szakaszokat úgy kell elhelyezni a fában, hogy azok megtartsák a relatív helyzetüket, azaz két szakasz nem lehet fordított sorrendben a faminta útján, mint az adatfa útján.

4. A következő szakasz eleje illeszkedhet arra az adatcsúcsra, ahova az előző szakasz vége is, de előbbre nem.

Természetesen az illeszkedést is szakaszonként kell kezelnünk, ha a léptetést is így oldjuk meg.

A minták útjait egyfajta visszalépéses kereséssel illeszttem az adatfa útjaira. Az itt felsorolt két objektum megvalósítja ezt a módszert, csupán a léptetés műveletét kell meghívni, majd ellenőrizni az illeszkedés tényét. Amennyiben illeszkedett, már el is tárolhatjuk a csúcsok pozícióit és a változók értékeit a megfelelő relációban.

A léptetés minden alkalommal az előzőtől különböző állapotot állít elő, és visszatér egy logikai értékkel, ami azt jelzi, hogy sikerült-e a művelet. Ha ez hamis, az azt jelenti, hogy nincs több lehetséges variáció, így befejezhetjük az illesztést. Az algoritmus elején előállítjuk a mintából a szakaszokat, amelyek belül csak szülő éllel kapcsolódnak. Ezek legmélyebb csúcsát eltároljuk, hiszen ezeknek fogjuk megkeresni a helyét az adatfában. Ez után inicializáljuk az ezekhez számon tartott adatokat, melyek a következők:

- egy logikai érték, amely igaz, ha a csúcs jelenlegi pozíciója illeszkedik a megadott feltételhez,
- egy állapotváltozó, amely 3 féle értéket vehet fel: léptethető, nem léptethető, inicializálatlan.

Számon tartjuk még azt a szakaszt is, amelynek éppen helyet keresünk. A léptetés a következő módon zajlik:

- Ha az aktuális szakasz inicializálatlan, akkor az őt a mintában megelőző szakasz utolsó csúcsának pozíciójára próbáljuk illeszteni. Ha illeszkedik, akkor regisztráljuk, hogy a változók megkapják a szükséges értékeket, és a következő szakaszt inicializálatlanra állítjuk, majd rálépünk, hogy a következő iterációban ő jöjjön. Természetesen a most illesztett szakasz állapotváltozóját is frissíteni kell a szerint, hogy lehet-e még őt léptetni. Itt használom a mélységre vonatkozó vágási feltételt.

- Ha az aktuális szakasz léptethető, akkor az adatfabeli pozíciója legyen a mostani pozíciójának szülőcsúcsa. Ha előzőleg illeszkedett, akkor ezt vissza kell vonni. A továbbiakban az előző ponthoz hasonlóan kell beállítani az állapotváltozókat, és regisztrálni, ha szükséges.
- Ha az aktuális szakasz nem léptethető, akkor inicializálatlanra állítjuk, és az őt megelőző szakaszra lépünk vissza.

Ez hasonlít a visszalépéses keresésre, de nem próbálja meg illeszteni az eleve rossz eseteket, hiszen az inicializálatlan szakaszokat a sorrend megtartásával kezdi illeszteni.

Redukáltság megtartása

Fontos feladatot jelent az adatfák redukált tulajdonságának megőrzése. Minden csúcs/részfa beszúrásánál ellenőriznünk kell, nem tartalmaz-e már a szülő csúcs megegyező gyermeket. Ezt az ellenőrzést teljes mélységig végre kell hajtani. A megvalósítás folyamán fontosnak tartottam, hogy működjön a program mindenféle címke és adattípusokra. Így sablonosan valósítottam meg az osztálykönyvtárat, melyre nem kötöttem ki a rendezés létezését, hiszen ezt nem felelne meg a definíciónak. Ebből adódóan még rendezetten sem lehet tárolni az egyes csúcsok gyermekeit, pedig ez jelentősen lerövidítené az ellenőrzés műveletigényét. Viszont rendezés nélkül minden testvérjelöltre ellenőrizni kell azt, hogy egyeznek-e az újonnan beszúrandó részfával, ráadásul ezt teljes mélységig végre kell hajtani. Az egyes szinten lévő gyermekek száma most legyen rögzített n . Ekkor két részfa azonosságának ellenőrzéséhez $n!$ variációban tudjuk minden szinten összepárosítani a részfákat és e szerint kell minden párosításnál eldöntenünk az egyenlőséget ugyanilyen módon. Így nemkívánatosan hosszan tartana a redukáltság megőrzése. Ezért első körben rendeljünk hozzá a csúcsok címkéihez egy hasító értéket! A csúcsok képezzék a saját hasító értéküket a következő módon:

- ha levél, akkor a címkéje hasító értékét veszi fel, legyen az függvény, adat, vagy címke;
- ha nem levél, akkor a gyermekei hasító értékéből képezi a sajátját, olyan módon, ami nem érzékeny a csúcsok átrendezésére, tehát kommutatív.

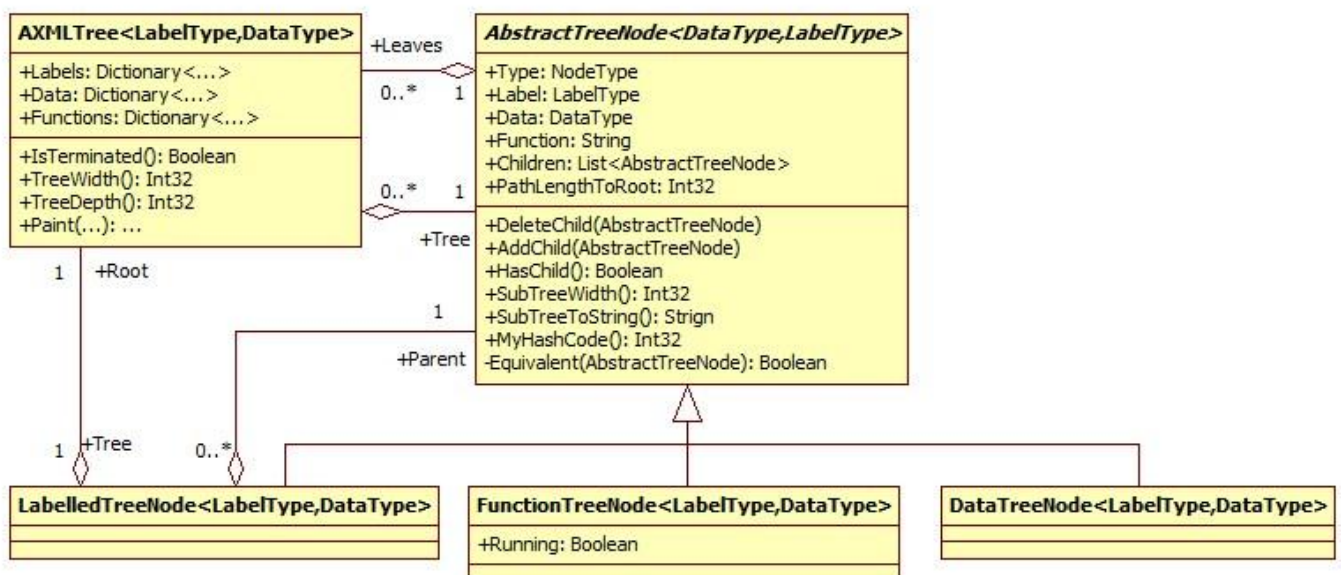
Érdekes ügyelni arra is, hogyha például az adat és a címke típusa megegyezik, és egy konkrét címke és adat értéke ugyanaz, általában akkor se legyen ugyanaz a hozzájuk tartozó hasító érték.

Így elérjük, hogy a különböző részfákból képzett hasító értékek várhatóan különbözzenek, akkor is, ha csak a csúcsok típusa más. A program a „bitenkénti kizáró vagy” műveletet használja a nem levél csúcsok hasító értékének előállítására. Levél szinten pedig a címke/adat típus alapértelmezett hasító értékét módosítottam különböző módokon.

Sajnos, ha két részfa hasító értéke megegyezik, akkor továbbra is ellenőriznünk kell a költséges módszerrel az egyezést. De ezt is gyorsíthatjuk pár szabállyal:

- Ha különbözik a csúcsok típusa, akkor a részfák is különböznek.
- Ha különbözik a csúcsok címkéje, akkor a részfák is különböznek
- Ha különbözik a gyermekeik száma, akkor a részfák is különböznek

Fák tárolása



5. ábra

A fák tárolását az illesztés hatékonyságának szem előtt tartásával tároljuk el.

A különböző típusú csúcsokat egy közös ősből származtatjuk, amely megvalósítja az alapvető követelményeket egy csúccsal szemben. A követelmények a teljesség igénye nélkül:

- a gyökértől való távolság eltárolása
- művelet a csúcs tartalmának lekérdezésére
- gyermekek eltárolása
- szülő eltárolása: ez mindig címkézett csúcs (lásd adatfa definíciója)
- gyermek hozzáadása/eltávolítása
- hasító érték számítása
- ekvivalencia ellenőrzése

A fa eltárolja a saját gyökerét, ami elégséges is lenne, ám sok helyen van szükségünk a levelek bejárására, vagy a függvénycsúcsok elérésére, ezért ezeket is számon tartjuk egy-egy szótárban, hogy ne kelljen bejárnunk az egész fát.

Az egész osztály általánosan lett van megvalósítva, tehát mi szabhatjuk meg az adat és a címke típusát, így az alkalmas komplex adatszerkezetek tárolására is. Az adatfa UML osztály-diagramot az 5. ábra mutatja. Mivel a program C# nyelven lett implementálva .Net környezetben, ezért a láthatóságok pongyolán vannak jelölve, hiszen az UML nem tartalmaz jelölésmódot a különleges láthatóságok jelölésére. Ilyen például az **internal**, vagy az, ha egy adattag lekérdezésének és beállításának különbözik a láthatósága.

A program használ még egy beolvasó osztályt, ami egy adatfolyamból ki tudja olvasni az ott reprezentált fát az elvárt formátumban. Ehhez szüksége van két függvényre, melyek képesek beolvasni szöveges formátumból a címke, illetve az adat típusait.

A többi fa (mintafa, szabályok feje) is hasonló formátumban van eltárolva.

Determinisztikus függvényhívások

A program a determinisztikus függvényhívást valósítja meg, ahol egyszerre kellene meghívni a hívható függvényeket, és visszatérni azokkal, amelyekkel lehet. A kódban

látható, hogy ezt két lépésben teszem meg. Először rendre meghívom a meghívható függvényeket, egymás után, egy ciklusban. Utána a visszatérésre váró függvények eredményeit írom vissza az adatfába, szintén egymás után. Ez nem szimultán, mégis megegyezik annak eredményével, hiszen teljesülni fog az, hogy mindegyik függvény ugyanazt fogja látni a meghívása pillanatában, mint amit az elején láthatott volna. Ugyanis a meghívott függvények az adatfából csomagolnak ki adatokat, és a saját munkaterületükre írják őket, de az eredeti fán nem hajtanak végre módosítást. A visszatérő függvények pedig csak a saját munkaterületükről bontanak ki információt, amelyet az adatfába írnak vissza, ám nem törődnek annak tartalmával, pusztán a redukáltság megőrzése céljából.

Program használata

Program szolgáltatásai

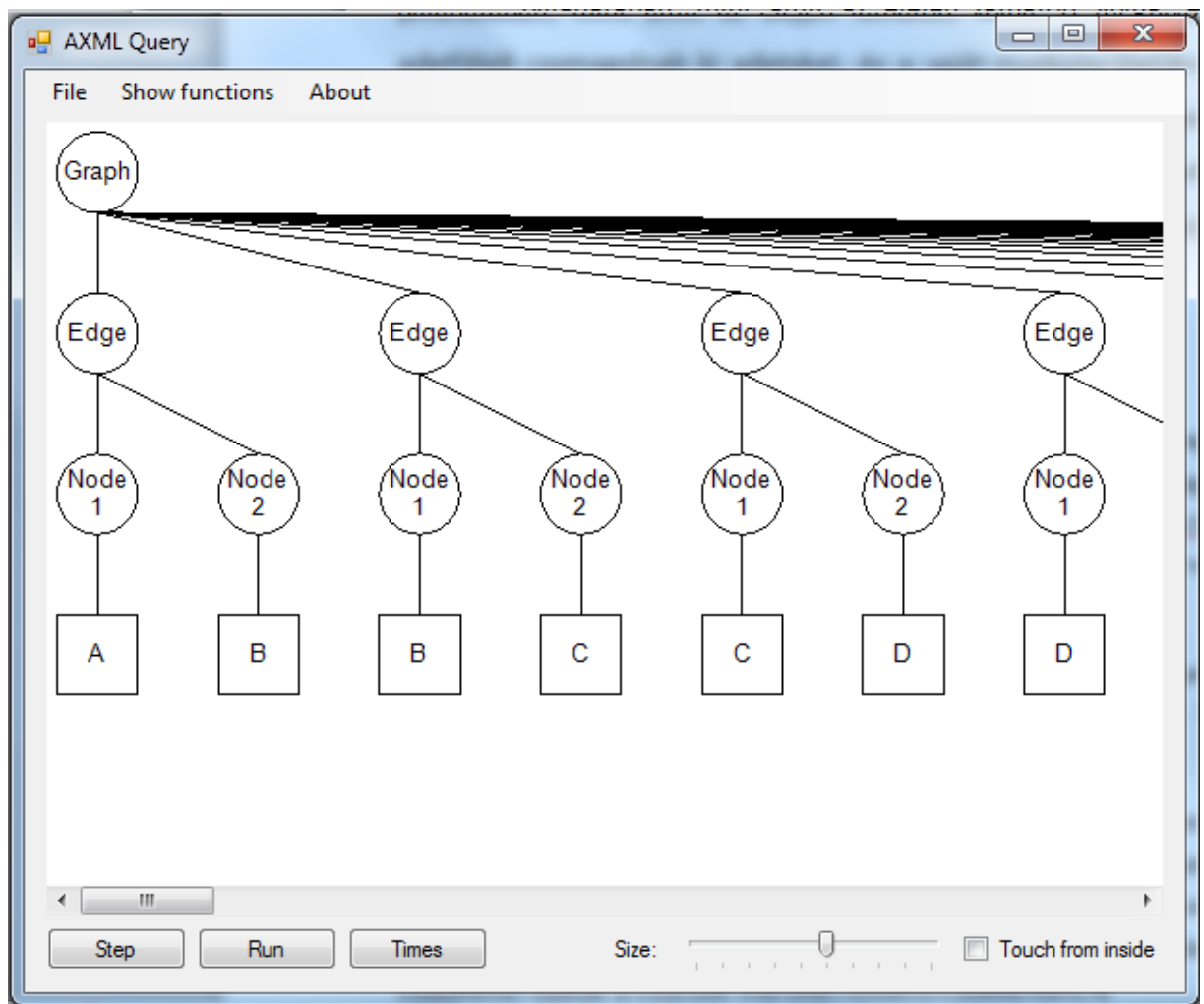
A program képes beolvasni egy QAXML programot, mely egy lekérdezést valósít meg. Ezt a „File” menü „Open program” menüpontjában lehet megtenni. Ez után meg tudjuk tekinteni a program által tartalmazott szabályokat a „Show functions” menüpont alatt. Itt láthatjuk felsorolva az összes függvény összes szabályát, melyek valamelyikére kattintva egy külön ablakban kirajzolódik a *Body* és *Head* fája.

A programot lépésenként tudjuk végrehajtani a „Step” gombra kattintva, mindaddig, amíg az adatfa nem statikus; de akár az egészet lefuttathatjuk, ha a „Run” gombra nyomunk. Minden lépés (futtatás) végeredményét láthatjuk a rajzterületen.

A rajzterületen az adatfa van kirajzolva. Alapértelmezetten akkora méretben, hogy az elférjen. Ez hamar olyan kicsivé zsugoríthatja a csúcsokat, hogy olvashatatlan lesz a tartalmuk, vagy akár a fa struktúrája. Ebben az esetben érdemes átváltani olyan nézetre, ahol megadható a csúcsok mérete. Ezt „Touch from inside” előtti pipa eltávolításával tehetjük meg. Ekkor a mellette található csúszkával tudjuk a csúcsok méretét növelni/csökkenteni. (6. ábra)

A szabályok illesztésének idejét a „Times” gombra kattintva lehet megtekinteni. Itt az idő milliszekundumban van megadva, továbbá láthatjuk az adatfa szélességét és mélységét is.

A „File” menüben bármikor elmenthetjük a fa aktuális kinézetét képként, vagy szöveges formátumban.



6. ábra

AXML Program szöveges reprezentációja

Egy program 5 fő részből áll, amelyeket pontosvesszővel elválasztva kell megadni:

- AXML fa
- függvények
 - belépési őrfeltétel (jelenlegi verzióban üresen kell hagyni)
 - visszatérési őrfeltétel (jelenlegi verzióban üresen kell hagyni)
 - input query-k
 - output query-k

A pontos definíció:

```

Program ::= AXMLTree ';' Functions

Functions ::= Function ';' Functions | ε

Function ::= InputGuards ';' ReturnGuards ';' InputQuerys ';'
           ReturnQuerys ';'

InputGuards ::= ε (jelen verzióban nincs benne)

ReturnGuards ::= ε (jelen verzióban nincs benne)

AXMLTree ::= Node

Node ::= NodeContent '(' Children ')' | NodeContent

Children ::= Nodes

Nodes ::= Node ',' Nodes | Node

NodeContent ::= Label | Data | Function

Function ::= '{' String '}'

Data ::= '[' String ']'

Label ::= String

String ::= ['A'-'Z' 'a'-'z' '0'-'9']*

InputQuerys ::= InputQuerys '|' InputQuery | InputQuery

InputQuery ::= Body '=>' Head

OutputQuerys ::= OutputQuerys '|' OutputQuery | OutputQuery

OutputQuery ::= Body '=>' Head

Body ::= PRoot '(' PChildren ')'

PatternNode ::= EdgeType PNodeContent '(' PChildren ')' |
               PNodeContent

EdgeType ::= '-' | '='

```

```

PChildren ::= PNodes

PNodes ::= PatternNode ', ' PNodes

PNodeContent ::= Variable | Constant

Constant ::= ''' NodeContent '''

Variable ::= StructVar | DataVar

Head ::= HNode '(' HChildren ')'

HNode' ::= PNodeContent '(' HChildren ')' | PNodeContent

HChildren ::= HNodes

HNodes ::= HNode ', ' HNodes | HNode

```

Elemzés

A program sokféle lekérdezés futtatására képes, melyek különböző bonyolultságúak lehetnek számításelméleti szempontból. Így a lekérdezések futási idejét nincs értelme vizsgálni, sokkal inkább a minta illesztését.

Illesztés műveletigénye

Vegyünk egy P mintát, mely behelyettesíthető feltétellel van adva. Legyen a leveleinek száma w_P és legyen a maximális mélysége d_P ! Vegyünk egy F adatfát is, melynek w_F darab levele van és d_F a maximális mélysége. Ekkor minden útját a mintának illesztjük minden útjára az adatfának, tehát összesen $w_F * w_P$ illesztést végzünk el. Az utak illesztésének műveletigénye nagyban függ a minta útján lévő szakaszok számától (azaz a leszármazott kapcsolatok számától). Álljon az út k szakaszból, ekkor minden lehelyezésnél a szakaszoknak $d_F - d_P$ csúcs marad szabadon. Ha most szeretnénk kiszámolni a lehetséges lehelyezéseit a szakaszoknak, akkor tulajdonképpen a szakaszokat kell elhelyezni a szabad csúcsok között. Ezt $\binom{d_F - d_P + k}{k}$ féle képen lehet megtenni, tehát maximum ennyi leképezést tudunk találni. Ezt magyarázza a fentebb említett NP-teljessége a problémának, és a [6] forrásban található tétel, amely a nemdeterminisztikus automaták szimulálásáról szól determinisztikus automatákkal, és ezek műveletigényéről. Ez azt mondja ki, hogy ha nemdeterminisztikus automatával $f(n)$ időben oldható meg egy probléma, akkor ezt $2^{f(n)}$ időben lehet megoldani egy vele ekvivalens determinisztikus automatával. Ez nemkívánatosan nagy műveletigény, de az algoritmus valójában jelentősen gyorsabb lehet, hiszen ha egy szakasz illeszkedését az alsó szinten zárjuk ki, akkor nem is állítjuk elő az ezen illeszkedést tartalmazó leképezéseket.

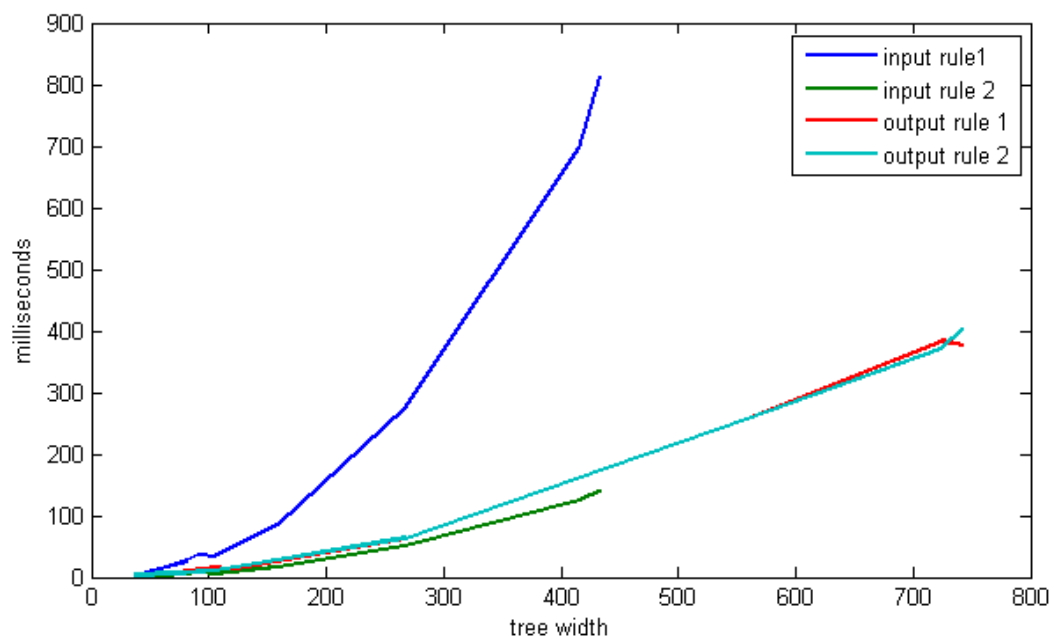
Ezért mérésekkel vizsgálom meg a program viselkedését.

Utak összekapcsolása

Az utak leképezéseit relációkban tárolom. Ahány levele van a mintának, annyi reláció jön létre, amelyeket utána össze kell kapcsolni. Ez jelentősen megnövelheti a műveletigényét egy szabály kiértékelésének, hiszen a relációk sorai nem rendezettek. A

minél gyorsabb kiértékelés érdekében próbálom mindig a legnagyobb szelektivitású összekapcsolást végrehajtani. (Az erre vonatkozó technikákról a [8]–ban található bővebb leírás.) Ez sajnos nem egy egyértelmű feladat, hiszen a mintáról semmi speciálisat nem tudunk. Így a legtöbb közös attribútummal rendelkező táblákat kapcsolom össze, remélve, hogy ezzel elérem a kívánt célt. Ez a helytakarékoság szempontjából is előnyös megközelítés.

Mérési eredmények



7. ábra

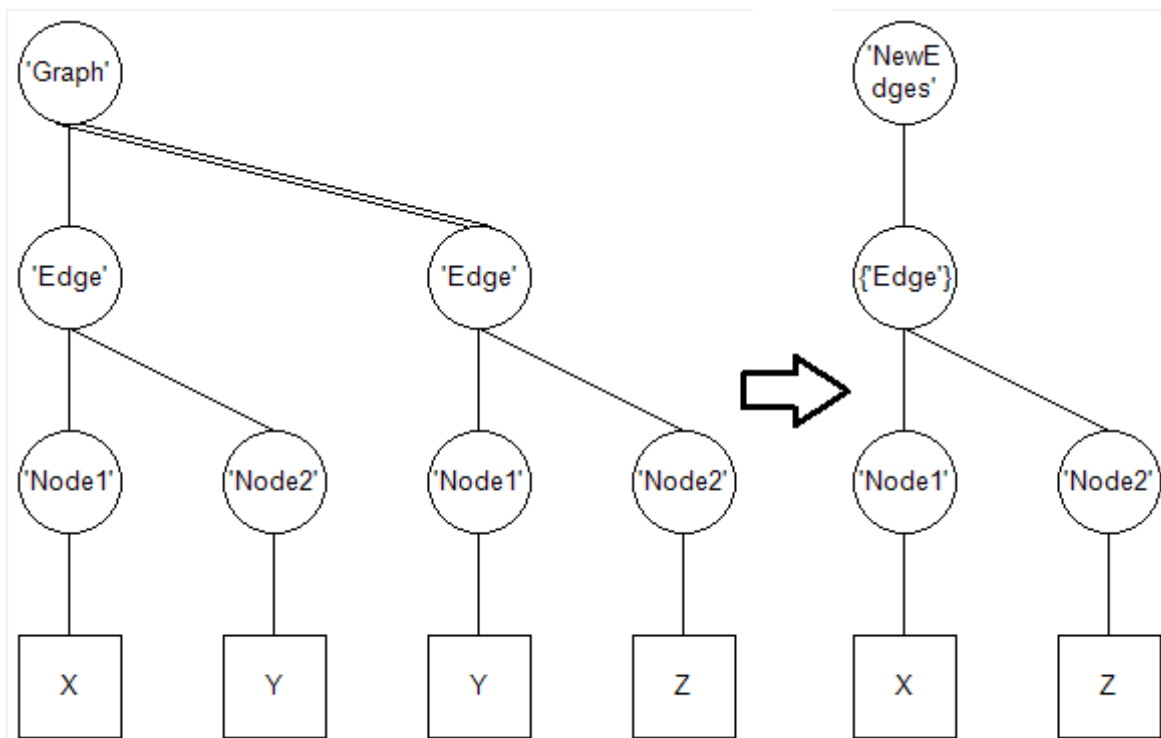
Az egyik lekérdezés, amit a mérésekhez használtam, egy gráf tranzitív lezárását megvalósító program. Ez a lényegét egy 'Iterate' függvényben valósítja meg, mely négy fő szabályt használ. Ezek futási ideje látható az 7. ábra-n a feldolgozott adatfa levélszámának függvényében. Itt a fa útjai nagyjából egyenlő hosszúak. (A „Program_Closure.xml” állományban található a program.)

- *input query 1*: Ez bontja ki az éleket az adatfából és összekapcsolja őket, ha lehet. Láthatjuk, hogy ez a leglassabb művelet. Ez magyarázható azzal, hogy a futás folyamán majdnem minden úton találunk illeszkedést, tehát a vágási feltételek itt nem érvényesülnek. Ráadásul a

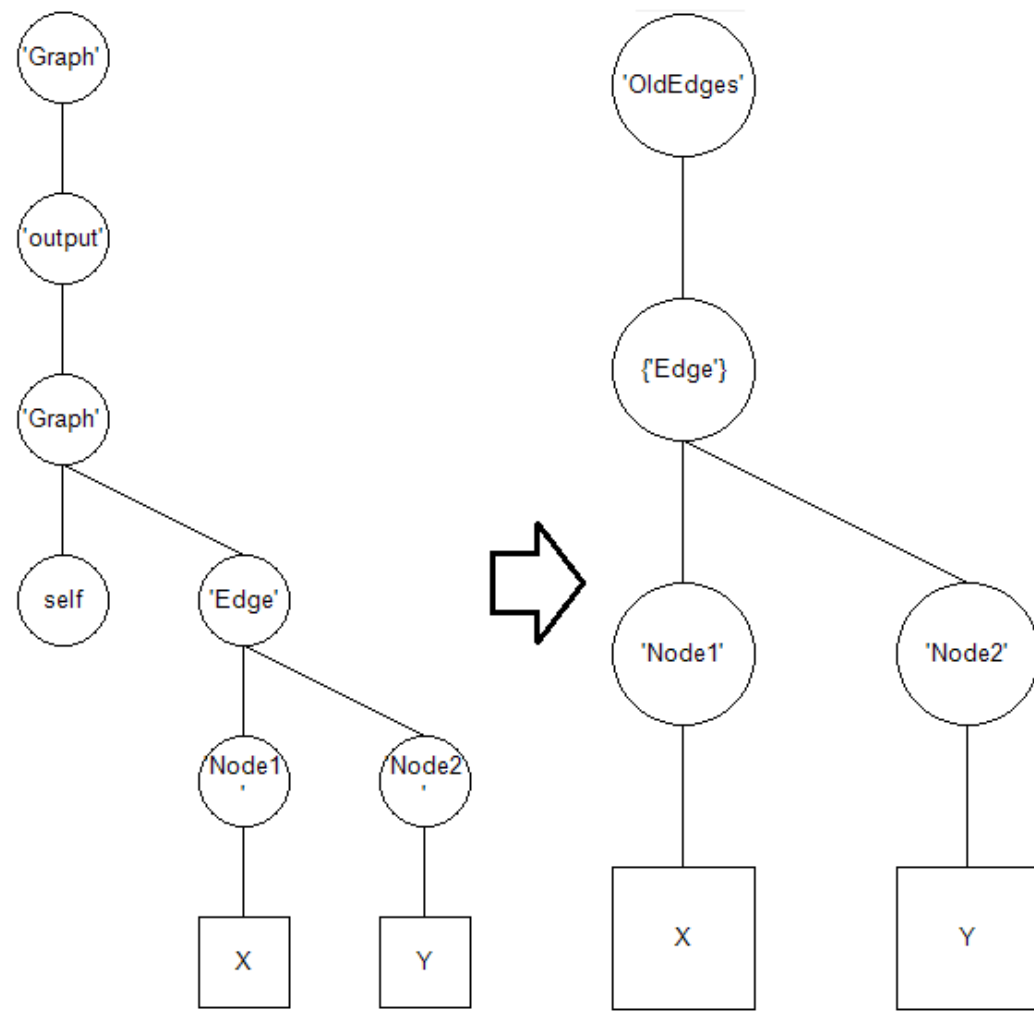
minta 4 útból áll, tehát az utak összekapcsolásának műveletigénye is érződik. (8. ábra)

- *input query 2*: Ez a lekérdezés a már létrehozott csúcsokat másolja át. Gyorsan lefut, hiszen csak 2 levele van a mintának és az adatfája is kicsi. (9. ábra)
- *output query 1*: A lekérdezés a munkaterületen fut le, és azokat a csúcsokat keresi meg, amelyek új éleket reprezentálnak, de nincsen velük azonos a régi élek között.
- *output query 2*: Ugyanazt teszi, mint az előző lekérdezés, csak eredménye nem egy részfa, hanem egy függvényhívás.

A vonalak meredeksége azzal is magyarázható, hogy a gráf lezárásánál sokszor van szükség az adatfa redukálására, amely megnöveli a futási időt. Továbbá a minta útvai mindig illeszkedni tudnak, ami a reprezentációból következik. Így láthatjuk, hogy egy kevésbé barátságos esetben is jól teljesít az adatfa szélességének függvényében.

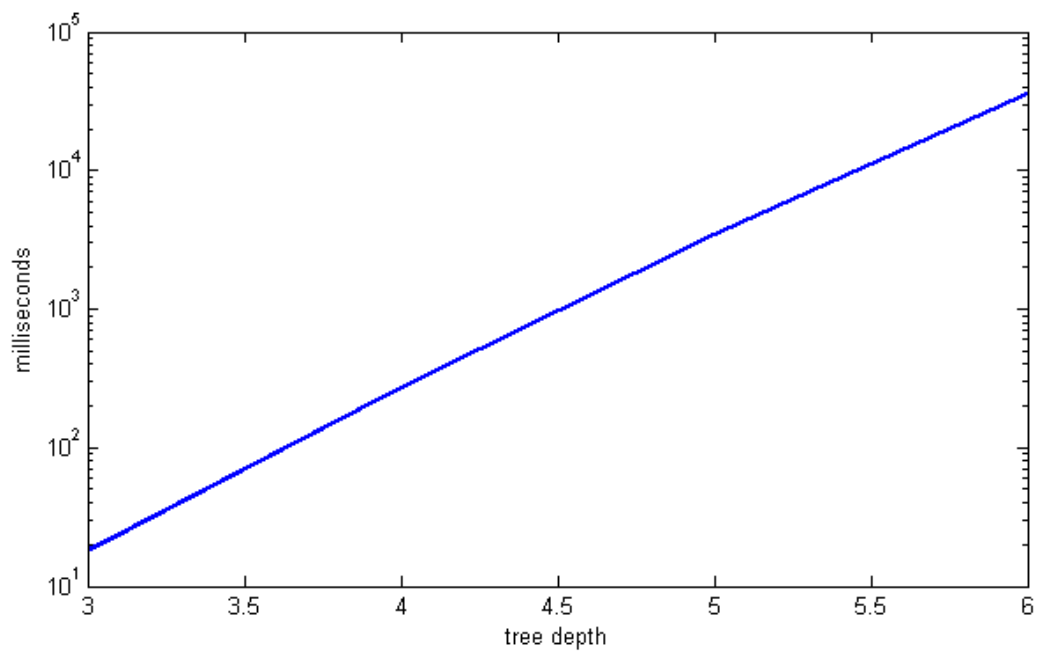


8. ábra

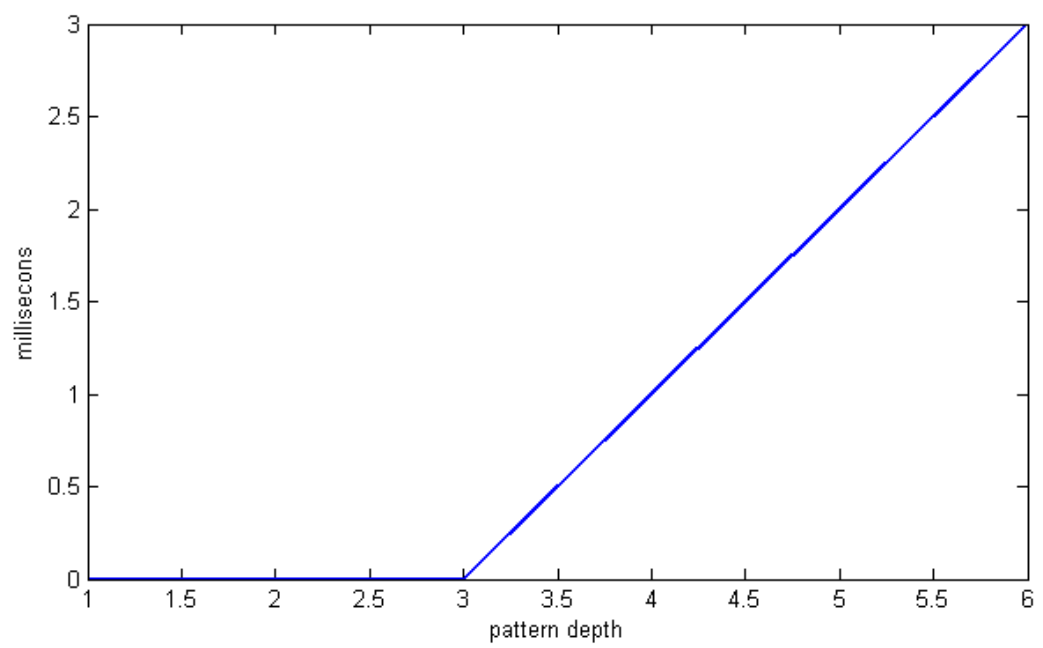


9. ábra

Megvizsgáltam az is, hogy mi történik, ha a minta magasságát változtatom. Itt egy olyan lekérdezést használtam, amelynek strukturális változóira nincs megkötés, tehát bárhová illeszkedni fog. Ekkor a műveletigény alig jobb, mint exponenciális. Ezt mutatja a 10. ábra. (A program a „Program_Deep_Unbound.axml” állományban található.)



10. ábra



11. ábra

Ellenben ha megvizsgáljuk ugyanezt a fastruktúrát, de most már nem hagyjuk a változókat szabadon illeszkedni, akkor a futási idő minimálisra csökken. Ezt mutatja a 11. ábra, ahol már milliszekundumok alatt lefut a hasonló lekérdezés. Ez sejteti mennyire fontos az ismert információk beépítése a lekérdezésbe, ezzel növelve a szelektivitást, és segítve a vágási feltételek érvényesülését.

Kitekintés

Dolgozatomban adtam egy algoritmust, mely képes az AXML lekérdezések egy jól definiált részhalmazát kiszámolni. Munkám fő terméke maga a mintaillesztés, és annak gyorsítása a speciális megszorítások kihasználásával.

A témában még sok nyitott kérdés marad, melyet érdemes lehet megvizsgálni részletesebben. Ilyen például az AXML fák belső reprezentációja, melyet esetleg nem láncolt szerkezettel is meg lehet valósítani, mely megnehezítené a karbantartást, de lehetővé tenné más megközelítés alkalmazását a minta illesztéséhez, mint amit a [7]-ben találhatunk.

További lehetőség lenne az általános feltételek megengedése. Ekkor valószínűleg nem feltétlenül tudnánk „monoton” módon keresni illeszkedést (azaz egy rossz próbálkozásról csak a végén derülne ki, hogy nem teljesíti a feltételeket, habár a fa szerkezete megfelelő), így csak a fa struktúrája alapján tudunk leképzéseket keresni, és ezeket utólag megszüntetni a feltétel szerint.

Hasznos lenne pontosabban megvizsgálni, milyen mintákra viselkedik jól az illesztő algoritmus, és egyéb módszereket is megvizsgálni. Így a minták előzetes vizsgálatával lehetőség nyílna az optimális illesztő algoritmus kiválasztására. Ilyen módszer lehet például az, ha a mintát nem utanként illesztjük, hanem globálisan kezdünk hozzá. Bizonyos típusú mintákra ez gyorsabban működhet.

A programot tovább lehet fejleszteni sok tekintetben, például a rugalmasabb szöveges reprezentációval, tehát az AXML programokat érdemes lehet valóban XML-ként tárolni. Vagy az őrfeltételek megvalósítását, mely gyakorlatilag megegyezik a minta illesztésével, de ott csak az illeszkedés ténye az érdekes, ezért célszerűbb lehet egy más mintaillesztési módszerrel megoldani, például olyannal, mint amit fentebb említettem. De a fák kirajzolását is lehetne közép-re igazított módon megoldani, ami lassabb rajzolást eredményezne, de kényelmesebb az emberi szemnek.

Köszönetnyilvánítás

Ez a dolgozat nem jöhetett volna létre oktatóim segítségével.

Köszönöm dr. Kiss Attilának a türelmét, segítő munkáját és motiválását!

Köszönöm dr. Fekete Istvánnak a szakmai jellegű kérdésekre adott iránymutatását!

Továbbá köszönöm dr. Gazdag Zsolt és Tichler Krisztián értékes tanácsait!

Irodalomjegyzék

- [1] Serge Abiteboul, Pierre Bourhis, Victor Vianu: Highly Expressive Query Languages for Unordered Data Trees, ICDT '12 Proceedings of the 15th International Conference on Database Theory, Pages 46-60
- [2] Claire David: Complexity of Data Tree Patterns over XML Documents, MFCS '08 Proceedings of the 33rd international symposium on Mathematical Foundations of Computer Science, Pages 278 - 289
- [3] W3C tutorial: <http://www.w3schools.com/xsl/>
- [4] W3C tutorial: <http://www.w3schools.com/xpath/>
- [5] W3C tutorial: <http://www.w3schools.com/xquery/>
- [6] Gazdag Zsolt: Bevezetés a számítástudományba (előadás jegyzet), 68. oldal
- [7] Gabriel Valiente, Conrado Martínez: An algorithm for graph pattern-matching, In Proc. 4th South American Workshop on String Processing, volume 8 of Int. Informatics Series, pages 180-197
- [8] Hector Garcia, H. G. Molina, Jeffrey D. Ullman, Jennifer Widom: Adatbázisrendszerek megvalósítása, Kiskapu kiadó, 423-434 oldal