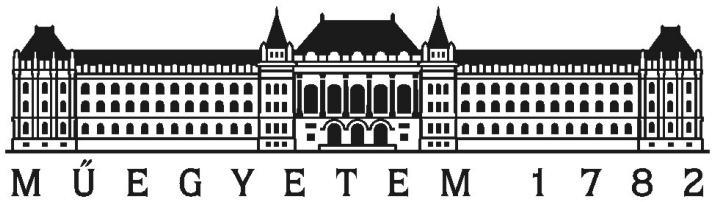


BUDAPESTI MŰSZAKI ÉS GAZDASÁGTUDOMÁNYI EGYETEM
VILLAMOSMÉRNÖKI ÉS INFORMATIKAI KAR
AUTOMATIZÁLÁSI ÉS ALKALMAZOTT INFORMATIKAI TANSZÉK

Gráftranszformációs keretrendszer adaptálása többmagos környezethez

TDK dolgozat

KÉSZÍTETTE:
Imre Gábor

KONZULENS:
dr. Mezei Gergely

2011.

Tartalomjegyzék

Kivonat

Dolgozatomban a modellezésben használt gráftranszformációk lehetséges párhuzamosítását és az ez által elérhető teljesítménynövekedést vizsgálom. A modellvezérelt programozás esetén a modellek feldolgozásának egyik legnépszerűbb módja a *gráftranszformáció.* A gráftranszformáció gráfújraírási lépések sorozatából áll, ahol egy újraírási lépésben egy megadott modellmintát keresünk, ill. cserélünk le. Nagyobb modellek, és minták esetén a transzformáció időigénye ugrásszerűen megnő, ezért kulcskérdés, hogy hogyan lehetséges a gráftranszformációt optimalizálni, a mintaillesztést és a cserét felgyorsítani. Az egyik lehetséges irány a transzformáció műveletének, különösen a mintaillesztés fázisának párhuzamosítása. Figyelembe véve a napjainkban rendelkezésre álló számítógépek felépítését, természetszerűleg merül fel az ötlet, hogy vizsgáljuk meg egy gépen belül *több processzormagon* végzett konkurens futtatás lehetőségeket.

Dolgozatom célja ezen a területen végzett kutatásom részleteinek ismertetése. A dolgozatban az elméleti és technológiai háttér bemutatását követően ismertetem a transzformációk párhuzamos végrehajtásának lehetőségeit. A dolgozat nem korlátozódik a kidolgozott megoldás elméleti bemutatására, a gyakorlati megvalósítás részleteit is ismerteti. A megvalósítás során az AAIT tanszéken készült VMTS keretrendszert vettem alapul. A dolgozatom egy komplett esettanulmány bemutatását is tartalmazza, az esettanulmány egy népszerű videómegosztó oldal feltöltései közötti kapcsolatokon végez transzformációt. Az esettanulmányban szerepelnek a kidolgozott párhuzamosítási módszerek hatékonyságát igazoló konkrét mérési eredmények is.

Abstract

In this paper, I examine the possibilities of parallelization and performance enhancement in software model based graph transformations. In model driven development using *graph transformations* is one of the most popular ways to process models. A graph transformation consists of multiple graph rewriting steps, in each step a pattern (called the *match*) is searched and replaced by the given rule. As the size of the model grows, the processing time increases exponentially. Therefore, optimizing graph transformations and speeding up pattern searches are of key importance in model processing. One possible direction is to apply the transformation – particularly the pattern matching phase – in parallel. Considering the personal computer architecture of nowadays, the idea of *multicore execution* comes natural.

The goal of this thesis is to elaborate the results of my research in parallel graph transformation in multicore systems. The thesis starts with a brief introduction about the mathematical and software development background of graph transformation. As next, the details of my approach are elaborated. The thesis is not limited to the theoretical background of the proposed solution, but it also introduces the practical realization of the system. The implementation of the approach is based on the VMTS framework developed at AAIT. Along the introduction of the development, a complete case-study with various measures is elaborated showing the efficiency of parallelization methods in practice.

1. Bevezető

Dolgozatomban bemutatom egy modellezésben használt gráftranszformációs megközelítésnek a párhuzamosítását. Az alkalmazott megközelítés típusos gráfokat használ, a keresendő mintákat és változtatásokat is típusosan írja le. A kutatásom során a VMTS modellezési keretrendszer megvalósításából indultam ki, és a célom az, hogy eredményeim a VMTS-ben is használhatóak legyenek.

A *Bevezető* után egy rövid *Történelmi visszatekintés* segítségével értelmezem a modellvezérelt paradigmák helyét az informatika világában. A következő fejezetben a modellezés és a párhuzamosítás néhány alapgondolatát foglalom össze, ezek a dolgozatban ismertetett megoldások megértéséhez elengedhetetlenek. A *3. fejezetben* a VMTS rendszert és a gráftranszformációk párhuzamosítási lehetőségeit ismertetem. Miután megvizsgáltam az elméleti alapokat, a fejlesztési környezetet és a lehetséges megvalósítási irányokat, a fejezet végén pontosan összefoglalom munkám célját.

A *4. fejezetben* bemutatom azokat a modelleket és adatokat, melyekkel a tesztelést végeztem. Az *5. fejezet* első részében ismertetem azokat a fejlesztési lépéseket, melyek során eljutottam a választott párhuzamosítási megoldáshoz, a fejezet további oldalain ezt a megvalósítást ismertetem elvi oldalról és futási eredmények alapján.

A *6. fejezetben* a munkám helyét és a jövőbeli fejlesztés irányait vizsgálom, a *7. fejezetben* pedig összefoglalom a dolgozatom főbb eredményeit. A *Függelékben* hasznosnak ítélt részletes elveket és megoldásokat ismertetek.

1.1. Történelmi visszatekintés

A múlt század közepétől a mai napig a számítógép folyamatos fejlődésének és egyre nagyobb térnyerésének lehetünk tanúi. A növekedés megállíthatatlannak látszik, újra és újra szemtanúi lehetünk korábban el sem gondolt vagy megvalósíthatatlannak hitt megoldások létrejöttének. A Moore-törvény – némi túlzással élve – lassan a világunkat leíró természeti törvények közé lép.

A számítógépek fizikai fejlődését mind a mai napig kíséri az informatika és a szoftverfejlesztés evolúciója. A nagyobb, gyorsabb rendszerek a programok fejlesztésében is sokkal jobb lehetőséget biztosítanak, és az évtizedek alatt összegyűlt tudás és tapasztalat is új megközelítésekben, új mentalitásban kondenzálódott.

A „hőskorban” gépi kód állt a programozók rendelkezésére, lyukkártyás gépek, jobb esetben egy-egy **assembler**. 1973-ban Brian Kernighan és Dennis Ritchie által megszületett a C programozási nyelv (Kernighan & Ritchie, 1988), amely processzor-közeliségének („hordozható Assembly”), ugyanakkor könnyebb használhatóságának köszönhetően hamar elterjedt, és mind a mai napig az alacsonyszintű fejlesztés etalonjának számít. Habár nem ez volt az első strukturált nyelv, mégis azt mondhatjuk, hogy a C-vel a programozás új szintre emelkedett: eljött a **strukturált programozás** ideje.

Egy évtized múlva Bjarne Stroustrup létrehozta a C++ nyelvet, amely – mint ahogy a neve is utal rá – a C-n alapul, de mégis: minőségileg nyújt többet, mint az elődje. A C++ számít ma az **objektum-orientált megközelítés** alapjának (Stroustrup, 2000), bár szintén nem az első volt a sorban, ld.

Simula67 (Simula - Wikipedia), Smalltalk (Smalltalk - Wikipedia). Az objektumorientált fejlesztés a strukturált programozás alapján áll, de egy nagyságrendi lépést jelent a fejlesztés átláthatósága és a programok újrafelhasználhatósága felé.

A 90-es évek több ponton is áttörést hozott az informatika világában. Megjelent az **internet**, aminek a társadalomformáló hatását, a mai napig nem látjuk át egészében. Elkészült a Java nyelv első verziója is, ami egyrészt a legjobban elterjedt **interpretált nyelvvé** vált, továbbá amelyet a *letisztult OO szemlélet* megjelenítőjeként is szoktak emlegetni. Azóta a különböző köztes kódon álló interpretált vagy **JIT-fordított nyelvek**[1] (pl. C# és .NET) – továbbá a komolyabb futásidejű feldolgozást igénylő szkriptnyelvek – a programozás számtalan területén egyre nagyobb teret nyernek. A legtöbb mai programozási projektben már nem létfontosságú a program alacsonyszintű optimalizálása, sokkal fontosabbá vált a gyors fejlesztés, és ezzel együtt az átláthatóság és a magas absztrakciós szint használata. A köztes nyelvek legnagyobb pozitívuma mégis a virtuális futtatókörnyezet segítségével elérhető platformfüggetlenség.[2]

A programozási szemléletmód evolúciója folytatódott tovább. A Windows különböző verzióival a programozók számtalan kellemes, vagy kevésbé kellemes (viszont mindenképp mély nyomot hagyó) tapasztalatra tettek szert a programjaik fejlesztése, verziózása kapcsán (pl. „*DLL hell*"). Megszületett a **komponens-alapú fejlesztés**, melyhez a CORBA, majd a COM (és utódai) képezték az alapot. Az objektumorientáltságban megismert kód-újrafelhasználás új szintre emelkedett. Mindeközben az *open-source* világból is több megoldás tört be az üzlet világába és kavarta fel azt, pl. Linux, Ruby on Rails (Hansson).

Az utóbbi években a **szolgáltatás-orientált architektúra** terjed a nagyvállalati rendszerekben, immár teljesen heterogén rendszerek interneten keresztül történő együttműködését lehetővé téve. Mindeközben az **aspektusorientált megoldások** segítik a programozók munkáját és az logikai egységeken, osztályokon átívelő feladatok megoldását.

A szoftverfejlesztés evolúciójában az *egyik legfontosabb mai irány* a **modellvezérelt programozás**, mint következő lépcső a szoftverfejlesztés evolúciójában. Az összes eddigi technológia és szemléletbeli újítás célja lényegében az *absztrakciós szint növelése,* a *fejlesztés gyorsítása* és a *programozási hibalehetőségek elkerülése* volt, rendszerint a teljesítmény rovására. (Fontos kérdés még a *platformfüggetlenség*, de ez dolgozatomhoz nem igazán kapcsolódik.) A strukturált programozás és az objektumok a fejlesztés magasabb absztrakciós szintjét szolgálták, az objektumok adatrejtése, a garbage collector és a menedzselt futtatási környezet pedig a hibák lehetőségét csökkentette, ezzel is segítve a gyors fejlesztést. A köztes kód értelmezése, illetve JIT-fordítása a platformfüggetlenséget valósítja meg.

A modellvezérelt programozás pontosan ebbe a sorba illeszkedik. Célja olyan modellek vagy szakterület-specifikus leírónyelvek[3] készítése, melyekkel *gyorsan*, *hatékonyan*, *hibáktól mentesen* és akár *platformfüggetlenül* lehet jól körülhatárolható feladatokat végrehajtó programokat, programrészleteket előállítani – mindezt úgy, hogy közben a program készítőjének elég ha ismerni a

[1] *Just-In-Time* fordítás: a program egy köztes assembly-jellegű nyelvre van fordítva, a futtatókörnyezet a végrehajtás során végzi el a natív kódra fordítást.
[2] Illusztrációként a Java nyelv támogatottságához: egyes ARM processzorok a Java bájtkód végrehajtását natív szinten támogatják, erre van a *Jazelle* futási mód. (Jazelle - ARM)
[3] Más néven DSL: *Domain Specific Language*, szakterületi nyelv

szakterületet, a programozáshoz nem kell értenie. Gondoljunk csak a *Form Designer* és más GUI-készítő eszközökre, vagy a *WorkFlow* ábrákra. Számtalan hasonló eszközre van szükség az ipar különböző területein, bankokban, nagyvállalatokban, ahol olyan keretrendszert érdemes használni, amely a megoldandó problémahalmazt fedi le, de nem szükséges, hogy annál többet adjon. A modellvezérelt programozás tehát a hatékony és hibamentes szakterületi programfejlesztést szolgálja.

2. Modellezési és párhuzamosítási alapok

A rövid történelmi-motivációs bevezető után tekintsük át a szükséges ismereteket, amelyek a dolgozatban vizsgált kérdések és a megoldások megértéséhez nélkülözhetetlenek. Először a modellvezérelt szemléletmód főbb gondolatait és néhány aktuális részletét mutatom be, majd a gráftranszformációk működését és a kutatásban felmerülő megvalósításának az alapjait, valamint a probléma matematikai aspektusait fogom röviden tárgyalni.

2.1. Modell, metamodell, meta-metamodell

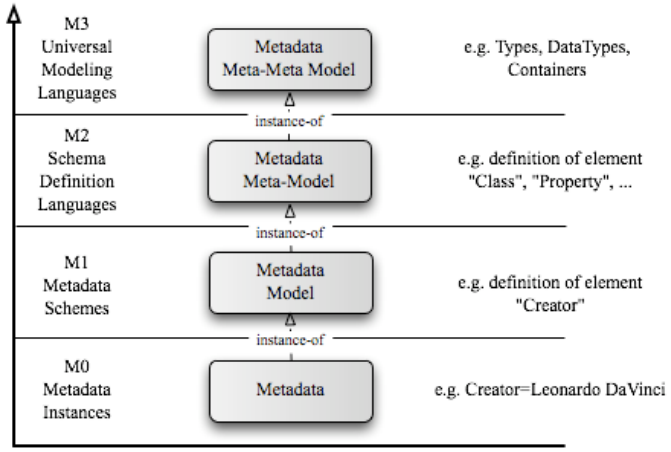
A metamodell alapú szakterületi programozásban legtöbbször az általános programozási megoldások helyett egy *speciális területre* koncentrálunk. A modelljeink fix eszközkészletből dolgoznak, nem tudunk velük tetszőleges feladatokat megoldani – cserébe viszont amit meg kell oldanunk, azt sokkal gyorsabban és egyszerűbben tudjuk elérni.

Az elkészítendő programot *modellnek* nevezzük. Ez írja le a konkrét feladatot megvalósító struktúrát, vagy algoritmust. Gondolhatunk itt egy banki ügyintézés folyamatára, egy GUI felületre, egy fizikai kísérlet összeállítására, vagy ezekhez hasonló szakterületi megoldásokra.

A modellünk fix eszközkészletből dolgozik, azaz előre definiáltak azok az elemek, amelyekből fel tudjuk azt építeni. Ilyen lehet például az ügyintéző általi adatrögzítés (név, cím, SZIG szám) vagy egy-egy GUI elem (*Button*, *TextBox*).

A modell értékkészletét a *metamodell* biztosítja. A metamodell tartalmazza mindazokat az elemeket, amelyek fontos szereppel bírnak az adott szakterületen. A metamodellből példányosítással kapunk egy konkrét modellelemet, és a különböző valós futtatási állapotok illetve az elkészült programok pedig a modellnek a példányai.

Felmerül a kérdés, hogy hogyan érdemes definiálni a metamodelleket, létezik-e megoldás általános metamodell-definiáló nyelvre. A kutatók több ilyen megoldással álltak elő, ezek közül a legfontosabb a MOF (OMG, 2006). A MOF specifikációjában az alábbi módon jelenik meg a négy szint:



1. ábra. A MOF 4 szintje (OMG, 2006)

A meta-metamodellek biztosítják azokat az általános koncepciókat, melyekkel le tudunk írni tetszőleges metamodelleket. Ezek között rendszerint szerepel az UML-ben megismert három fő kapcsolat az elemek között: asszociáció, kompozíció, öröklődés. Lényegében az UML is egy metamodell családnak tekinthető, amelyet általános programok tervezésére használhatunk – viszont többnyire túl általános ahhoz, hogy egy szakterületet hatékonyan le tudjunk vele írni, ezért hoznak létre további modelleket.

> **Megjegyzés.** A modellezésben nem szükségszerű az itt vázolt négy szint megléte, léteznek más megközelítések is, ahol kevesebb vagy akár több szinten is le tudjuk írni a rendszerünket. Mindazonáltal erre ritkán van szükség, nem véletlen, hogy a MOF is megelégszik a négy szinttel.

Tehát a modellezési keretrendszer bocsátja rendelkezésünkre a meta-metamodellt, ami lehetővé teszi a saját metamodellünk definiálását. Az általunk létrehozott metamodell a meta-metamodell egy példánya, amelyben meghatározzuk a modelljeink értékkészletét, ezzel együtt a modellezni kívánt rendszer határait. A metamodell definiálja a szakterületet leíró *DSL*-t.[4] Miután elkészítettük a szakterületet leíró metamodellt, létrehozhatjuk a metamodell konkrét példányait, azokat a modelleket, mellyel a valóságot akarjuk leírni. A modellek elkészítésére már az általunk definiált környezetet tudjuk használni, a fejlesztés itt már a szakterület fogalmait használja. Végül a futó programok a modelleket példányosítják és dolgozzák fel.

2.2. Modellek és gráftranszformációk

A modellvezérelt fejlesztés alapvetően egy szemléletmód, mentalitás, így nem meglepő, hogy több, gyökeresen eltérő implementációja is létezik. Az egyik legelterjedtebb modell-leírás gráfokkal történik. Ha a modelleket grafikusan készítjük, akkor kézenfekvő valamilyen UML-re hasonlító nyelv használata (értsd: dobozok és nyilak), ezeket pedig praktikusan tudjuk gráfok segítségével definiálni. A dolgozatomban ezen modell-leírásokat veszem alapul, azaz kutatásom során a modelleket címkékkel és attribútumokkal felruházott irányított éleket és pontokat tartalmazó gráfok formájában kezelem. Ilyen módon a gráfok lényegesen több információt tartalmaznak, mint a klasszikus matematikai gráfok, mint ahogy dolgozatom sem marad matematikai keretek között, hanem első sorban informatikai megoldásokról szól.

Gráfokkal leírt modellek *viselkedését* kézenfekvő **gráftranszformációkkal** leírni. A modellt képző gráfot bizonyos szabályoknak megfelelően alakítjuk, akár új elemeket adunk hozzá vagy törlünk, vagy attribútumokat módosítunk, gyakran a relációkat változtatjuk. A transzformációkkal le tudjuk írni a modellünk változásait, lehetséges alakulását, ami több szempontból előnyös: matematikailag precízen kezelhető, grafikus, az esetek nagy részében intuitívan megérthető, és meglehetősen hatékony.

Gráftranszformációk definiálására az egyik lehetőség az **LHS—RHS szabályok** megadása. Egy *Left-Hand Side (LHS)* és *Right-Hand Side (RHS)* szabálypár definiálja azt, hogy a gráf milyen részgráfját

[4] DSL (*Domain-Specific Language*): Szakterület-specifikus nyelv, melyet egy-egy jól körülhatárolható szakterület számára készítettünk, nem általános programozásra. Az eszközkészlete korlátozott, de éppen ezért lehet vele nagyon hatékonyan dolgozni az adott szakterületen belül. Jellemző példák DSL-re: áramkörtervező program, GUI- vagy Workflow-készítő grafikus felület.

milyen részgráffá alakítjuk. A feladat az LHS-nek megfelelő részgráf megtalálása a modellben, majd az RHS által leírt szükséges módosítások elvégzése. Az LHS keresése több koncepció szerint történhet, ezt a következő fejezetben részletezem. A hatékonyság és könnyű kezelhetőség érdekében egy vagy több szabály egyszeres, illetve többszörös futtatásából szoktuk összeállítani a teljes transzformációt.

2. ábra. LHS és RHS szabályok a Pacman világából

2.3. A gráftranszformáció matematikai háttere

Általános esetben a gráftranszformációk során az LHS minták keresése visszavezethető az NP-nehéz **izomorf részgráfkeresés** algoritmusára. A programozók általában nagy kihívásnak tekintik az NP-nehéz problémákat, mivel az ilyen bonyolultságú feladatok nagy része informatikailag nehezen kezelhető.[5]

Az LHS illesztése során a modellelemek a gráf pontjainak, az éleknek a megfelelő kapcsolatoknak feleltethetőek meg, mi pedig keressük a modellnek olyan, a mintával izomorf részeit, melyekre teljesülnek az egyéb típusbeli és kényszerekben megadott elvárások. Ekkor az algoritmus lépésszáma:

$$O(n,k) = \sum_{i}^{N} n_i^k,$$

ahol N a transzformációs lépések száma, n_i az alaphalmaz, melyen a transzformációt végezzük és k a keresett minta nagysága (Mezei, Transformation-based Support for Visual Languages, 2007).

Újra találkozhatunk azzal a jelenséggel, hogy a matematikusok és az informatikusok világa néha nagyban eltér egymástól.[6] Az informatikában magától értetődően alkalmazunk olyan megoldásokat, amelyek megkönnyítik az algoritmizálást és a feladat végrehajtási idejére is kedvezően hatnak, de a szemléletmódjuk elüt a matematikában megszokottól. Jelen esetben ez a megközelítés a típusosság és a példányosítás lesz, amivel ezt a láthatóan exponenciális algoritmust könnyebben fogjuk tudni kezelni.

2.3.1. Példányosító megközelítés

Az informatikusok a legritkább esetekben használnak csak olyan gráfokat, mint amilyenekkel a matematikában találkozunk. Az informatikában a gráfok rendszerint *típusosak*, az illesztések *névvel*

[5] Tipikus elméleti példaként szokták felhozni az **utazó ügynök problémát**, ahol általános esetben az optimális megoldást csak közelíteni tudjuk (Jordán, Recski, & Szeszlér, 2004). Sokkal húsbavágóbb a **biztonsági protokollok kérdése**, pl. az RSA titkosítás. A biztonsági protokollok feltörési módszerei rendszerint az implementáción találnak rést, az NP-teljes problémát megkerülve, mert matematikai oldalról a probléma ugyancsak megoldhatatlan. (Buttyán & Vajda, 2004)

[6] Ez gyakran azt eredményezi, hogy az informatikában hiányzik egy precízen kidolgozott matematikai alap, vagy éppen meglenne a matematikai háttér, de a kivitelezés ütközik nehézségekbe.

ellátottak (és gyorsan kereshetőek), valamint gyakran eleve számolhatunk az *élek multiplicitásával*. Nézzük tehát sorban ezeket a tulajdonságokat!

A minta első illeszkedő pontjának megtalálását a **típusosság** nagyban megkönnyíti. Egyrészt, a gráf elemeit könnyen el tudjuk menteni eleve típusos listákba, szótárakba, vagy más, hasonló tárolókba, így az elemek könnyen kereshetőek és elérhetőek lesznek a típusuk alapján. Másrészt a gráfon belüli navigálást is nagyban megkönnyíti a típusok rendelkezésre állása, hiszen kizárólag a potenciális találati lehetőségek irányába lépünk tovább, a felesleges próbálkozások nagy részét így ki tudjuk szűrni.

> **Megjegyzés.** Bár az implementáció szempontjából kevés a különbség, de matematikailag teljesen más szemléletet tükröz az, hogy a mintának megfelelő izomorf részgráfot keresünk-e a modellünkben, vagy pedig a *metaszinten megfogalmazott találat egy példányát*. Márpedig a mi rendszerünkben ez utóbbi eset áll fenn.
>
> Azaz eleve ott keressük a lehetőséget, ahol annak az előfordulását a metamodell lehetővé teszi. Az implementáció szempontjából ezt a típusos és névvel azonosított élváltozók alapján történő bejárás garantálja.

A típusosságon túl a típusos imperatív programozás adatszerkezetei garantálják, hogy a minta további éleit is direkt módon kereshetjük a modellünkben, hiszen az LHS szabályban rögzítve van, hogy mely változó mentén tudunk továbblépni. Így a bejárás során **egyértelműen azonosított éleken** tudunk haladni.[7] Ez azt jelenti, hogy a gráf bejárásakor nem kell az összes éllel foglalkoznunk, csak az adott metatípusúakkal – ez pedig lényeges gyorsítást eredményezhet.

A harmadik bonyolultsággal kapcsolatos faktor az élek **multiplicitása**. Az adataink nem változnak attól, hogy a multiplicitást rögzítjük a metamodellben,[8] ezért a multiplicitás ismeretével első sorban csak pontosabb becsléseket, valamint jobb heurisztikákat adhatunk. Az algoritmusunk akkor is ugyanúgy profitál az egyszeres multiplicitású élek meglétével, ha tetszőleges multiplicitást vár, tehát a multiplicitás leginkább a modell szintjén érdekes. Viszont ha a metamodell szintjén is ismerjük, akkor hatékonyabb algoritmusokat tudunk tervezni.

A gyakorlati problémákat leíró gráfokban az élek jelentős része zérus, illetve egyszeres multiplicitású. A multiplicitásokra ugyanakkor nem lehet szigorú állításokat kimondani, hiszen a kérdés teljesen metamodell függő, de heurisztikákat lehet rá alapozni. Magától értetődő, hogy zérus, ill. egyszeres multiplicitású élekkel rendelkező gráfban a hagyományos algoritmusok a keresési minta és a bemeneti modell méretétől konstans módon függenek. Ez az arány akkor nem teljesül, ha megjelennek a többszörös multiplicitások.

A mintaillesztés bonyolultsága a relációk fokszámának függvényében:

[7] Konkrétan: adott a metamodell szintjén megfogalmazott élváltozó vagy éllista, amely konkrét memóriacímen található, az objektum címének ismeretében egy, vagy néhány utasítással elérhető. Más változót nem kell megvizsgálnunk, csak ezt az egyet, illetve a lista elemeit.

[8] Bár az adatszerkezet meg tud változni. Most tekintsünk el ettől az esettől, állításom arra vonatkozik, amikor már összeállt az alkalmazni kívánt modell.

$$C \leq K \cdot \prod_{i=0}^{N_r} d_i,$$

ahol C a keresés költsége, K az egy elem ellenőrzésének egységi költsége, N_r a vizsgált relációk száma, d_i pedig az i. reláció fokszáma. *A korlátozott multiplicitások jelentősen csökkenthetik a modellillesztés bonyolultságát*, a bizonyítást lásd a **Függelékben**.

***

A fenti három faktor, tehát a *típusok szerinti keresés*, az *egyértelműsített gráfbejárás* és a *maximum egyszeres multiplicitások jelenléte* nagyban növeli a mintaillesztés hatékonyságát, azonban a probléma elvi NP-nehézségét nem oldja fel.

2.4. A párhuzamosítás lehetőségei

A processzorok felépítése lassan kvantummechanikai határokat súrol (32 nanometer - Wikipedia), ezért ezeket az eszközöket a hagyományos sebességnövelésen túl az újabb magok beépítésével fejlesztik. Egyes mobiltelefonok már többmagos mikroprocesszorokkal rendelkeznek (Coretex-A5 Processor - ARM). Dolgozatom célja a modellezés során használt gráftranszformációk teljesítményének a növelése párhuzamosítás által, így röviden áttekinteném a többmagos processzorok által kínált párhuzamosítási lehetőségeket és nehézségeket.

A többmagos környezet főbb jellemzői a következők:

1) **Közös memóriaterület**
   Praktikusan: A nagyobb, blokkos adatok minden szálból elérhetőek, olvashatóak és írhatóak.
2) **Szinkronizálás**
   A fenti miatt az adatmódosítást szinkronizálni kell. A szinkronizálás erőforrás igényes művelet, ezért kulcskérdés a szálak minél nagyobb függetlensége.
3) **Egyszerű és gyors szálközi kommunikáció**
   Legegyszerűbben közös memóriaterületen, szinkronizált eléréssel. Cél az egyszerűség és a kommunikáció minimalizálása, a minél kisebb többletmunka.

A többmagos rendszereken kívül a további főbb párhuzamosítási lehetőségek a *több processzoros rendszerek* (szimmetrikus vagy aszimmetrikus) és szuperszámítógépek, valamint az *elosztott rendszerek* különböző típusai (pl. klaszterek és gridek) – csak hogy néhány fontosat említsek.[9]

A fentieket figyelembe véve törekedtem a minél hatékonyabb többmagos rendszereken futtatható párhuzamos megoldások kidolgozására, valamint a teljesítmény mérésére.

2.4.1. Cache

A többmagos rendszerek programozása elvileg a legegyszerűbb az összes párhuzamosítási lehetőség közül. A helyzet egyszerűségét sajnos erősen rontja a modern processzorok felépítése, ezen belül a

[9] Ide tartozna még a különböző speciális rendszerek és eszközök is, például SIMD processzorok (*Single Instruction – Multiple Data*), vagy grafikus vezérlők általános programozása.

gyorsítótárak használata. A mai processzormagokban két- vagy háromszintű cache található, ami a processzor minél gyorsabb adatelérését gyorsítja. Viszont, mivel külön cache van a processzormagokhoz, ez nehézséget okozhat, amennyiben a két processzor ugyanazt a memóriaterületet szeretné használni.

A probléma akkor jelentkezik, ha az adott memóriaterület (értsd: változó) két cache-ben is szerepel, és az egyik cache-ben a beolvasások után módosítják. Ekkor ugyanis a másik cache automatikusan nem fogja frissíteni az adatokat. Egyáltalában is rendszerfüggő az, hogy a módosítótól a „dirty cache" mikor kerül visszaírásra, de általában is, a cache célja az, hogy gyors legyen, ezért nem várható azonnali szinkronizáció minden alkalommal. Hagyományos esetben erre nincs is szükség.

A megoldást a főbb imperatív nyelvekben rendelkezésre álló *volatile* kulcsszó adja, ami gyakorlatilag tiltja a változó gyorsítótárazását, mondhatni bármikor megváltoztathatja azt egy másik szál. Így megoldhatjuk a szálak által közösen használt változók értékének a helyes elérését, viszont a módosítások további problémákat vetnek fel.

2.4.2. Szálak ütemezése

A másik fő problémát az adja, hogy a szálak végrehajtása szinte teljességgel mellőzi a determinizmust. Többszálú program végrehajtása futtatásról futtatásra változik, mert az operációs rendszer mindig máskor szakítja meg az egyik szál futását, hogy a másik szálat folytathassa.

A megoldást az *atomi műveletek*, vagy a *kölcsönös kizárás* jelentik: garantálnunk kell, hogy egy-egy megszakításra érzékeny műveletsorozat nem szakad meg, vagy legalábbis megszakítása esetén másik szál nem fog olyan programrészletet végrehajtani, ami bajt okozhatna. Azaz a közös memória vagy változó inkonzisztenciája nem vezet problémához. Az említett két megoldás szintén megtalálható az általánosan használt programozási nyelvekben.

> **Megjegyzés.** A szálkezelés és az ehhez tartozó szinkronizációs megoldások konkrét implementálása programozási környezetenként és operációs rendszerenként változnak, de az alapvető megoldásokat minden környezet rendelkezésre bocsájtja.

Többmagos környezetben a többi párhuzamos rendszerhez képest sokkal több lehetőségünk van, viszont a hibák számára is sokkal több a lehetőség.

2.4.3. Nyelvi lehetőségek

A párhuzamos programozás során a következő eszközökre van szükségünk:

1. Hatékony párhuzamos végrehajtás
2. Szinkronizációs, illetve kommunikációs lehetőségek

A széles körben használt imperatív programozási nyelvek (első sorban C++, C#, Java) hasonló lehetőségeket biztosítanak a fenti feladatokra. A párhuzamos végrehajtást az **újabb szálak indítása** adja. A létrehozott további szálak ugyanazt a memóriaterületet látják és a szülő szállal párhuzamosan futnak. A szálak indítása viszonylag költséges művelet, ezért gyakran használunk ún. ***Thread Pool*-okat**, olyan száltárolókat, melyek futásra kész szálakat biztosítanak a programjainknak. Az egyik

mérés során saját *ThreadPool* osztályt is készítettem, melyet összehasonlítottam a C# beépített megoldásával.

A C# nyelvi szinten biztosít több magas szintű megoldást a párhuzamosításra, melyek elrejtik a szálak létrehozását és kezelését az egyszerűbb használat érdekében. Az egyik ilyen megoldás a *Parallel* osztály, mely tipikus ciklusok párhuzamos futtatását teszi lehetővé, így például olyan *for* ciklust tudunk vele megvalósítani, melyben a ciklusmagot egy-egy egész paramétert megvalósító függvény adja, és a rendszer ezeket egymással párhuzamosan futtatja. A LINQ[10] is biztosít párhuzamos lekéréseket, de ezeket a dolgozat elkészítése során nem használtam fel.

A szinkronizációra, illetve kommunikációra (mert a szinkronizáció is egyfajta kommunikáció) pedig a klasszikus **Semaphore** (szemafor) és **Mutex** megoldásokat, valamint ezek különböző verzióit használhatjuk (Schmidt, Stal, Rohnert, & Buschmann, 2000).

A szemaforok akkor használatosak, amikor az egyik szállal be akarjuk várni egy másik eredményét, pl. forrás—nyelő szálak kezelése. A mutex-ek (*MUTual EXclusion*) pedig a kölcsönös kizárást valósítják meg, ilyen szokott lenni egy több szálból módosított változó. A C#-ban nyelvi szinten szerepel a *lock* parancs, de ez újdonságot nem jelent, mert egy objektumhoz kötött kölcsönös kizárást valósít meg átláthatóbb módon (Microsoft, 2011).[11]

Nem elegáns, de sajnos megkerülhetetlen a korábban említett *volatile* kulcsszó használata. Így jelöljük azokat a változókat, melyeket több szálból is váltakozóan írhatunk és olvashatunk, értéküket a processzor minden adateléréskor újratölti a memóriából.

Elméletileg rendelkezésre állnak ezeknél jóval szofisztikáltabb megoldások is, azonban ezek rendszerint a teljesítmény rovására mennek – és az egyszerűbb algoritmusokban többnyire nincs is rájuk szükségünk.

> **Példák.** A szálkezelés magasabb szintű megoldásai közé tartoznak az *üzenetsorok* és a *pipeline* használata. Ezekkel tetszőleges adat küldhető a szálak között, a feldolgozás történhet aszinkron módon – azaz nagyon kényelmesen használható megoldást adnak. Viszont a háttérmunkát az operációs rendszer végzi, amire hagyatkozni algoritmizálási feladatokban általában nem jó döntés.
>
> Ritka esetekben (vagy alacsonyszintű rendszerekben) a teljesítménykritikus programok külön megvalósítják az operációs rendszer némely szolgáltatását is, pl. saját memória allokátorral rendelkeznek.

Összességében: a klasszikus alacsonyszintű lehetőségek minden széles körben használt nyelvben hasonlóan állnak rendelkezésre, emiatt az itt ismertetett megoldások elméletileg programozási nyelvtől függetlenek.

***

[10] LINQ: *Language Integrated Query*: A C# egyik új megoldása, többek közt a programozási nyelvbe illeszthető típusos és fordítási – vagy gépelési – időben ellenőrzött SQL-jellegű lekérdezéseket tesz lehetővé.
[11] A *lock* utasításról részletesen: http://msdn.microsoft.com/en-us/library/c5kehkcz(v=vs.71).aspx

Az előbbi néhány fejezetben dióhéjban ismertettem a szoftverfejlesztés történelmi evolúcióját, és rámutattam, hogy a fejlődésbe a modellvezérelt megközelítés illeszkedik, sőt bizonyos területeken akár a következő lépcsőt is jelentheti. A modellvezérelt szoftverfejlesztésben több irányzatot láttunk, ezek közül az egyik leggyakrabban használt megvalósítás a modellt egy gráfként írja le, és a rendszer működését gráftranszformációkkal modellezi.

A gráftranszformációk matematikai hátterét és a matematikai és informatikai megközelítés különbségeit röviden bemutattam, végül, de nem utolsó sorban pedig a párhuzamos programozás néhány alapkérdését fejtettem ki.

A következő fejezetben ismertetem a keretrendszert, amelyben dolgoztam és a gráftranszformációk párhuzamosításának módjait, ezek után konkretizálom a kutatásom céljait. A további fejezetekben a párhuzamosításban, illetve egyéb teljesítménynövelésben elért eredményeimet ismertetem, majd bemutatom a további fejlesztés lehetséges irányait.

3. A feladat

A dolgozatom célja egy, már meglévő rendszerben a gráftranszformációk teljesítményének a növelése a párhuzamosítás lehetőségeinek kihasználásával. Ennek érdekében bemutattam a főbb előismeretek témaköreit (modellezés, párhuzamosítás), az alábbiakban pedig a konkrét környezetet, azaz a VMTS keretrendszert ismertetem, majd a rendszer által biztosított párhuzamosítási lehetőségeket vizsgálom meg. Mindezek után pedig konkretizálom a dolgozatom operatív szintű céljait.

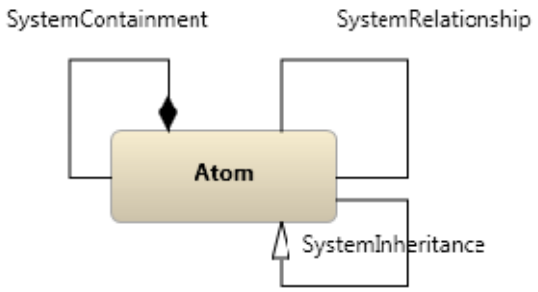
3.1. A VMTS által biztosított környezet

A *Visual Modelling and Transformation System* az AAIT tanszék által készített grafikus keretrendszer, mely lehetőséget ad metamodellek és modellek készítésére, valamint az ezeken végzett transzformációk összeállítására és elvégzésére.

3.1.1. Modellezési megfontolások

A VMTS n-szintű modellezési rendszert használ, azaz a modellszintek száma nincs korlátozva. Ez azt jelenti, hogy tetszőleges példányosítási lépésen keresztül állíthatjuk össze a kívánt keretrendszerünket, bár négy szintnél többre általában nincsen szükségünk.

A rendszer ős modellje (a legfelső, önleíró mentaszint) egy elemet és három relációt biztosít. Az elemekben és a relációkban létrehozhatunk egyszerű vagy komplex attribútumokat, a relációk multiplicitását beállíthatjuk, meghatározhatunk rendszer-szintű OCL kényszereket, stb. A VMTS dokumentációját a (BME AAIT, 2011) tartalmazza. A meta-metamodell ábrája a következő:



3. ábra. A VMTS meta-metamodellje

A VMTS rendelkezik azzal a sajátossággal, hogy minden, azaz *minden* modellelemnek a meta-metamodell az alapja. Ez azt jelenti, hogy mind a szakterületi nyelveink, mind a transzformációt leíró, vagy akár a gráfújraírást leíró nyelvnek a metamodellje is ennek az ős metamodellnek egy példánya. A rendszer alapvetően minden szinten ugyanazt a megjelenítő és kezelő motort használja, bár ez szakterületenként kibővíthető ill. átalakítható, igény szerint.

Egy teljes szakterületi keretrendszer elkészítése általában a következő lépésekből áll.

1. Létrehozzuk a **metamodellt**

Itt határozzuk meg azt, hogy a modelljeinkben milyen elemek, milyen attribútumok, valamint az elemek között milyen kapcsolatok lehetségesek.

2. Létrehozzuk a **modellpéldányt**
   A létrehozás általában a VMTS rendszeren belül történik, de lehetséges saját modell importálása is, mivel a rendszer a metamodellből osztálykönyvtárt készít.
3. Meghatározzuk a **gráftranszformációs szabályokat**
   A gráftranszformációkat LHS és RHS szabályokkal írhatjuk le, azaz meghatározzuk, hogy mit keresünk (*Left-Hand Side*), és azt, hogy mivé alakítjuk (*Right-Hand Side*). A két oldalt egyszerre ábrázolva látjuk
4. A szabályokból **transzformációkat** készítünk
   Ha megvannak a szabályok, akkor meghatározhatjuk a szabályok végrehajtásának a módját és sorrendjét. Egy szabályt futtathatunk egyszer, vagy mindaddig, amíg van találat, illetve attól függően, hogy volt-e találat, a végrehajtás elágazhat.

Innentől a metamodellnek megfelelő modelleken tetszés szerint végezhetjük el a kívánt szabályokat, illetve transzformációkat.

3.1.2. A VMTS belső működése

A VMTS teljes egészében a .NET keretrendszeren alapul. Minden elkészített modell után (legyen az egy saját modell, egy szabály, vagy szabályokat tartalmazó transzformáció) a rendszer bináris, pontosabban bájtkód-alapú[12] komponenst készít, amelyet az újabb projektekben már fel is használ, ezzel rendszer-szintű támogatást biztosít. Így például egy transzformáció számára bemeneti modellként határozhatjuk meg az általunk készített modelleket, ezáltal a transzformáció látja a lehetséges elemeket és azok attribútumait.

A fejlesztés során ezen a ponton kapcsolódtam be a VMTS rendszerbe. A párhuzamosítás alapját mindig a VMTS által generált kód képezte kisebb-nagyobb módosításokkal, ezeket a lépéseket a következő fejezetben ismertetem.

Mivel a VMTS mindenhol .NET technológiákat használ, jelen fejlesztési fázisban én is maradtam ennél a keretrendszernél. Megjegyzendő ugyanakkor, hogy általában véve az algoritmizálási feladatokra a C# vagy Java típusú (JIT fordított, illetve interpretált) nyelvek nem hatékonyak. Teljesítménykritikus esetekben nem megkerülhető a C, vagy legalábbis a C++ használata. (Ezzel foglalkozik többek közt (ucblockhead, 2002)) Ennek ellenére a fejlesztés jelenlegi szakaszában maradtam a .NET környezetnél, hogy megmaradjon a kompatibilitás a VMTS többi részével. Viszont az itt bemutatott algoritmusok elég általánosak ahhoz, hogy a C++-ra való átállás ne okozzon különösebb nehézséget.

> **Megjegyzés.** Az átállás leginkább annyiban okozhat problémát, hogy amit C# alatt megérte több szálon futtatni, azt nem biztos, hogy ugyanúgy meg fogja érni C++ natív kódban több szálon futtatni.

[12] CLR – *Common Language Runtime*: a .NET programok köztes kódja, ami a fordítás után, de a JIT-fordítás előtt készül.

A hosszú távú célok között szerepel a VMTS egyes algoritmusainak C++-os implementálása. Mivel a metamodellből mindig egy sablon alapján állítjuk elő a szakterületet leíró bináris állományt, egy másik sablon segítségével ez átalakítható C++ implementációvá.

3.2. Párhuzamosítási lehetőségek

Korábban volt szó a párhuzamosítási lehetőségekről általánosságban. Most megmutatom a VMTS működésével összhangban azt a két fő párhuzamosítási lehetőséget, amivel a gráftranszformáció teljesítménye növelhető.

A korábban említettek szerint a VMTS-ben egy transzformációt két lépésben határozhatunk meg. Először megadhatjuk az egyes **transzformációs szabályokat** az LHS és RHS gráf együttes ábrázolásával, azaz itt meghatározhatjuk, hogy mit keresünk, és hogyan változtatjuk meg a találatot. A kész szabályokból pedig teljes **transzformációkat** komponálhatunk, ahol megadhatjuk a korábban létrehozott szabályokból a teljes transzformációs munkafolyamot.

A párhuzamosítás alapvető kérdése így az, hogy szabály-, vagy transzformációs szinten szeretnénk-e a párhuzamosítást megvalósítani. Ezekben a témákban aktív kutatás is zajlik, röviden utalok ennek eredményeire.

3.2.1. Szabályok párhuzamosítása

A legegyszerűbb párhuzamosítási lehetőség abból áll, hogy párhuzamosan dolgozunk fel egyugyanazon szabályhoz tartozó találatokat. Egy szabály végrehajtása két részből áll: először meg kell találni a szabály LHS példányát a modellben (említettem, hogy típusos, metamodell-alapú keresést végzünk – ezért nem a klasszikus értelemben vett izomorfiáról van szó), utána pedig az RHS eléréséhez szükséges módosításokat kell elvégeznünk. Mivel rendszerint nagy modellekkel dolgozunk, általában a minta megkeresése bizonyul a nehezebb, időigényesebb feladatnak, míg a módosítások megvalósítása általában gyorsan elvégezhető.

A gyakorlati megvalósításokban a *keresés és a módosítás szakaszait erősen szétválasztottam*, ennek fő okát a **Függelék B. részében** ismertetem. A szabályok párhuzamosítása alatt innentől azt értem e dolgozatban, hogy párhuzamosan keresem a találatokat, majd valamilyen stratégia szerint leállítom a keresést és elvégzem a felgyülemlett módosításokat.

A találatok kezelését alapvetően meghatározza az, hogy mennyi találatot keresünk, hány módosítást akarunk egy futtatás alatt elvégezni. Gyakorlatban ez az első vagy az összes lehetséges találatot szokta jelenteni. A párhuzamosítás az utóbbi esetben ad hatékonyabb megoldást.

3.2.2. Transzformációk párhuzamosítása

A (Mezei, Transformation-based Support for Visual Languages, 2007) disszertáció ismertet egy megoldást, melynél egy transzformációban a felhasznált szabályokat olyan blokkokba sorolja, melyeket egymás után szekvenciálisan kell végrehajtani, de a blokkokon belül lévő szabályokat tetszés szerint párhuzamosíthatjuk.

A blokkokra bontás első megközelítés szerint annak a függvényében történik, hogy az alkalmazott szabályok a metamodell milyen elemeiből indulnak ki, ha ugyanis nincs közös elemük, akkor elméletileg sem hathatnak egymásra a különböző transzformációk – tekintsünk el attól az esettől, amikor úgy vizsgáljuk egy él multiplicitását, hogy maga az él már kilóg a keresett mintából. Megjegyzem, hogy ha az LHS tartalmaz minden érintett élt, akkor a Függelék B. részében leírt törlő módosítás hibalehetősége itt nem fordulhat elő. A megoldás tovább finomítható azzal, hogy olvasásra rendelkezésre bocsátunk azonos típusú elemeket a transzformációk között, viszont a módosítás, törlés, illetve új elem létrehozása továbbra is kizárólagosan foglalná le a metaszinten definiált típust – azaz más transzformáció nem férhet hozzá ugyanilyen típusú elemekhez. Ez elvi szinten zár ki bármiféle átfedést a találatok között.

Az algoritmus tehát probléma nélkül használható lenne, viszont egy gyakorlati nehézség merül fel ezzel kapcsolatban: Az ismertetett megoldás egy „mindent vagy semmit” jellegű megközelítés. Azaz vagy gond nélkül tudunk párhuzamosítani, vagy nem, viszont akkor garantált az egyszálú futás. Ezzel együtt felmerül annak a kérdése is, hogy vajon az egyes szabályok a felhasznált típusok tekintetében mennyire függetlenek a gyakorlatban? Ugyanis ha a gyakorlati alkalmazásokban sok az átfedés, akkor a párhuzamosítás ritkán lesz lehetséges. Ennek a kérdésnek az eldöntésére további kutatás szükséges, dolgozatomban nem foglalkoztam vele részletesen.

***

Léteznek más megközelítések és hibrid megoldások (pl. (Mezei, Transformation-based Support for Visual Languages, 2007)), de itt nem találtam olyan, viszonylag egyszerű, ugyanakkor kellően hatékony algoritmust, amelyet többmagos környezetben igazán érdemes lenne használni.

3.3. Célok

A rendelkezésre álló, gyakorlatban is használt gráftranszformációkat felmérve arra jutottam, hogy a transzformációk során többnyire összes találatot kereső szabályokat alkalmazunk.

> **Példák.** Tipikusan összes találatot keresünk egy modell refaktorálásakor, mert itt az összes régi modellelemet újjal szeretnénk helyettesíteni. Hasonló a helyzet, amikor egy modellt általánosabb értelemben fel szeretnénk dolgozni, rendszerint a modell teljes terén el akarjuk végezni a lehetséges műveleteket. Valamint a VMTS eddigi projektjei is többnyire kimerítő szabályokat futtatnak.

Előfordulnak ugyan egyszeres találatot kereső szabályok is, de ezek viszonylag ritkák. Az egyszeres találatok legtöbbször elágazásként funkcionálnak a transzformáció végrehajtásában, más úton megy a feldolgozás, ha van találat, és más úton, ha nincs.

Elsődleges számú feladatnak egy *hatékony szabály-szintű párhuzamosítás megtervezését és kivitelezését* tűztem ki, mivel ez általánosan használható, ellenben a transzformáció-szintű párhuzamosítással. A transzformáció-szintű párhuzamosítás nem mindig használható, és van, amikor használható, de keveset nyerünk vele, mert az egyik transzformáció futási ideje lényegesen eltér a másikétól – így a terheléseloszlás sem lesz egyenletes.

A szabály-szintű párhuzamosítás teljesítményét növeli, ha az *összes találatot keressük* – ezt a mérési eredmények világosan ábrázolják. Ezekkel az elvárásokkal kezdek neki a párhuzamos algoritmus kidolgozásának.

Viszont a feladatom nem csupán a gráftranszformációk párhuzamosításából áll. A párhuzamosítás hosszú távú célja az, hogy a VMTS jelenleg használt kódgeneráló algoritmusát módosítsam úgy, hogy az a többmagos rendszerek teljesítményét is jobban ki tudja használni. A végső cél az, hogy *a VMTS automatikus szabály- és transzformáció-generálása már eleve párhuzamosan futtatható kódot készítsen*.

Felmerül a kérdés, hogy vajon érdemes lesz-e a jelenleg C#-ban megvalósított megoldásokat alacsonyabb szinten is használni. Tehát a teljesítmény ugrásszerű növekedés nem teszi-e feleslegessé az itt létrehozott megoldásokat, illetve mennyire maradnak használhatóak a jelenlegi mérési eredmények? Amire biztos számíthatunk: egy C++-os implementáció nem fog ugyanilyen eredményeket adni, de az itt használt megoldások nem lesznek messze az optimumtól. Finomhangolásra bizonyára szükség lesz, de gyökeres változtatásra nem.

4. A felhasznált modellek és transzformációk

Az algoritmusaim hatékonyságát külső forrásból választott adatokon teszteltem. A párhuzamosítás hatékonyságának vizsgálatára egy olyan adathalmazra volt szükségem, amely kellően nagy méretű, nem teljesen homogén elemekből áll, és tartalmaz attribútumokat. Az attribútumokra a kényszerek vizsgálata során van a legnagyobb szükség. Cél volt továbbá, hogy ne egy általam generált mesterséges modellen végezzem a transzformációkat, hanem egy természetes struktúrán.

4.1. YouTube videodata

A (Cheng, Dale, & Liu, Statistics and Social Network of YouTube Videos, 2008) publikáció számol be egy olyan adathalmazról, amely tökéletesen megfelel a dolgozatomban kapcsán támasztott kritériumoknak.[13] A Simon Frasen egyetem munkatársai a YouTube oldalról töltöttek le videó statisztikákat egy *crawler* program segítségével, majd ezeket elemezték, publikálták. Írásaikban sok érdekes állítást tesznek a YouTube közösségi hálózata kapcsán – ami viszont számunkra is fontos: a bejárási eredményeket publikussá tették (Cheng, Dale, & Liu, YouTube Dataset, 2008).

A bejárás több alkalommal történt meg, így különböző méretű és jellegű gráfok állnak rendelkezésre. A bejárás az adott időpontban a YouTube által kategóriánként megjelenített legnépszerűbb, legtöbbet nézett, stb. videókból indul ki, majd további három lépésben az előző körben talált videóktól induló első 20 kapcsolódó videót emeli be a meglévők mellé. Az indulási videók száma nagyságrendileg 200 körüli, és minden körben a videók száma exponenciálisan nő. A találatok átlagosan 50—200 000 videót tartalmaznak. Az ismétlések eloszlása változó, emiatt elég nagy a szórás a gráfok elemszámában.

Az elérhető adatbázis tartalmazza a videókról a következő adatokat:

- Videó azonosítója, feltöltő neve és kategória neve
- Videó feltöltésének ideje, hossza
- Megtekintések, értékelés, értékelések száma, kommentek száma
- Maximum 20 kapcsolódó videó azonosítója

Az adatok szöveges formában állnak rendelkezésre. Egy példasor:

```
hFFH8DaOHQg istothehalfabee 592 Music 286 1759
    4.45 539 244 hFFH8DaOHQg ZIo-7BBDaPo
    83SpuBijrBY…
```

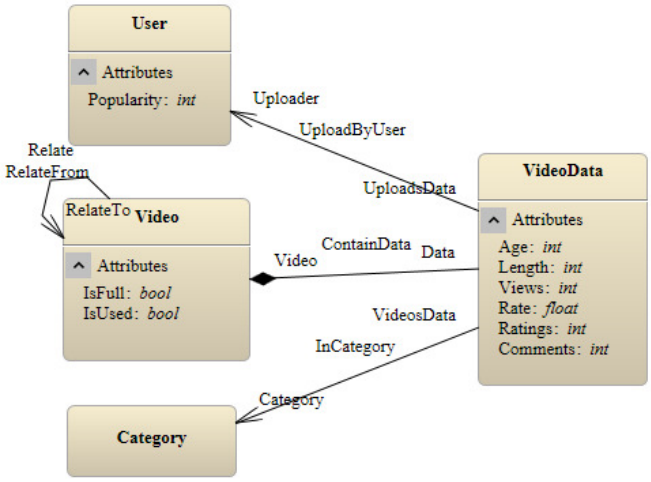
A félkövéren jelzett elemek a videó azonosítója, a feltöltő neve, feltöltési idő napokban, és a videó kategóriája. Az összes kapcsolódó videót nem soroltam fel, csak az első 3-at.

Készítettem egy egyszerű beolvasót, amely a szöveges fájlok alapján létrehozza a metamodell egy példányát, és feltölti a rendelkezésre álló adatokkal.

4.1.1. Létrehozott modell

[13] Az útmutatást köszönöm Gulyás Gábornak, a TMIT munkatársának.

A VMTS segítségével az alábbi metamodellt hoztam létre.



4. ábra. A készített metamodell

Az adatok beolvasása saját programmal történik, viszont a felhasznált adatszerkezet a VMTS által generált modell-leíró interfészek implementációján alapul. Így a beolvasott adatszerkezetet a VMTS módosítás nélkül tudja kezelni.

A metamodellben több típus szerepel, de egyértelműen a *Video* és a *VideoData* típusok dominálnak. Ez nem okoz problémát, hiszen a valós rendszerek nagy részében is van domináns típus, típusok.

> **Megjegyzés.** A *Video* és a *VideoData* szétválasztására azért volt szükség, mert az utolsó sorozatban érkező videók kapcsolódó videóinak csak kis részét szedték le a korábbi lépések. Azaz rengeteg videó van, amire egyszer vagy néhányszor utalunk, de semmit nem tudunk róluk. Ha a két típus a jelenlegi 1..1—0..1 multiplicitású kompozíció helyett egy egészet alkotna, akkor rengeteg üres videóelem létezne, ahol nem tudjuk az adatokat. A jelenlegi megoldásnak is nagy memóriaigénye van, ezt nem lenne célravezető tovább növelni.

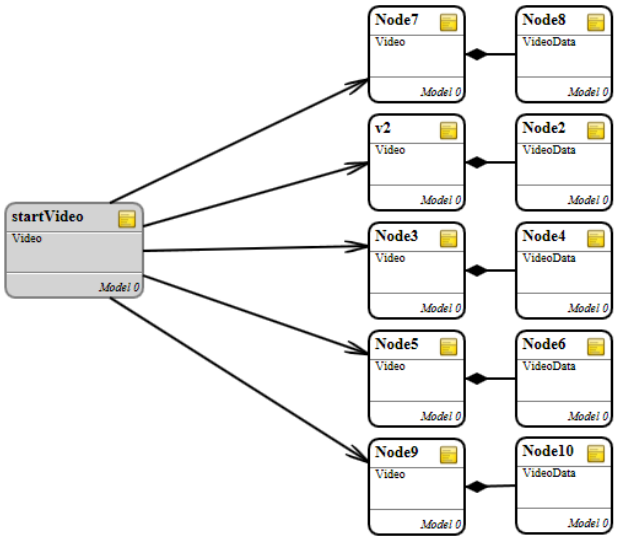
4.1.2. Az összeállított szabályok és transzformációk

Összesen 4 szabályt készítettem, majd mindegyik szabályhoz egy nagyon egyszerű transzformációt. A mérések során ezeket a transzformációkat használtam fel.

1. **YouTubePopular:** Olyan videók keresése, melyek nagy nézettségű videókra mutatnak
2. **YouTubePopular2:** Olyan videók keresése, amelyekre nagy nézettségű videók mutatnak
3. **YouTubePopular3:** Az előző szabály azzal a módosítással, hogy olyan nagy nézettségű videókat nézünk csak, melyre további nagy nézettségű videók utalnak (azaz a keresés mélysége a központi elemtől számítva 2)
4. **YouTubeBack:** A fenti transzformációk inverztranszformációja (azok ugyanis egy központi elemet módosítanak úgy, hogy a módosítás egyértelműen invertálható)

Az első transzformáció, mellyel a teszteléseket végeztem, a **YouTubePopular**, azokat a videókat keresi, melyek legalább 5 nagy nézettségű (1 millió megtekintés) videóra utalnak.

A keresett minta az alábbi módon néz ki:



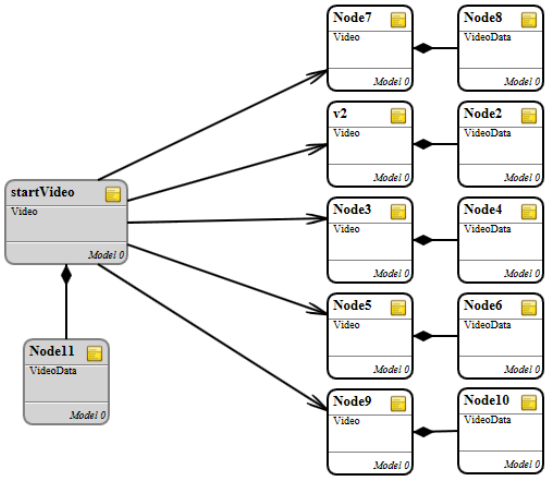
5. ábra. YouTubeTransformation szabálya

Szürkével jelölt a kiindulási elem, mert ott egyetlen attribútumot módosítok (*isUsed := true*), mely jelöli, hogy ez az elem már a találat részét képzi.

A szabály tulajdonságai:

- Viszonylag kis méretű (11 elem)
- Közepesen lassú a keresése (5 db 20-szoros multiplicitású éllel rendelkezik)
- A szabály párhuzamos (kvázi sztochasztikus) futtatása mindig ugyanazt az eredményt adja
    - A *startVideo* elem módosításra is kerül, ugyanakkor más elemek kapcsolódó videójaként (pl. *v2*) is tud szolgálni
    - A találatok több további közös elemet tartalmazhatnak, melyet csak olvasnak, de nem módosítanak.

A szabálynak létrehoztam egy *variánsát*, mely során újabb találatok keletkezhetnek a meglévők módosításával. Ekkor ugyanis a központi találati elem nézettségét módosítom úgy, hogy ez is teljesítse a szükséges kritériumokat. A módosított szabályba csak a szükséges *VideoData* elemet kellett felvenni, melynek módosítottam a nézettségi attribútumát:



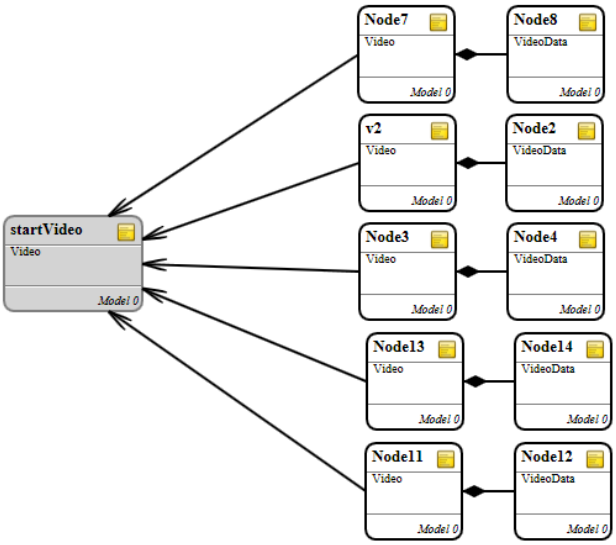
6. ábra. YouTubeTransformation – variáció a szabályra

A fenti szabály alapján létrehozott transzformáció a következő módon néz ki:

7. ábra. YouTubeTransformation

A transzformáció addig futtatja a szabályt, míg van találat (és módosítás), majd kilép. Bizonyos esetekben úgy módosítottam manuálisan az algoritmust, hogy az első találatig fusson.

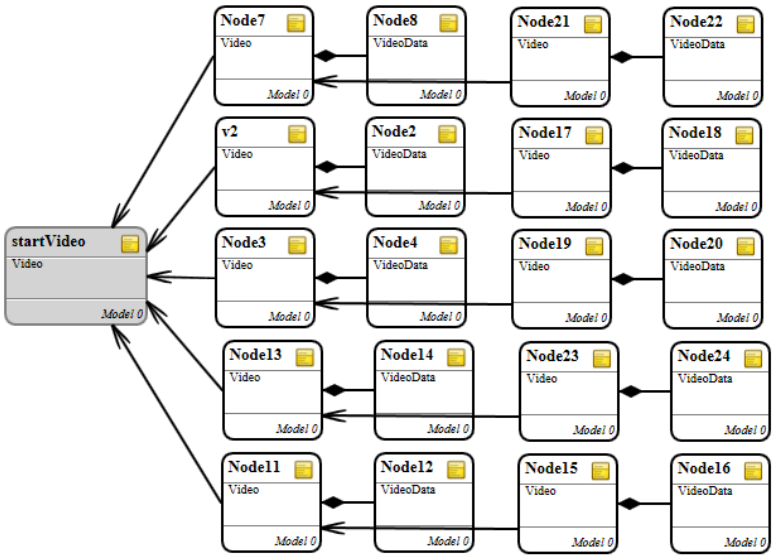
A **YouTubePopular2** az előző transzformációhoz nagyban hasonlít, a különbség annyi, hogy itt az ellentétes irányba vesszük a kapcsolatot, azaz olyan videókat keresünk, melyre sok nagy nézettségű videó mutat. (Itt a nagy nézettséget 100 000-nél határoztam meg.)



8. ábra. YouTubeTransformation2 szabálya

Ennek a transzformációnak is elkészítettem az előzőhöz hasonlóan a nézettség-módosító változatát.

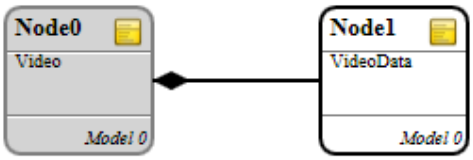
A **YouTubePopular3** transzformáció célja az volt, hogy egy nagyobb modellel nagyobb teljesítményigényű keresést tudjak végezni. Az izomorf részgráfkeresés NP-teljességéből, a 20-as multiplicitású élekből és a minta méretéből következően ennek a mintának a kereséséhez már jóval nagyobb időre volt szükség, mint a korábbiak esetében.



9. ábra. YouTubeTransformation3 szabálya

A kényszerek itt is azt tartalmazzák, hogy az egyes kapcsolódó videók nézettsége 1000 000 felett legyen. A kezelhetőség érdekében létrehoztam a szabály egy módosított változatát is, ahol a fenti öt kapcsolódó videóág helyett csak három van, viszont a kényszerek a központi elemtől távolabb már milliós nézettség helyett elfogadnak 100 000-t is. A szabály egyszerűsítése nélkül a futási idő túl hosszú lett volna.

A **YouTubeBack** szabály célja az volt, hogy az előbbi transzformációk után visszakaphassuk az eredeti rendszert. Egyszerűbb esetben csak a találat *isUsed* mezőjét kell visszaállítani (ld. ábra), továbbá elkészítettem a fenti variánsoknak megfelelő módosított változatokat is.



10. ábra. YouTubeBack szabálya

Ennek a szabálynak az a különlegessége, hogy nagyon egyszerű, gyorsan futtatható, és ütközés elvi szinten sem fordulhat benne elő.

5. Párhuzamosítási lehetőségek

A fejezetben először bemutatom a VMTS által generált kódot röviden, majd sorban az általam megvalósított módosítások lépésekeit, és a kapcsolódó mérési eredményeket. A fejezet végén ismertetem azt az algoritmust, amely a párhuzamosítást a lehető leghatékonyabban megvalósítja.

5.1. A jelenlegi megoldás

Mint már említettem, a gráftranszformációk alapvetően két részből állnak: transzformációs **szabályokból**, és magából a **transzformációból**. A transzformációk elkészítésekor a VMTS mindkettőhöz generál egy-egy C# forrásfájlt, amit új projektbe illeszt és fordít, így kapjuk meg a kész programot, ami majd megkeresi és elvégzi a transzformációt.

5.1.1. Szabályok

A jelenleg használt algoritmus találat esetén azonnal módosít is, azonban a párhuzamosítás során szükség van arra, hogy a keresést és a módosítást egymástól külön is elvégezhessük (ld. Függelék A.).

Az algoritmus a gráf élei alapján keresi a megoldást, a találatokat nem a gráf pontjai, azaz a modellelemek között, hanem a kapcsolatok alapján állítja fel. Mivel nem csak az elemek, hanem az élek is szigorúan típusosak, ez legalább olyan hatékony megoldást ad, mint a gráf pontjai alapján történő keresés. Természetesen elvégzi a szükséges ellenőrzéseket a *node*-ok kapcsán is, például hogy nincs-e ütközés, és hogy a találat kielégíti-e a külön definiált kényszereket. Tehát a mintaillesztés a minta egy adott élének kiválasztásával kezdődik, a modellben pedig DFS-jellegű keresést[14] végzünk az összes lehetséges kombinációra, amelyet a minta választott élének megfelelő típusú modellbeli élekből elérünk.

A jelenleg generált kódban a mintaillesztés logikai lépései a következők:

```
// First, outter for cycle
connetingRelationList = model.chosenTypeList; // List of items of given type
for (n = 0 to connetingRelationList.GetSize()) {
    possibleNext = connetingRelationList[n];
    if (ConstraintsFail(possibleNext) OR CollisionWithPrev(possibleNext))
        continue; // Constraints fail or repetition of a chosen node
    next = possibleNext; // Save
    connetingRelationList = next.chosenEdgeList; // Selecting next edgeList
    // Next embedded for cycle
}
```

Azaz a jelenlegi konstrukció minden mintaillesztésnél a kiindulásnak megfelelő típusú első elemtől kezdi a keresést, azaz a sokadik futtatásra általában rengeteg felesleges illesztési próbálkozást végez el, míg eljut a valós találatokhoz. Ez a megoldás ellenben biztosan helyes megoldást ad minden esetben, még akkor is, ha a minta újraírása során új találatok állnának elő.

[14] DFS (*Depth-First Search*): Mélységi keresés, a gráfban egy elemet egy fa útvonalú bejárással keresünk úgy, hogy addig megyünk végig egy ágon, míg levélig nem jutunk, és csak akkor fordulunk vissza. Ez a keresés hasonlóan történik, csak itt az adott mélységi szint az adott sorszámú mintaelemnek való megfeleltetést jelenti.

A fenti állítás természetesen nem általános érvényű, előfordulhat, hogy egy találatot úgy módosítunk, hogy az továbbra is találat maradjon sok futtatásig, ekkor ez az állítás nem állja meg a helyét. A fejezet végén látni fogjuk, hogy mikor éri meg az itt elkészített párhuzamosítást használni.

5.1.2. Általános elvárások a szabályok kapcsán

Az implementálás és a mérések során arra jutottam, hogy a transzformációs szabályok kapcsán a következő elvárásokat állíthatjuk fel:

- A szabály **ne hibázzon**
  Azaz ne adjon hamis találatot (*false positive*), vagy ne mulasszon el találatot (*false negative*).
- A szabály lehetőleg **gyorsan adjon találatot**
  - Akár „első találat", akár „összes találat" algoritmus szerint dolgozunk
  - A gyorsaság előfeltétele: **kerülje a felesleges újraellenőrzéseket**
    Természetesen minden módosításnak megvan az elméleti lehetősége, hogy új találatot hozzon létre (ennek a detektálásával most nem foglalkoztam). Ezért előfordulhat, hogy a már ellenőrzött részen új találat jön létre, de ez viszonylag ritkán fordul elő (heurisztikus megközelítés).
- A szabályt **lehessen párhuzamosan futtatni**
  A párhuzamos futtatás előfeltétele az, hogy az algoritmus keresési és a módosítási részét szétválasszuk.

5.1.3. Futtatási környezet

A teszteket egy kétmagos, 1,6 GHz-es gépen futtattam. Itt megérte két szálat futtatni, de hármat már többnyire nem, ahogyan az várható is volt. Felhasználtam továbbá egy nagy teljesítményű négymagos, 3,3 GHz-es számítógépet is. Itt a feladatokat értelemszerűen négy szálon futtattam.

A bemeneti adathalmaz 25 000, illetve 230 000 videót tartalmaz, előbbinél nagyságrendileg 500 000, utóbbi esetén kb. 4,5 millió relációs él található. A bemeneti változók sorrendjét véletlenszerűen határoztam meg egy egyszerű lineáris eltolásokból és maradékműveletekből álló algoritmus segítségével.[15]

5.2. Strukturális változtatások

A VMTS a transzformációkból és a szabályokból automatikusan kódot generál, amelyben a metamodellnek megfelelő bemeneti modellen megvalósítja a találatok megkeresését (*matching*), valamint a módosításokat (gráftranszformáció).

A VMTS jelenlegi verziója által generált kód tartalmaz *goto* utasítást. A *goto* kétségtelenül a leghatékonyabb megoldás bizonyos algoritmusok esetén, ezért az alacsonyszintű, illetve teljesítménykritikus feladatokban rendszerint használják is (a generált kód esetén a többszintű, egymásba ágyazott ciklusokból való kiugrásra). Ugyanakkor az első *goto* megjelenésének elvi

[15] Az algoritmus a bemenetet várhatóan egyenletesen osztja el a teljes bemeneti téren, így a jellemző, indexhez köthető csoportokat felbontja

jelentősége van, mert ez egy nyilvánvaló választás a teljesítmény mellett akár az átláthatóság és módosíthatóság rovására. Így a *goto* előnyére is válhat egy optimalizált generált algoritmusnak, de amennyiben az algoritmus módosításra kerül (ld. most), a fejlesztő első lépése a meglévő megoldás átstrukturálása, a *goto*-k eltávolítása. *Kérdés viszont, hogy a program strukturálása mennyiben rontja a teljesítményt.*

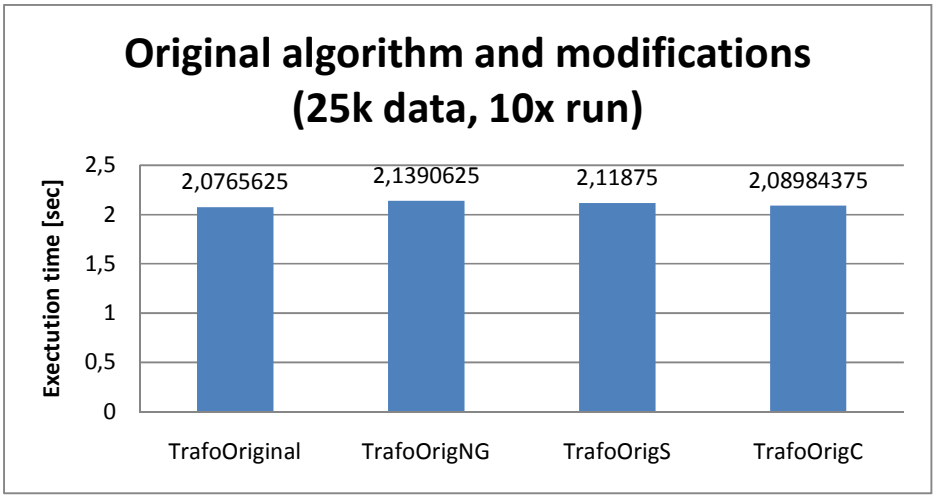
Mindezek ellenére egy későbbi algoritmusba kénytelen voltam újra a *goto* eszközéhez nyúlni.

Kiindulási algoritmusnak a YouTube videókat tartalmazó modellben a népszerű kapcsolattal bíró videók kiszűrésének a szabályát vettem. A transzformáció megkeresi az összes, szabálynak megfelelő találatot és egy apró módosítással jelöli meg a találatot (és így akadályozza meg az újbóli megtalálást).

A következő transzformáció-verziókat készítettem el és hasonlítottam össze:

1. **TrafoOrig:** Eredeti algoritmus és transzformáció
2. **TrafoOrigNG:** Eredeti algoritmus *goto* utasítások nélkül
3. **TrafoOrigS:** Eredeti algoritmus strukturálva, a keresési adatok és algoritmus külön struktúrába kiemelve
4. **TrafoOrigC:** Eredeti algoritmus strukturálva, a keresési adatok és algoritmus külön osztályba kiemelve

Jellemző futási eredmények:



11. ábra. Strukturálási költségek

Összességében azt mondhatjuk, hogy a legkönnyebben használható, Trafo*OrigC* néven azonosított strukturált megoldás nem okoz lényegi teljesítményvesztést. Innentől ezt a megoldást, illetve ennek a módosított verzióit használtam a munkám során.

Megjegyzés. A strukturálás célja a gyors, flexibilis fejlesztés. A végleges verzió visszaalakítható, ha szükséges.

Természetesen ez a megoldás még csak egy szálon fut az eredetihez teljesen hasonló módon. Az algoritmus minden találat esetén elejéről kezdi az illesztést. Ez lesz egy következő optimalizálási lépés.

A módosított transzformáció vázlata:

```
class TrafoOrigC : TransformationBase
{
	private AlgC alg;

	// Algorithm execution starts here
	public override void Execute()
	{
		MatchCnt = 0;
		while (alg.StartFind_0(0, Model_0.Relate.Count)) ; // Continues while there is match
	}

	// Called by alg if one match found
	private bool OnMatch(AlgC match)
	{
		MatchCnt++;
		match.Modify();
		return false; // Halt execution -> restart from beginning
	}
}
```

A *matcher* algoritmus:

```
class AlgC
{
	// Parts in match
	#region Match elements
	public YouTube2.YouTubeModel.VideoData Node10;
	//...
	#endregion

	// Starts seeking with given indices in the list
	public bool StartFind_0(int fromIndex, int toIndex)
	{
		bool hasMatch = false;
		for (int n_0 = fromIndex; n_0 < toIndex; n_0++)
		{
			// … many for cycles in each other
			// Success
			hasMatch = true;
			if (!Callback(this)) // -> TrafoOrigC.OnMatch
				return true;
			// Otherwise continue
		}
		return hasMatch;
	}

	// Basic modification
	public void Modify()
	{
		startVideo.Attributes.IsUsed = true;
	}

}
```

Ez a konstrukció a *callback* lehetőségének köszönhetően kellőképpen rugalmasnak és hatékonynak bizonyult, a későbbiekben is ezt használtam több-kevesebb változtatással.

5.3. Mintaillesztés párhuzamosítása

Az alábbiakban összehasonlítom az egyszeres találatot és az összes találatot kereső algoritmusok teljesítményét. Ennél az összehasonlításnál a modellben nem végzek semmilyen módosítást. Ezek a mérések a kétmagos gépen születtek.

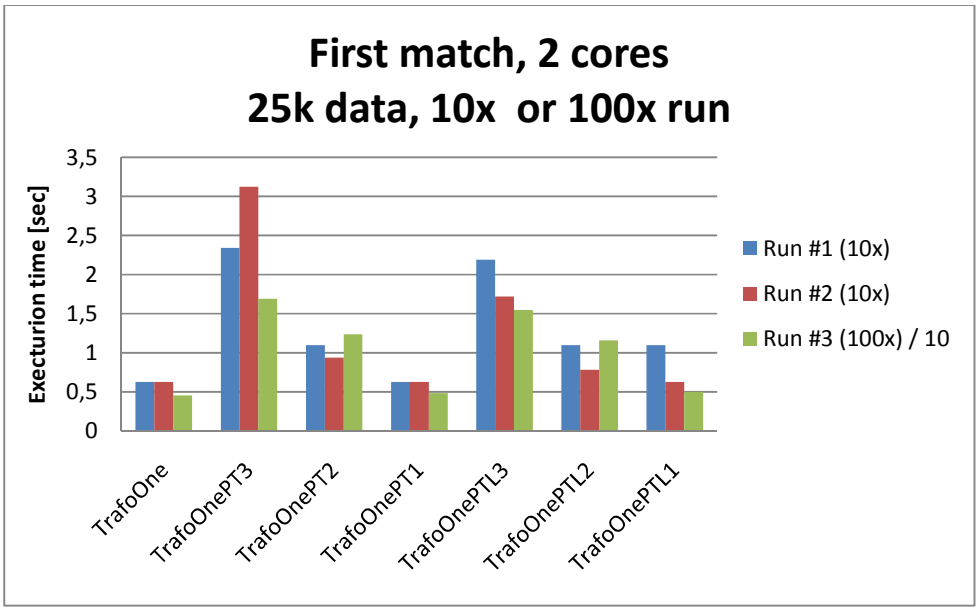
5.3.1. Első találat keresése

A kereső algoritmus egy, az LHS-nek megfelelő példányosítást keres a bemeneti modellben, amely kielégíti a megadott további feltételeket is.

Itt gyakoriak a **YoutubeTrafo** számára megfelelő találati lehetőségek, így az első találat lokalizálása viszonylag hamar megtörténik. Az a kérdés, hogy az új szálak indítása, leállítása és bevárása, a szinkronizáció és minden további pluszmunka használata vajon megéri-e, azaz nyerünk-e annyi időt az effektív munka során, ami a járulékos költségeket fedezné.

Az alábbi algoritmusokat hasonlítottam össze, mindegyik az előző fejezetben bemutatott *AlgC* algoritmust használja.

- TrafoOne: Egyszálú futtatás
- TrafoOnePT: Többszálú futtatás, a név után jelzem a használt szálak számát (3, 2, 1)
- TrafoOnePTL: Többszálú futtatás szigorúbb szálbiztonsággal, az előzőhöz hasonlóan itt is változtatható a használt szálak száma (3, 2, 1)



12. ábra. Első találat keresése

Meglepő módon az *egyszálú algoritmus volt a leghatékonyabb*, ezt követték a többszálú megoldások egy szállal futtatva. Ennek oka az, hogy a szálkezelés nem hatékony, valamint a keresések *túl hamar* találják meg az első találatot (még mielőtt a második processzormag igazán hatékonnyá tudna válni).

A három szálas futtatás azért teljesített ilyen gyengén, mert a harmadik szál indulásakor az első szál gyakran már eljutott a találatig, sőt, egyes esetekben már le is állt.

Ezeknek a párhuzamos algoritmusoknak az egyik legnagyobb hibája, hogy a *Thread.Abort()* függvénnyel szakítják meg a szálak végrehajtását. A *Thread.Abort()* meghívásakor a futtatott szálban a keretrendszer egy *ThreadAbortedException* kivételt dob, ami végső soron a szál leállását okozza. A kivételkezelés teljesítménykritikus környezetben lassú és kerülendő megoldás.

Ugyanakkor a szál leállását *mindenképp* meg kell várni, hiszen anélkül nem haladhat tovább a feldolgozás. A főszálon nem folytathatjuk a feldolgozást mindaddig, amíg az összes kereső szál le nem állt.

***

A párhuzamosítás jelen formájában first match *keresésekor nem éri meg.* Milyen módosításokkal lehetne növelni a rendszer hatékonyságát, hogy mégis megérje?

5.3.2. „Kegyes megszakítás" és *ThreadPool*

A következő lépésben az *Abort()* helyett a „kegyes megszakítást" alkalmazom („*graceful abort*"), azaz a szálat nem kívülről állítom le, hanem az algoritmusba építek bele egy ellenőrzési lépést, amely vizsgálja, hogy folytatható-e a végrehajtás. Az algoritmus a következő módon működik:

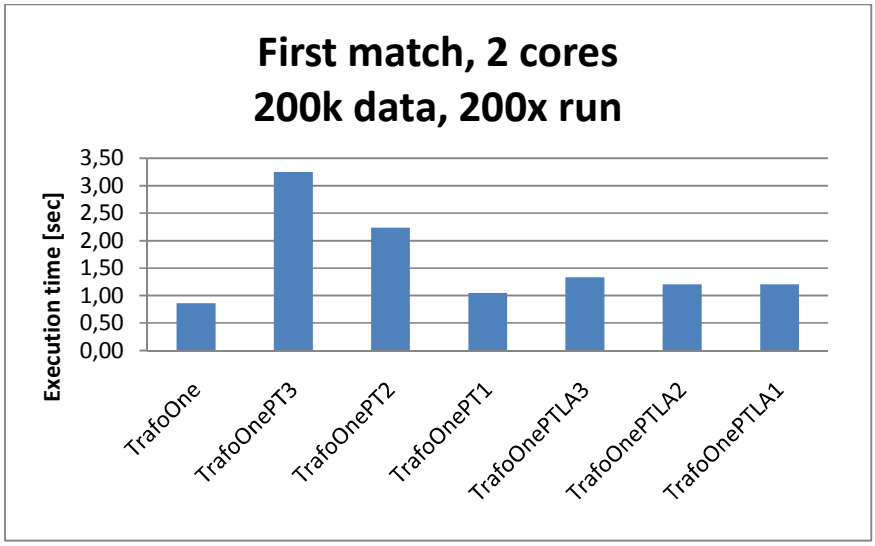
```
class AlgCA
{
  public class SyncObject
  {
      private bool sync = false;
      public bool Value {
            get { lock (this) return sync; }
            set { lock (this) { sync = value; } }
      }
  }
  private SyncObject isAborting;

  public bool StartFind_0(int fromIndex, int toIndex) {
      bool hasMatch = false;
      for (int n_0 = fromIndex; n_0 < toIndex; n_0++) {
            if (isAborting.Value)
                  return hasMatch;

            // Continue algorithm
            // ...
                  // Success
                  hasMatch = true;
                  if (Callback(this) == false)
                        return true;
            // ...
      }
  }
}
```

A kegyes megszakítás használatával jelentős teljesítménynövelést sikerült elérnem. Az alábbi diagramok a teljes futási időt mutatják 100-szoros ismétléssel. A versenyző megvalósítások az előző

részben bemutatott *TrafoOne* és *TrafoOnePT*, az új transzformáció a *TrafoOnePTLA* 3, 2 és 1 szállal futtatva.

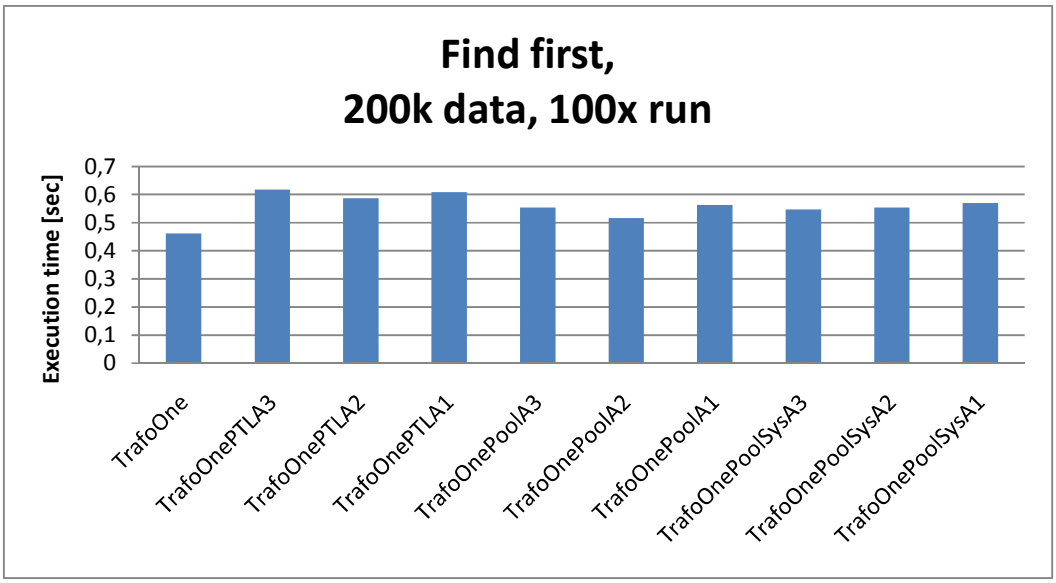


13. ábra. Első találat keresése – fejlesztve

A kegyes megszakítással sokkal jobb eredményt tudunk elérni.

> **Megjegyzés.** Az algoritmus rendelkezik egy nagy gyengeséggel. Nem kifizetődő túl gyakran vizsgálni a szinkronizációs objektumot, de ha túl ritkán tesszük, akkor az algoritmus hosszabb ideig fog feleslegesen dolgozni, míg felismeri, hogy nem kéne tovább futnia. Bár ez csak akkor kérdés, ha a futást meg akarjuk szakítani.

Az algoritmuson végeztem még egy fejlesztést: ne hozzunk létre minden futtatáskor új szálat, hanem vegyük a szálakat egy ***ThreadPool***-ból! A szálak kezelésére használtam a beépített *ThreadPool* osztályt (*TrafoOnePoolSysA#*), de egy saját megvalósítást is implementáltam (*TrafoOnePoolA#*).



14. ábra. Első találat – fejlesztve 2.

Összességében a komolyabb szálkezeléssel valamennyit sikerült javítani az algoritmusok futási idején. Látható az, hogy a kétmagos tesztelő környezetben a kétszálú megoldások egy kicsivel jobbak lennének az egy vagy három szálat használóknál, de a különbség nem jelentős. Láthatjuk azt is, hogy a saját szálkezelés a beépített szálkezelésnél talán egy kicsit jobban teljesít, de a különbség itt sem jelentős.

Összességében elmondható, hogy ha az algoritmus viszonylag hamar jut találathoz, akkor a többszálú megoldások még mindig gyengébben teljesítenek az egyszálúnál, viszont már nem nagy a futásidő különbsége. Ne felejtsük el azt, hogy ezek a teljesítményvesztések abszolút mércével nézve kicsik, hiszen igen gyors futásokról van szó! Itt végig olyan algoritmust vizsgáltunk, ami az egyszálú megoldásnak kedvez, hiszen alig jut idő a többszálú futás teljesítményének a kiaknázására.

5.3.3. Összes találat keresése

Most vizsgáljuk meg azt az opciót, amikor még mindig nincs módosítás, de az összes lehetséges találatot megkeressük, azaz kimerítő keresést végzünk. A talált mintákat ezek az algoritmusok még nem mentik el, éppen ezért jó becslést kaphatunk arra nézve, hogy milyen hatékony az első elem párhuzamos keresése, ha az első találatra a keresés legvégén kerül sor.

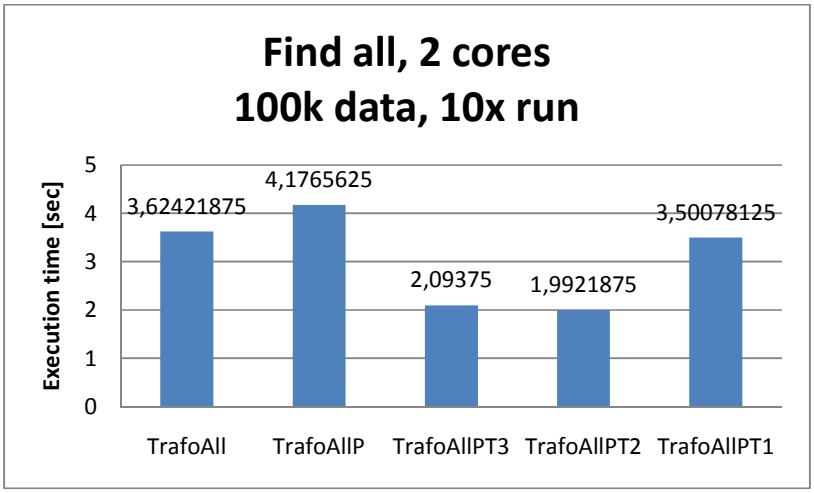
> **Megjegyzés.** Az ütközések miatt nem biztos, hogy minden így megtalált *match* valóban módosításra fog kerülni, továbbá az sem garantált, hogy a transzformációk elvégzése nem hoz létre újabb találatokat.

Az összehasonlított algoritmusok:

- TrafoAll: Egyszálú megoldás
- TrafoAllP: A *Parallel.For()* hívást használva,[16] a kezdőpontokat egyenként vizsgálva
- TrafoAllPT3: Egyszerű párhuzamos végrehajtás 3, 2 és 1 szálon.

Ezeknél az algoritmusoknál nem volt szükség megszakításokra, hagytam minden szálat végigfutni, az kezdőpontokat kiosztó objektum úgyis jelez, ha már elfogytak a vizsgálandó elemek. Valamint nem láttam nélkülözhetetlennek a *ThreadPool* használatát, hiszen itt a futási idő nyeresége messze kárpótolja a szálindítás költségeit, ez anélkül is jó közelítés.

[16] A *Parallel.For()* a C# egyik beépített magas szintű megoldása a párhuzamosításra. Lényegében egy tipikus *for* ciklus párhuzamos megvalósítását teszi lehetővé, ahol az egyes ciklusmagok külön szálban futnak, futhatnak.



15. ábra. Összes találat keresése

A fenti ábrából számos következtetés vonható le.

- Nagy modellek és hosszas keresés esetén megéri a párhuzamosítást használni.
  - A párhuzamos futtatás közel fele annyi idő alatt végez
    - A szinkronizációs és egyéb pluszköltségek azért a kétszeres sebességet nem sikerült elérni
- A *Parallel.For()* megoldás a jelen helyzetre nem hatékony
  - A probléma a kezdeti elemek terének partícionálásával áll kapcsolatban. Nem hatékony, ha egyesével osztjuk ki a keresendő elemeket a szálaknak
- Az egyszálú és az egy szálon futó párhuzamos algoritmusok futási ideje nagyjából megegyezik
  - Azaz a szálindítás költsége a kereséshez képest kicsi
  - Van mérési hiba, hiszen az egyszálú algoritmusnak kéne gyorsabbnak lennie, mégsem az. A hiba talán a cache kapcsán keresendő[17]

5.3.4. A mintaillesztés tapasztalatai

A cél továbbra is egy olyan szabályszintű párhuzamosítás kidolgozása, amelyben a keresés több szálon fut, majd ezután az eredményeket összegezve történik a gráftranszformáció. A keresés fázisával kapcsolatban a következő kritériumokat tudjuk megfogalmazni:

- A keresés több szálon zajlik
- Az eddig megismert megoldások (első sorban *TrafoOnePoolA*) képezik az algoritmus alapját
- A keresés a mintaillesztés legalsó szintjén vizsgálja a külső leállítást (*graceful abort*)
- Saját *ThreadPool* biztosítja a szükséges szálakat

A modell kapcsán az alábbi kritériumoknak szükséges teljesülniük, hogy az algoritmus valóban hatékony legyen:

[17] Bár tulajdonképp a cache is erősen kérdéses, mert a két futtatási sorozat egy programban, egymás után történt, így a második *TrafoAll* futásakor már minden cache-folyamat lezajlott egyszer. Az eltérés viszont elég kicsi, úgyhogy mérési hibának tekintem.

1. Nagy modellel dolgozunk
2. Az elvégzendő módosítások nem túl nagyok
3. A transzformációkban sok az összes találatot kereső (kimerítő) szabály és kevés az egyszeres
4. Gondoskodni kell a találatok mentéséről és tárolásáról
5. Gondoskodni kell a találatok konfliktusainak kezeléséről

Az utóbbi két kritériumot a következő részben tárgyalom részletesen, és a harmadik pont kapcsán is lesz még észrevételem.

5.4. Mintaillesztés és módosítás párhuzamosítása

Ahhoz, hogy a teljes transzformációs folyamatra érvényes párhuzamosítást dolgozzak ki, a következő kérdésekre kell válaszolni:

- Hogyan tároljuk a találatokat, míg sor kerül a módosításra?
- Hogyan kezeljük a konfliktusban lévő találatokat?

5.4.1. A találatok tárolása

A párhuzamos keresésektől kapott találatok mentése az algoritmus sarkalatos pontja. Ha túl sok adatot kell menteni, akkor az algoritmus nem lesz hatékony a memóriakezelést illetően. Ugyanakkor cél, hogy az egyszer már megtalált minták további adatfeldolgozás nélkül is rendelkezésre álljanak.

A találatok tárolására két fő megoldást látok. Tárolhatjuk a *teljes illesztést*, viszont ennek kétségkívül nagy a helyigénye. Ha a találatokban sok olyan elem van, amit nem érintenek a módosítások, akkor jobban megéri egy sokkal célirányosabb tárolási mintát használni: tárolhatunk olyan *speciális adatszerkezeteket*, melyek csak a *módosítandó elemeket* és a *módosításhoz szükséges adatokat* tárolják. Ez a megoldás természetesen nem minden esetben vezet jobb memóriakezelésre, de sok esetben kétség nélkül megéri használni. A dolgozatomban ezért ez utóbbi a megoldást használtam fel.

5.4.2. A konfliktusok kezelése

A párhuzamosítás legnagyobb feladata a konfliktusban lévő találatok kezelése. Két találat konfliktusban van, ha különböző sorrendben történő futtatásuk eltérő eredményre vezet. (Részletesebben ld. (Ehrig, Ehrig, Prange, & Taentzer, 2006)).

Ha teljes találati mintát mentünk a keresés alatt, akkor megtehetjük, hogy utólag, a *módosítás fázisában* ellenőrizzük az esetleges konfliktusokat. Ehhez viszont a teljes találat mentése kell és így is számításigényes, valamint a keresés sok felesleges találatot adhat. A másik lehetőség, hogy még a *keresés közben* figyelünk arra, hogy ne foglaljuk le olyan elemet, amely már szerepel egy találatban. Itt értelemszerűen a szálközi kommunikáció és a több feltételvizsgálat miatt lesz gyengébb a teljesítmény. Az általam vizsgált modell esetében mindenképp ez utóbbit érdemes használni, mert a transzformációk túl sok konfliktusos találatot adnának. A konfliktusos találatok mentése jelen esetben nem éri meg – talán általában sem – de ezt a kérdéskört mélyebben nem vizsgáltam.

A keresés közbeni találatfoglalásra is több stratégia létezik. A **pesszimista zárolás** szerint amint találunk egy, a keresésnek megfelelő elemet, azonnal le is foglaljuk. A gyakorlati megvalósítással több probléma akad. Létrejöhet olyan *erőforrás-foglalás*, mely során ugyanannak a találatnak a két felét két külön szál zárolja. Ekkor előfordulhat, hogy mindkét szál feladja a keresést, és *hamis negatív* eredményt kapunk – pedig van még egy találat a gráfban.

Másrészt a rengeteg szálbiztos elemfoglalás és felszabadítás tetemes pluszköltséggel jár. Összességében ezt a lehetőséget elvetettem.

> **Megjegyzés.** Lefoglalás alatt a *gráf pontjainak* a lefoglalását értem. Az éleket nem szükséges külön lefoglalni. Ha a transzformációkban sehol sem használunk olyan élt, melynek nincs mindkét vége a találatban – ez eleve csak imperatív kódból lenne lehetséges – akkor az élekkel külön nem kell foglalkozni.

Az **optimista keresés** csak akkor foglalja le a találat elemeit, ha a találatnak minden része rendelkezésre áll. Így nagyban csökken a szálbiztos műveletek száma, a futás gyorsabb lesz. Mivel a modell mérete több nagyságrenddel nagyobb az egy szál által épp lefoglalt elemek számánál, kicsi az esélye, hogy két szál egymást átfedő találaton végezné a keresést. A korábban megtalált elemeket pedig az algoritmus külön megjelöli, így a *konfliktus lehetőségét* már keresés közben elkerülhetjük. Csak olyan találatokat mentünk el, melyek *biztosan nincsenek konfliktusban*. Így a módosítások végrehajtása során már nincs szükség ellenőrzésre.

A találatok jelölését read-write foglalással lehet még fejleszteni. Minden *node*-hoz hozzáadunk két tulajdonságot:

- **Írási foglalás:** Jelzi, ha a *node* bármilyen szinten változik (módosul, törlésre kerül, éleket adunk hozzá vagy törlünk)
- **Olvasási foglalás:** Azt mutatja, hogy az adott *node* csak adatforrás, vagy a találat egy eleme egyéb funkció nélkül, de nem módosul

Minden keresés—módosítás körhöz egyedi azonosítót rendelünk, amit minden kör elteltével inkrementálunk. A keresés során figyelmen kívül hagyjuk azokat a foglalásokat, melyek a jelenlegi körnél használt azonosítónál korábbit tartalmaznak – az ezekhez tartozó módosítások ugyanis már megtörténtek, mi már úgyis a végeredményt látjuk. Így a foglalások felszabadításával nem kell külön törődnünk. Ez a találatok tárolása kapcsán fontos szempont, nem szükséges a teljes *match* mentése.

A foglalások költsége a jelenlegi alkalmazásnál 8 bájt, de ezt le lehet szorítani akár 2 bájtra is. Ekkor 255 transzformáció futtatása után végig kell menni a gráf összes pontján és nullázni kell a számlálókat. Ez a tárhely- és időmennyiség a transzformáció többi része mellett jó eséllyel nem számottevő, de általában a 8 bájt is bőven használható.

> **Megjegyzés.** Ennél az algoritmusnál újra előkerült a korábban részletezett *goto* parancs. Tagadhatatlanul nem szép, de bizonyos feladatokra egyszerűen nincs jobb.

5.4.3. Mérési eredmények

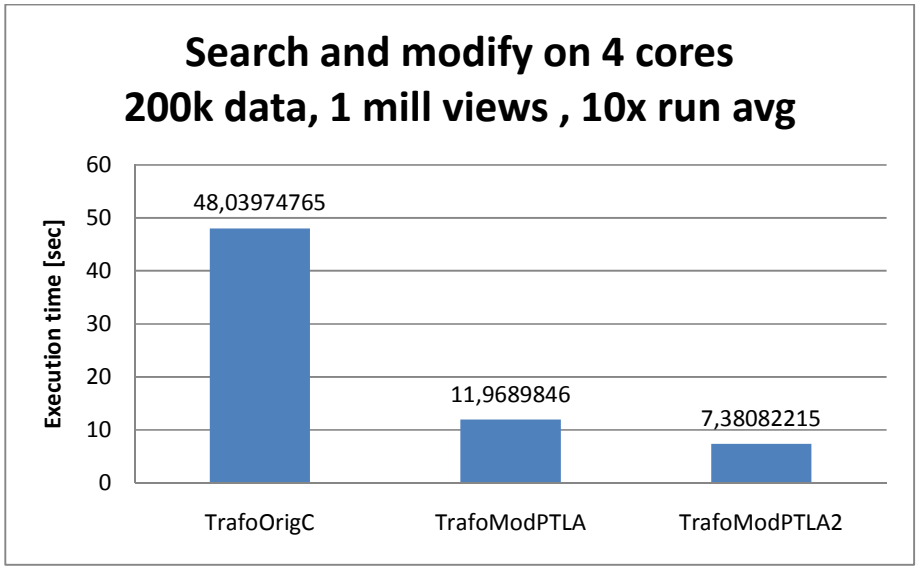
A lényegi összehasonlítás előtt fontosnak tartom leszögezni, hogy a kimerítő keresést végző módosító algoritmusok, amelyeket készítettem, mindig jobb – és általában nagyságrendileg jobb – eredményt adtak, mint a VMTS által jelenleg használt algoritmus. Ennek a fő oka az, hogy a mostani algoritmus minden találat után a legelső elemtől kezdi újra a keresést, ami értelemszerűen nem jó stratégia, mert újra át kell fésülnie azt a területet, ahol már megtalálta a lehetséges találatokat. Új találatok ugyan keletkezhetnek, de a találati arány ezzel együtt is alacsonyabb lesz, mint a nem vizsgált régiókban.[18]

Az **első transzformáció** (YouTubeTrafo) futtatásának eredményei az alábbiak. Azokat a videókat kerestem, amik legalább 5 külön egymilliós nézettségű videóra mutatnak. Itt nagyságrendileg 300 találat található a kb. 230 000 videó adataiból.

A használt algoritmusok:

- TrafoOrig: Eredeti megoldás, egyszálú futás
- TrafoModPTLA1: Többszálú megoldás egy szálon futtatva
- TrafoModPTLA4: Többszálú megoldás négy szálon futtatva

A négymagos gépen a következő futási eredményeket kaptam.



16. ábra. Keresés és módosítás

Látható, hogy a használt algoritmus lényegesen jobb teljesítményt nyújt, mint az eredeti. Viszont a teljesítménynövekedés első sorban nem a párhuzamosításnak tudható be, hiszen a párhuzamos algoritmus egyszálú végrehajtása is lényegesen gyorsabb.

***

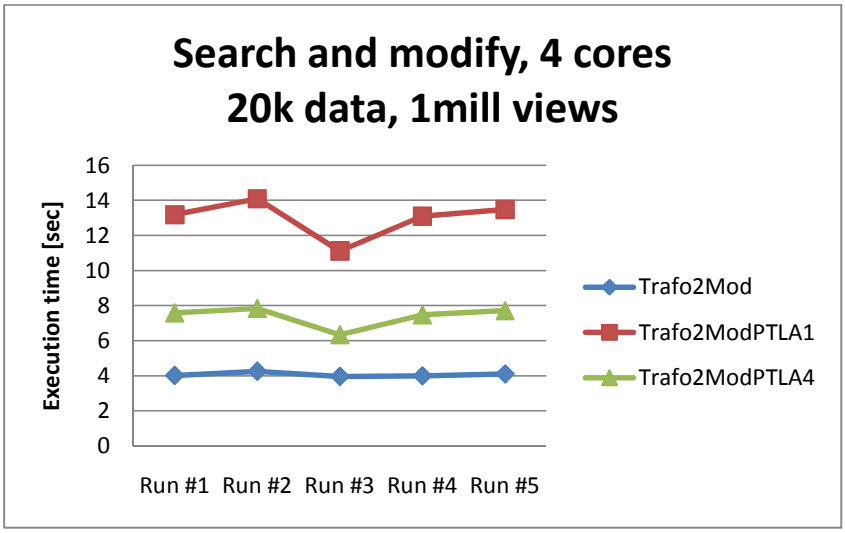
[18] A precizitás kedvéért: ez persze sztochasztikusan igaz, már amennyiben az elemek eloszlása és az új elemek keletkezése egyenletes a találati térben, amit persze nem garantált. Gyakorlati oldalról azonban szinte mindig így van.

A **második transzformáció** (YouTubeTrafo2) az elsőhöz hasonlóan néz ki, de a kapcsolatok az ellentétes irányba mutatnak. Ez jelentősen változtatott a futási eredményeken, az eltérő multiplicitások lehetnek a különbség okai. (Egy videóra tetszőleges számú utalást mutathat, de egy videó maximum 20 másikra utalhat.)

Az összehasonlításból kihagytam az eredeti algoritmust, mert az itt is egy nagyságrenddel rosszabbul teljesít. A felhasznált algoritmusok:

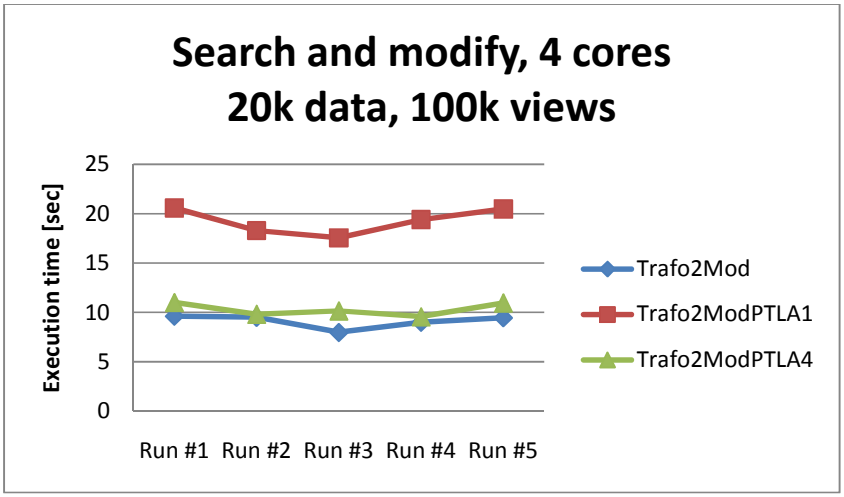
- **Trafo2Mod:** Egyszálú megoldás, módosított keresési stratégia
- **Trafo2ModPTLA:** Többszálú megoldás 1 vagy 4 szálon futtatva

A keresést két variációban is elvégeztem: a kényszerekben a nézettségi határt egymilliónál, illetve százezer megtekintésnél húztam meg.



17. ábra. Keresés és módosítás – összehasonlítás

Az előző 1 milliós nézettségű kapcsolódó videókat keresett. Kicsit könnyebb feltételek mellett (100 000 megtekintés), ahol lényegesen több találat születik, ugyanez az algoritmus:



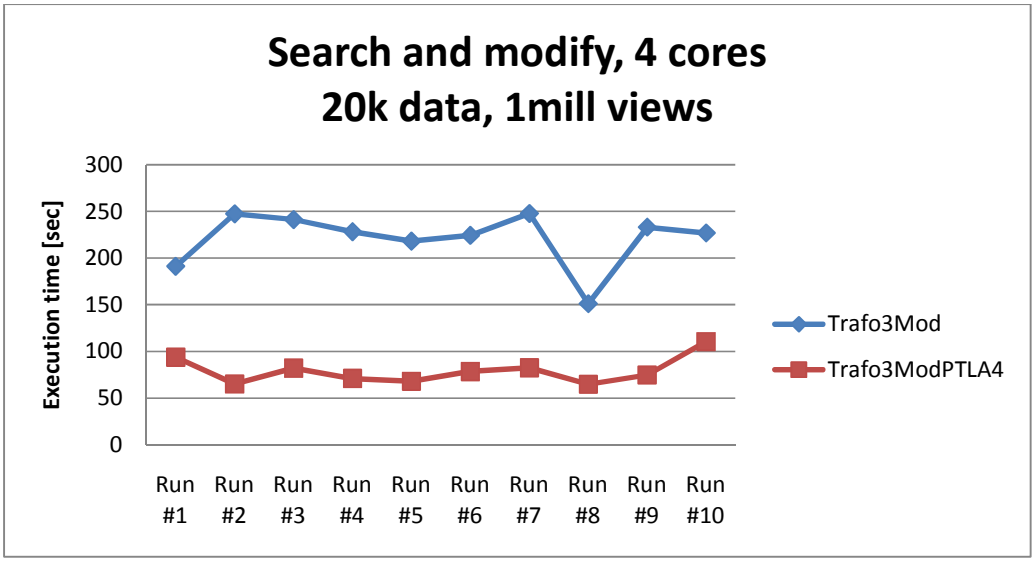
18. ábra. Keresés és módosítás – összehasonlítás 2.

Azt látjuk, hogy az egyszálú megvalósítás a leggyorsabb. De ahogy nő a számítási igény, és így a futási idő, úgy kezdi beérni a valóban többszálú megvalósítás, az utóbbi esetben az egyszálú algoritmus már csak nagyjából 10%-kal gyorsabb. A többszálú algoritmus egyszálú futását is ábrázoltam az összehasonlítás kedvéért. Értelemszerűen ez lesz a leglassabb, mert csak egy szálon fut, és a szinkronizációs megoldások még felesleges pluszmunkát is eredményeznek. Innentől ezt a megoldást már nem fogom külön mérni.

Felmerül a kérdés, hogy mikor érdemes foglalkozni a párhuzamosítás kérdésével. Van, amikor a módosított egyszálú megoldás lényegesen jobban teljesít, mint a párhuzamos végrehajtás.

***

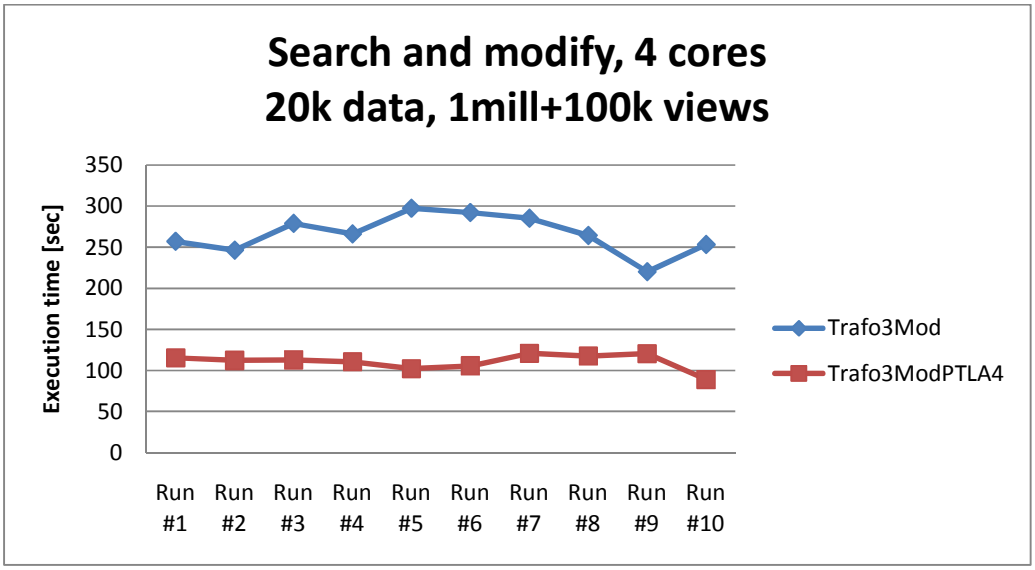
A **harmadik transzformáció** (YouTubeTransformation3) már sokkal nagyobb keresési mintával dolgozik, mint a korábbi transzformációk – és ez a futásidejében is megmutatkozik. Az első futtatásnál az adott videóra mutató egymilliós nézettségű videókat keresek, melyekre további egy-egy milliós nézettségű videó mutat. A következő eredményeket kaptam.



19. ábra. Nagy teljesítményű keresés és módosítás

Ez az a pont, ahol azt mondhatom, hogy *a párhuzamosítás alkalmazása látványosan növeli a teljesítményt*. A négy szálon dolgozó algoritmus messze jobban teljesít, mint az egyszálú.

A mérés komoly hibája ugyanakkor, hogy nincsen találat, mert a feltételek túl szigorúnak minősültek. Kicsit enyhébb kényszereket határoztam meg és a transzformációt módosítottam. Ennél a variánsnál a gyökérelem már csak három darab, két *Video* elemből álló ágat tartalmaz, valamint a távoli levelekben a nézettségi korlátot 100 000 főben állapítottam meg a korábbi millió helyett. Így a transzformáció már több találatot ad. A futási idők:



20. ábra. Nagy teljesítményű keresés és módosítás 2.

Itt is az előzőhöz hasonló gyorsulást tapasztalunk, az egyszálú algoritmus futásideje a négyszálúnak kb. 250%-a. Itt már van találat – úgyhogy van módosítás is – ugyanakkor a találatok száma a kereséshez képest továbbra is elenyésző.

***

A mérési eredmények elég beszédesek, nyilvánvalóan mutatják, hogy a párhuzamosításnak megvan a maga létjogosultsága. A párhuzamosítás akkor éri meg a leginkább, amikor a legnagyobb szükség van rá: ha nagy modellekben kell nagy teljesítményű kereséseket végezni. Láttuk, hogy kis keresési minták esetén nem érdemes használni, de ha a megfelelő környezetben alkalmazzák, akkor a párhuzamos futtatás 2-3-szorosára tudja növelni a teljesítményt. A megfigyelések szerint kb. *10 másodperces futási idő felett* már megérte használni a párhuzamosítást.

Az algoritmus jelen formájában már alkalmazható a VMTS keretrendszerben, természetesen további mérések pontosíthatják az eredményeket. Az alkalmazáshoz olyan heurisztikára van szükség, amely jó becslést ad arra, hogy érdemes-e párhuzamosított kódot generálni – azaz a futási időt kell tudni jól megbecsülni.

6. További optimalizálás

Az előző fejezetekben láthattuk, hogy a párhuzamosítás bizonyos esetekben hatékony. Láttuk viszont azt is, hogy a párhuzamosítás eszközkészlete is korlátozott, nem mindig használható. Vizsgáljuk meg azokat a további módosításokat, amelyek segítségével a modelltranszformáció teljesítménye jelentősen javítható!

6.1. Programozási környezet

Az első és legfontosabb fejlesztési lehetőség a **natív nyelvekre való átállás**. Köztudott, hogy az interpretált, illetve JIT-fordított nyelvek nem érik el a natív nyelvek teljesítményét, különösen az algoritmizálási feladatokban – és ez persze irreális elvárás is volna.

Ha valóban kritikus a modelltranszformációk teljesítménye, akkor a transzformációt generáló szkripteket kell átírni, hogy ne C#, hanem C++ kódot írjanak. Ez önmagában nem túl nagy feladat, mert a két nyelv (különösen a generált kód vonatkozásában) elég közel áll egymáshoz. Ez viszont felveti annak a kérdését, hogy hogyan lehet majd a C++-os modelltranszformáció kódját integrálni a VMTS jelenlegi struktúrájába integrálni. A következő kérdések merülnek fel.

- C++ kód előállítása (szkript átírása)
- C++ kód fordítása
- Lefordított program integrálása és futtatása
- Biztonsági kérdések

C++ alatt a biztonságra fokozottan kell ügyelnünk, úgyhogy a kényszerek ellenőrzésénél is illene egy ellenőrző szakaszt beiktatni. Ráadásul azt szinte lehetetlen kivédeni, hogy a felhasználó ne tudjon veszélyes futásidejű hibákat generálni. Mindezekkel együtt a natív környezet lenne a legnagyobb hatással a teljesítményre.

6.2. Algoritmusgenerálás

A megvalósított algoritmusok teljesítményében a legnagyobb előrelépést nem a párhuzamosítás jelentette, hanem a **szabályok végrehajtásának módosított stratégiája**.

A korábbi rendszer mindig az első elemtől kezdte a következő találat keresését, a mostani egyszálú és párhuzamos végrehajtások pedig sorfolytonosan haladnak a lehetséges kezdőpontokon, míg van olyan elem, amit az adott körben módosítottak. Ez a módosítás önmagában ugrásszerű teljesítménynövekedést okozott, mindenképp érdemes lenne a későbbiekben a VMTS kódgenerálásába beépíteni.

Nem foglalkoztam a **keresési algoritmus összeállításának kérdésével**, de a téma bizonyára rejteget még számtalan lehetőséget. A VMTS részben használ heurisztikákat, amikkel optimális bejárási stratégiát igyekszik kialakítani, de a lehetőségek közel sincsenek kihasználva. A heurisztikák kapcsán, mint általában, itt is fontos mérlegelni: mennyi erőfeszítést ér meg egy-egy fejlesztés?

6.3. Modellstruktúra

Külön foglalkoztam az **egyszeres multiplicitású élekkel**, melyeknek a jelenléte nagyban leredukálja a mintaillesztés bonyolultságát. A jelenlegi rendszer minden élt egy listán keresztül ér el, még akkor is, ha az elemek száma biztosan nulla vagy egy. Ezt lehetne módosítani úgy, hogy az egyszeres éleket közvetlenül tudjuk elérni. Ez természetesen „elrontja" a bejárás egyszerűségét és többletterhet okoz az algoritmusok készítésére, viszont a teljesítményben biztosan megjelenne a hatása. Az általam ismert modellekben rendszeresen (talán többségében) vannak azok az élek, melynek legalább az egyik oldalán a multiplicitás nulla vagy egy. Ezeknek az éleknek történhetne gyorsabban a bejárása, valamint garantáltan kevesebb helyet foglalna a gráf a memóriában.

7. Az eredmények áttekintése

Dolgozatomban összefoglaltam néhány fontos kutatási eredményt a modellvezérelt programozásban használt gráftranszformációk témaköréből. Ehhez kapcsolódóan egy elméleti megfontolással kezdtem neki saját munkám bemutatásának: a mintakeresés és az élek multiplicitása közti kapcsolatra adtam egy összefüggést.

Munkám fő része egy, a keretrendszerhez illeszkedő párhuzamosítási eljárás kidolgozásából állt. Az algoritmust elkészítettem, különböző részleges, majd teljes transzformációs lépésekben végeztem méréseket, és meghatároztam az elkészített alkalmazás korlátait: ez a párhuzamosítási megoldás akkor használható, ha nagy mintákat keresünk nagy gráfban, és nincs túl sok módosítási feladat. Tehát a *kisebb transzformációkban*, ahol a teljesítmény általában nem kulcskérdés, az itt ismertetett párhuzamosítás *kerülendő*. Viszont a *nagyobb rendszereknél*, ahol a transzformáció exponenciális volta kezdi éreztetni magát, a párhuzamosításra szükség van, és megéri használni. Az állításaimat az elkészített, valós adatokon álló modellen végzett mérésekkel támasztottam alá.

Munkám során született néhány észrevétel és fejlesztési javaslat a keretrendszer jelenlegi működésének fejlesztésére. Kiemeltem több ígéretesnek tűnő kutatási irányvonalat, amely felé lehet folytatni a kutatást.

Függelék

A. A mintaillesztés bonyolultsága az élek multiplicitásától függően

Ebben a dolgozatban szigorúan típusos gráfokkal foglalkozom, melyekben a mintaillesztés metamodell-alapú származtatás alapján történik – azaz mind a kiindulási elem kiválasztása, mind a gráfon belüli keresés szigorúan típusosan és a metaszinten definiált élváltozók szerint történik. Ebben a fejezetben azt a kérdést vizsgálom, hogy az élek multiplicitása hogyan befolyásolja a keresés bonyolultságát.

Tudjuk, hogy az alapvető probléma, az izomorf részgráfkeresés NP-nehéz, úgyhogy ennél jobb általános megoldásra nem számíthatunk. Viszont vajon milyen speciális feltételek mellett kapunk viszonylag kellemes bonyolultságú algoritmusokat?

Az alap algoritmus, amelyekből lényegében összeillesztjük a transzformációs szabályokat, a következő:[19]

```
// First, outter for cycle
connetingRelationList = model.chosenTypeList; // List of items of given type
for (n = 0 to connetingRelationList.GetSize()) {
    possibleNext = connetingRelationList[n];
    if (ConstraintsFail(possibleNext) OR CollisionWithPrev(possibleNext))
        continue; // Constraints fail or repetition of a chosen node
    next = possibleNext; // Save
    connetingRelationList = next.chosenEdgeList; // Selecting next edgeList
    // Next embedded for cycle
}
```

A szabály tehát a fentihez hasonló egymásba ágyazott ciklusokból áll. A külső ciklus a modell összes olyan típusú elemét vizsgálja végig, melyeknek a típusa megegyezik az első „match-elt” él (vagy *node*, gráfpont) típusával, az utolsó él pedig az ellenőrzések és a mentés után az eredmény kezelését végzi. Ez utóbbi jelen dolgozatban vagy egy callback hívást, vagy a módosítás azonnali elvégzését jelenti.

Nézzünk egy egyszerű példát, ahol két élt keresünk:

```
connetingRelationList = Model.RelationType1List;
for (n = 0 to connetingRelationList.GetSize()) {
    possibleStart = relationList[n];
    if (ConstraintsFailStart(possibleStart))
        continue;
    startEdge = possibleStart;
    startLeft = startEdge.Left;
    startRight = startEdge.Right;
    nextList = startEdge.ChoosenSide.NextRelationList;
    for (n2 = 0 to nextList.GetSize()) {
        possibleNext = relationList[n2];
        if (ConstraintsFailNext(possibleNext) OR possibleNext==startEdge)
            continue;
        nextEdge = possibleNext;
```

[1919] Ehhez jön hozzá a „kegyes megszakítás” vizsgálata, a szálbiztosság ellenőrző részei, valamint egyéb részletek, de ezek az algoritmus lényegét már nem változtatják.

```
            nextNode = possibleNext.OtherSide;
            Callback(this); // Match found
        }
    }
```

A fenti példában egyértelműen látszódik, hogy milyen lépésektől függ az algoritmus bonyolultsága:

1. Milyen nagy az a lista, melyből az első élt választjuk?
2. Az egyes vizsgált relációk milyen multiplicitással rendelkeznek?

A **kezdeti lista** nagyságának felső korlátja az élek száma, és általában jól becsülhető az élek számának konstans hányadával ($K \cdot N_e$).[20] A gráf pontjaiból az élek számának kifejezése nem triviális, hiszen elvileg megengedhetünk többszörös éleket is, úgyhogy szigorú felső becslést nem tudunk adni. Viszont a gyakorlatban az $K \cdot {N_v}^2$ megfelelő becslésnek minősül, de gyakran a $K \cdot N_v$ is jó közelítést ad. Sokszor ugyanis a gráf növelése során az új elemek (pontok) számával arányosan jönnek létre új élek.

A **multiplicitás** kapcsán feltételezem, hogy egy-egy elem ellenőrzéséhez egységnyi idő szükséges. Az ellenőrzés lépései a következők:

1. Adott éllistából elem kiválasztása,
2. Kényszerek ellenőrzése,
3. Él és kapcsolódó *node* mentése

Megjegyzés. Az egységnyi idő a gyakorlatban sokszor nem teljesen igaz a különböző kényszerek miatt, de nagyságrendi eltérés többnyire nem fordul elő.

Három elemből és két relációból álló találat keresése esetén a lépések a következők:

1. Az első reláció listáját a modelltől kérjük le. Multiplicitás $\approx N_e(typeof(e_1))$, azaz a mintában az e_1 típusú élek száma
2. Minden élre a listában (Multiplicitás $\approx d_1$, ahol d_1 a lista multiplicitása)
    a. Él ellenőrzése és mentése, ha mégsem jó, GOTO 2.
    b. A következő éllista minden elemére (Multiplicitás $\approx d_2$)
        i. Él ellenőrzése, ha jó, akkor mentés és **Találat**
        ii. Ha nem jó, GOTO 2/b
    c. Ha elfogytak az élek, GOTO 2
3. Ha elfogytak az élek, akkor **Nincs találat**

A keresés költsége így a következő módon számítható (ahol $d_i \geq 1$ egész):

$$C = N_e\big(typeof(e_1)\big) \cdot (d_1 + d_1 \cdot (d_2))$$

Tetszőleges számú elemre ez a következő módon néz ki:

[20] Az algoritmus többnyire él alapú kereséssel indít, bár előfordul, hogy egy adott típusú pont keresése az első lépés. Utóbbi lépésszámának felső becslése nem lehet nagyobb az élek alapján történő lépésszámának felső becslésével.

$$C = N_e\big(typeof(e_1)\big) \cdot \Big(d_1 + d_1 \cdot \big(d_2 + d_2 \cdot (\cdots)\big)\Big) =$$

$$= N_e\big(typeof(e_1)\big) \cdot (d_1 + d_1 d_2 + d_1 d_2 d_3 + \cdots)$$

Az első elem választásának a stratégiájával nem szeretnék foglalkozni, ezt megteszi a (Mezei, Transformation-based Support for Visual Languages, 2007) dolgozat. Viszont nagyon könnyen belátható, hogy a keresés során mindenképp olyan utat érdemes választani, ahol kis multiplicitású élekkel találkozunk. Ezen belül az ellenőrzés sorrendjében is megéri előre venni a kis multiplicitású elemeket.

Állítás. Egységnyi lépéseket feltételezve a fenti algoritmus szerinti mintát fix kezdőpontból úgy optimális végrehajtani, hogy az élek ellenőrzését a bejárási utat adó éllisták típusának multiplicitása szerint növekvő sorrendben végezzük.

Bizonyítás. Adott az alábbi bejárási költség.

$$C = d_1 + d_1 d_2 + d_1 d_2 d_3 + \ d_1 d_2 d_3 d_4 + \cdots$$

Az egyszerűség kedvéért felteszem, hogy nincs két azonos multiplicitás. Tegyük fel, hogy az állítás teljesül, azaz $d_i < \ d_j$ ha $i \leq j$. Ekkor két elem megcserélése mindenképp növelné C költségét. Az összeadás $n < i$ és az $n > j$ indexű elemei nem változnának, viszont az $i \leq n < j$ indexű elemekre teljesülne, hogy

$$\frac{d_j}{d_i} \cdot (P_n) > (P_n),$$

ahol P_n az n. tagja a költségnek. A bizonyítás a csere után is ugyanígy igaz, tehát az elemek további cseréjével sem tudjuk a költséget csökkenteni. ■

Azt láttuk tehát, hogy a futási idő a kezdetben választott elem modellbeli gyakoriságától, valamint a bejárás során érintett élek multiplicitásától függ. Láttuk azt is, hogy a bejárás során célszerű az alacsony multiplicitású élekkel kezdeni a bejárást.

A fentiek alapján azt mondhatjuk, hogy a bejárás lépésszáma az élek multiplicitásainak szorzatával arányos, azaz

$$C \leq K \cdot \prod_{i=0}^{N_r} d_i,$$

ahol d_i az i. él multiplicitása, N_r a *match* relációinak száma.

Ez azt jelenti, hogy az algoritmus mindenképp exponenciális futásidejű, viszont annyit nyertünk, hogy nem függ exponenciálisan a bemenet méretétől, csak a bemenet multiplicitásaitól. Ez azt jelenti, hogy n—0..1 vagy n—1..1 multiplicitású relációkkal való bővítés minimális pluszköltséggel jár.

B. A gráftranszformációk párhuzamosítása és a törlés problémája

Az alábbiakban egy futási alternatívára szeretnék rávilágítani, amely miatt a párhuzamos keresést és módosítást mindenképp kerülendőnek tartom. A jelenlegi algoritmus és adatszerkezet megtartását feltételezem, ennek megfelelően mutatok be egy kis valószínűségű, de igen veszélyes hibalehetőséget. Tegyük fel, hogy teljesülnek az alábbi feltételek.

I. Az egyik szálon adott egy találat, ahol a módosítás során a gráf egy pontját töröljük
   a. A törölt elem valamilyen relációban áll találaton kívüli elemekkel. (Ez a legtöbb törlés során csak nehezen lenne megkerülhető.)

II. A másik szál eközben keresést végez
   a. A keresés a törölt elem egyik találaton kívüli szomszédjától halad és érinti azt az állistát, melyben a törlésre szánt elem egyik éle található.

Ha ebben az esetben töröljük a törlendő elemhez kapcsolódó élt, akkor a nem determinisztikus szálkezelés miatt a következő helyzet állhat elő:

1. szál	2. szál
for ciklus méretének lekérése (következő elem a lista utolsó eleme lenne)	-
<megszakítás>	<folytatás>
-	elem törlése
-	kapcsolódó szálak törlése, mely során a másik szálban vizsgált állistából is törlünk egy elemet
<folytatás>	<megszakítás>
kapott indexű elemre ugrás → Hibás index! (A méret közben eggyel csökkent)	-

A futás során menedzselt kódból *OutOfBoundary* kivételt kapunk, nem menedzselt kódból jó esetben memóriakezelési vagy egyéb kivételt, rossz esetben pedig a programunk fut tovább, státusza: *undefined*… A hiba előfordulása nagyon ritka, de bekövetkezte súlyos problémákkal jár.

Ez ellen a hiba ellen legegyszerűbben úgy lehet védekezni, hogy a módosításkor szálbiztos blokkban jelöljük a kapcsolódó állistákat (vagy az élek túlfelén az elemeket), kereséskor pedig ezeket ugyancsak szálbiztosan ellenőrizzük, és figyelünk arra, hogy a módosítás ne foglalhassa le olyan élhalmaz elemét, amellyel épp foglalkozik a kereső szál. Ez a megoldás irreálisan sok pluszmunkát jelentene.

Nem tekintem át a részletes algoritmizálási lehetőségeket, csak annyit szeretnék leszögezni, hogy jelen keretek között erre a problémára nem találtam hatékony megoldást. (Valószínűleg azért, mert hatékony megoldás erre a problémára nem létezik.) Ezért dolgozatomból kihagytam minden olyan párhuzamosítási lehetőséget, amely párhuzamos, más kereséssel vagy módosítással potenciálisan átfedésben lévő módosítást tenne lehetővé.

Irodalomjegyzék

32 nanometer - Wikipedia. (dátum nélk.). *32 nanometer*. Letöltés dátuma: 2011. 10 26, forrás: Wikipedia: http://en.wikipedia.org/wiki/32_nanometer

BME AAIT. (2011). *Visual Modelling and Transformation System - VMTS*. Letöltés dátuma: 2011. 10 26, forrás: BME AAIT: http://avalon.aut.bme.hu/~tihamer/research/vmts/

Buttyán, L., & Vajda, I. (2004). *Kriptográfia és alkalmazásai.* Budapestq: TypoTeX.

Cheng, X., Dale, C., & Liu, J. (2008). Statistics and Social Network of YouTube Videos. *Quality of Service, 2008. IWQoS 2008. 16th International Workshop on 2-4 June 2008* , 229 - 238.

Cheng, X., Dale, C., & Liu, J. (2008). *YouTube Dataset*. Letöltés dátuma: 2011. 10 26, forrás: School of Computing Science, Simon Fraser University: http://netsg.cs.sfu.ca/youtubedata/

Coretex-A5 Processor - ARM. (dátum nélk.). *Coretex-A5 Processor - ARM*. Letöltés dátuma: 2011. 10 26, forrás: ARM - The Architecture For The Digital World: http://www.arm.com/products/processors/cortex-a/cortex-a5.php

Ehrig, H., Ehrig, K., Prange, U., & Taentzer, G. (2006). *Fundamentals of algebraic graph transformation.* Springer Verlag.

Hansson, D. H. (dátum nélk.). *Ruby on Rails*. Letöltés dátuma: 2011. 10 26, forrás: Ruby on Rails: http://rubyonrails.org/

Haslhofer, B. (2008). *MOF.* Letöltés dátuma: 2011. 10 26, forrás: TWR - Technology Watch Report: http://metadaten-twr.org/2008/09/22/mof/

Jazelle - ARM. (dátum nélk.). *Jazelle - ARM*. Letöltés dátuma: 2011. 10 26, forrás: ARM - The The Architecture For The Digital World: http://www.arm.com/products/processors/technologies/jazelle.php

Jordán, T., Recski, A., & Szeszlér, D. (2004). *Rendszeroptimalizálás.* Budapest: TypoTeX.

Kernighan, B., & Ritchie, D. M. (1988). *The C Programming Language.* Englewood Cliffs, NJ: Prentice Hall.

Mezei, G. (2007). *Transformation-based Support for Visual Languages.* Budapest: Budapest University of Technology and Economics, Department of Automation and Applied Informatics.

Mezei, G., Levendovszky, T., Mészáros, T., & Madari, I. (2009). Towards Truly Parallel Model Transformations: A Distributed Pattern Matching Approach. *International IEEE Conference devoted to the 150- anniversary of Alexander S.* , 403-410.

Microsoft. (2011). *MSDN Library*. Letöltés dátuma: 2011. 10 26, forrás: MSDN: http://msdn.microsoft.com/en-us/library/

OMG. (2006. 01 01). *MOF 2.0*. Letöltés dátuma: 2011. 10 26, forrás: Object Management Group: http://www.omg.org/spec/MOF/2.0/

Schmidt, D., Stal, M., Rohnert, H., & Buschmann, F. (2000). *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects.* Baffins Lane, Chichester: John Wiley & Sons, Ltd.

Simula - Wikipedia. (dátum nélk.). *Simula - Wikipedia*. Letöltés dátuma: 2011. 10 26, forrás: Wikipedia: http://en.wikipedia.org/wiki/Simula

Smalltalk - Wikipedia. (dátum nélk.). *Smalltalk - Wikipedia*. Letöltés dátuma: 2011. 10 26, forrás: Wikipedia: http://en.wikipedia.org/wiki/Smalltalk

Stroustrup, B. (2000). *The C++ Programming Language (Special Edition ed.).* Addison-Wesley.

ucblockhead. (2002. 06 25). *Language Comparison, C#, C++ and Java*. Letöltés dátuma: 2011. 10 26, forrás: Kuro5hin: http://www.kuro5hin.org/story/2002/6/25/122237/078