Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Gráfmintaillesztő rendszerek tesztelése automatikus gráfgenerátorokkal

TDK dolgozat

Készítette:

Bekő Mária

Konzulens:

Semeráth Oszkár

2018

Tartalomjegyzék

Kivonat

Napjainkban az adatokat többféle formátumban is tárolják. Ezek közé tartoznak a gráfadatbázisok, ahol csomópontok reprezentálják az entitásokat és az élek az entitások közötti kapcsolatokat. Az adatstruktúrához illeszkedve többféle gráflekérdező nyelv jött létre, amelyek képesek komplex struktúrák felírására.

A gráfmintaillesztő rendszerek tesztelése azonban komoly kihívást jelent, főképp automatizált megoldásokban nem bővelkedünk. A legnagyobb kihívást ebben az esetben a változatos modellek és lekérdezések automatikus és szisztematikus előállítása jelenti, melyek tesztbemenetként szolgálnak. Továbbá, gráfadatbázisok teljesítménymérését is nagyban segítené az automatikusan előállított modellkészlet.

Dolgozatom célja hogy ezekre a problémákra megoldást találjak. Kutatásom során megmutatom, hogy egy automatikusan előállított diverz modell halmazzal, amelynek modelljei lekérdezésként értelmezhetőek egy gráfmintaillesztő rendszerben (pl.: VIATRA vagy Neo4j), hogyan lehetséges az adott gráfmintaillesztő rendszer tesztelése.

Munkám során fejlett logikai következtetők alkalmazásával állítok elő modelleket, melyek diverzitását szomszédsági formákkal (neighborhood shape-ek) biztosítom. A logikai következtetők eredményeit lekérdezésekként, és adatbázisok tartalmaként egyaránt értelmezhetjük, amelyek eredményei különböző megvalósításokkal összehasonlíthatóvá válnak. A megoldásomat egy esettanulmány keretében prezentálom.

Ezzel a módszerrel lehetővé válik, nagyobb megbízhatóságú gráfmintaillesztő rendszerek fejlesztése olcsóbban. Illetve egy ekkora modell halmaz különböző gráfmintaillesztő rendszerekben lekérdezésekre fordítva és a válaszidőket lemérve teljesítménymérésekre is használható.

Abstract

Nowadays the data is stored in multiple formats. One of this is the graph database, where entities are represented as graph nodes and the relations between entities as edges. Utilizing rich data structure, a variety of graph querying languages have been created in order to query complex structures.

Testing of graph query engines is a challenging task, especially in an automated way. The greatest challenge in this case is the automatic and systematic creation of a diverse set of models and queries, which serve as test inputs. In addition, the development of performance benchmarks for graph databases would be greatly aided.

The purpose of my thesis is to find solutions to these problems. In my thesis, I will show an approach to automatically generate a diverse set of models, that can be interpreted as queries of the graph query engine under test (e.g. VIATRA model query language or Neo4j graph database queries). Additionally, I propose a testing process.

In the course of my work, I produce models with the help of state-of-the-art logic solvers (SAT solvers and Graph Solvers), and use neighborhood shapes to ensure their diversity and create effective equivalence partitioning. The results of the logic solvers are interpreted as queries and as databases content, and the result of query evaluation can be compared to other implementations. I illustrate my solution in a case study.

This method makes it possible to develop more reliable graph query engines at a lower cost. And such a set of models translated to multiple graph querying languages can be used in those graph query engines for performance measurements by measuring response times.

1. fejezet

Bevezetés

Kontextus. Napjainkban ....

Problémafelvetés. Azonban ...

Célkitűzés. Dolgozatom célja ...

Kontribúció. Dolgozatomban bemutatok ...

Hozzáadott érték. Ezáltal ...

Dolgozat felépítése.A második fejezetben bemutatom a dolgozat megértéséhez szükséges háttérismereteket. blabla ...

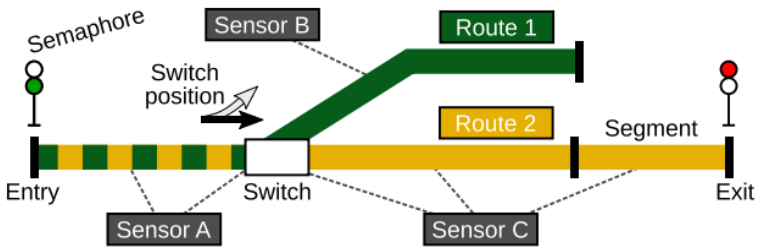
2. fejezet

Előismeretek

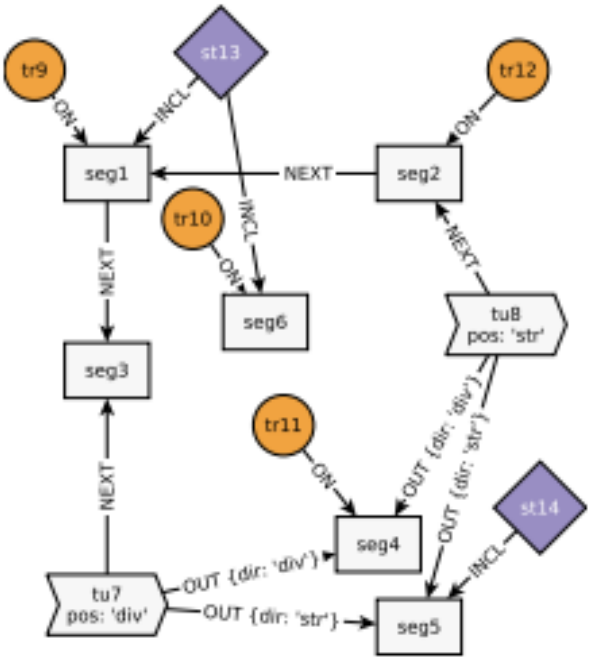
2.1. Esettanulmány

A dolgozatban elért eredményeket a Train Benchmark [14] esettanulmány segítségével fogom bemutatni. Ez a benchmark azért jött létre, hogy össze tudjuk hasonlítani különböző gráflekérdező rendszerek teljesítményét, beleértve gráfadatbázisokat, mint Neo4j [1], SparkSee [6], nagy teljesítményű modellezőeszközöket, mint VIATRA [10] és szokványos relációs adatbázisokat, mint az Oracle [9], főleg időigény és memória felhasználás szempontjából. A Train Benchmark egy vasúti modellezőeszköz esettanulmányán mutat be olyan, különböző terhelésprofilokat, amelyek hasonlítanak egy valódi modellezési feladathoz. Munkám során a benchmarkban specifikált formátumhoz illeszkedve generáltam lekérdezéseimet, ezért bemutatom, hogy milyen elemekből áll.

A 2.1-es ábrán látható egy a Train Benchmark modelljére alapuló részlet. Ebben a kontextusban egy vasúti útvonal(*Route*, zölddel és sárgával jelölve) nem más mint szegmensek(*Segment*, két fekete vonal közötti rész) és váltók (*Switch*, fehér téglalappal jelölve) sorozata, illetve a belépést és a kilépést egy-egy szemafor(*Semaphore*, a két színű lámpácskák jelölik) jelzi. Ahhoz, hogy biztonságos legyen a közlekedés szükség van szenzorokra, amelyek monitorozzák a különböző szegmensek és váltók kihasználtságát. Egy útvonal definiálásához, a felsorolt elemeken kívül a váltók adott útvonalhoz tartozó pozícióját is el kell tárolnunk. Egy útvonal akkor aktív, ha a rendszerben a specifikációjának megfelelően állnak a váltók.



2.1. ábra. Vasúti útvonal részlet (forrás: [14])



2.2. ábra. Tulajdonsággráf példa(forrás: [7])

2.2. Gráflekérdező rendszerek

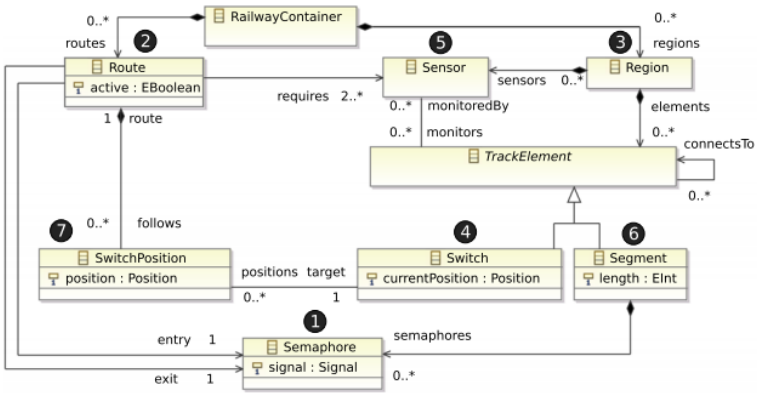
2.2.1. Tulajdonság gráfok

A gráfok intuitív formalizációt nyújtanak modellezési szempontból arra, hogy úgy írhassuk le a világot ahogy az ember gondolkozik róla. Tehát mint dolgok (csomópontok) és köztük lévő kapcsolatok (élek)[7]. A tulajdonsággráf (property graph) adatmodell kiterjeszti a gráfokat úgy, hogy címkéket/típusokat, illetve tulajdonságokat ad a csúcsoknak és az éleknek. A gráf adatbázisok alkalmasak tulajdonsággráfok tárolására, és az abban lévő adatok lekérdezésére komplex gráf minták használatával. Ilyen rendszerek például a Neo4j [1], OrientDB [4] és a SparkSee [6].

Ahhoz hogy jobban megérthessük mi is ez az adatmodell a 2.2 -es ábrán látható egy példa. Az ábrán minden ami valamilyen forma egy-egy csomópont, és minden csomópont reprezentál elemeket a Train Benchmark adatmodelljéből. A fehér téglalapok a szegmenseket, a sárga körök a vonatokat, a lila rombuszok az útvonalakat, és a fehér zászlók a váltókat. Minden elem fel van címkézve ezen kívül egy névvel (seg1, seg2, tr1 stb...), a formájuk és a nevük is tulajdonságok amelyek segítségével megkülönböztethetővé válnak,és amelyek egy egyszerű gráfban már nem lehetnének jelen. Az élek pedig kapcsolatokat mutatnak az egyes elemek között. Például az ábrán vannak összekötött vonatok és szegmensek, az összekötő élen ”ON” felirattal. A vonattól mutatnak a szegmens irányába. Egy ilyen irányított él azt reprezentálja, hogy az adott vonat rajta van az adott szegmensen(angolul : train is ON segment). Ilyen formán az élekhez is rendelhetőek tulajdonságok.

2.2.2. Neo4j

A Neo4J egy népszerű NoSQL tulajdonsággráf adatbázis és a Cypher lekérdező nyelvet kínálja lekérdezések írására. A Cypher egy magas szintű deklaratív lekérdező nyelv és mivel le van választava a lekérdező rendszerről, ezért az képes a Cypher nyelven írt lekérde-



2.3. ábra. A Train Benchmark metamodellje.(forrás: [14])

zések optimalizálására. A Cypher szintaxisa olyan gráf minták megírását teszi lehetővé, amelyeknek megértése nagyon egyszerű.

```
MATCH (tr:Train)-[:ON]->(seg:Segment)
RETURN tr, seg
```

A fenti példán egy olyan lekérdezést láthatunk, ami az összes olyan vonat, szegmens párral tér vissza, ahol az adott vonat rajta van az adott szegmensen. Dolgozatomban Cypher nyelvű lekérdezések generálásával foglalkozom.

Egy Cypher nyelvű lekérdezésben a `MATCH` kikötést arra használjuk hogy megkeresse a mintát amit leírunk benne. A `RETURN` kikötés meghatározza hogy mi kerüljön bele a visszatérési értékbe. () zárójelek között változókat definiálunk és meghatározhatjuk a címéjüket is például (tr:Train) a tr a változó neve, ami Train típusú. zárójelek között ezután további tulajdonságokat köthetünk ki. –> jelöli a kapcsolatot két változó között, ahol a két kötőjel között [] zárójelekben megadható a kapcsolatra vonatkozó címke.

A Neo4j képes arra hogy egy ilyen lekérdezést beolvasson, és kiértékeljen majd visszatérjen a lekérdezésre adott válaszokkal. Azonban, mint minden szoftver a lekérdező rendszerek is tartalmaznak hibákat,amelyek következtében hibás kimeneteket adnak a lekérdezések eredményeként. Ezért a lekérdező rendszerek tesztelése kiemelten fontos.

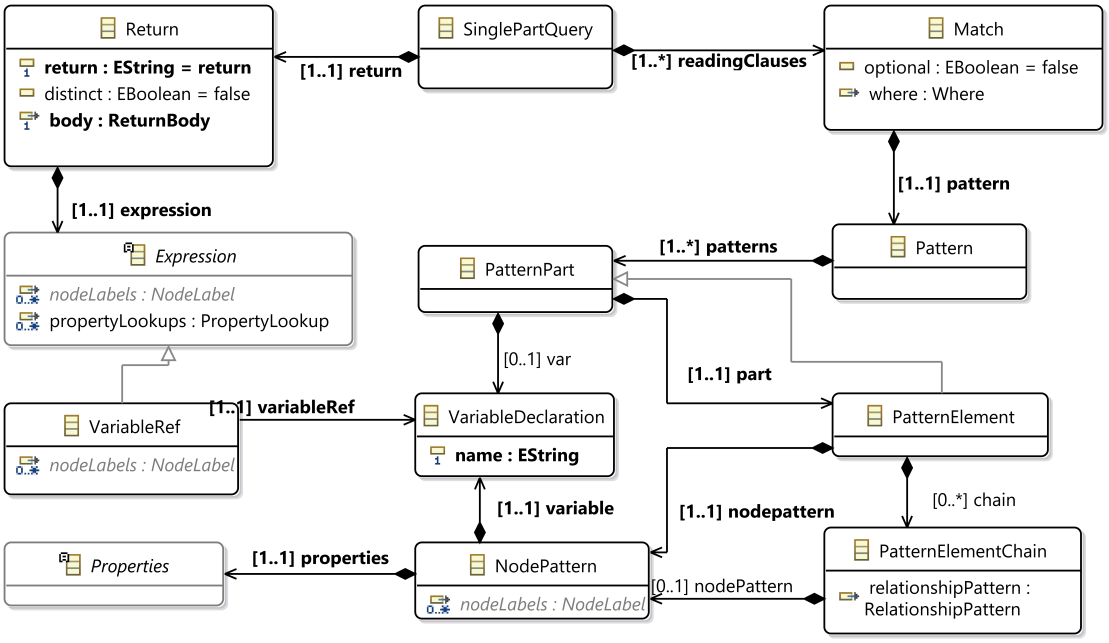
2.3. Modellezés és metamodellezés

A metamodellezés egy tecknika arra, hogy definiáljunk új modellező nyelveket. A Train Benchmark metamodellje a 2.3 -es ábrán látható.

A céldomén legfontosabb fogalmait és kapcsolatait foglalja össze a metamodell, így specifikálva a modellek alap struktúráját [11]. Dolgozatomban az Eclipse Modeling Framework (EMF) [5] -öt használtam metamodellezésre. EMF esetén a fogalmakat osztályokkal (EClass) a kapcsolatokat referenciákkal és attribútumokkal (EReference és EAttribute) írjuk le.

2.3.1. Cypher query-k metamodellje

A 2.4 -ös ábrán látható a korábban említett Cypher nyelv egyszerűsített metamodellje. A `SinglePartQuery` elem reprezentálja a modell gyökerét. Egy ilyen elem két részből áll : Egy `Match` és egy `Return` elemből. A `Match` elem minták összességéből áll (`Pattern`), amelyek részekre (`PatternPart`) bonthatóak. Egy ilyen részben pedig vagy tartalmaz változó dek-



2.4. ábra. Cypher metamodell

larációt, vagy egy belső részt, ami tartalmaz változó deklarációt. (`VariableDeclaration`). Ezáltal az összes változót a `Match` elemen belül deklaráljuk. A `Return` elem pedig egy kifejezést (`Expression`) tartalmaz, amelyben mindenképp szerepel egy változó referencia (`VariableRef`) is, így összekötve egymással a `Match` és a `Return` elemet.

2.3.2. Xtext

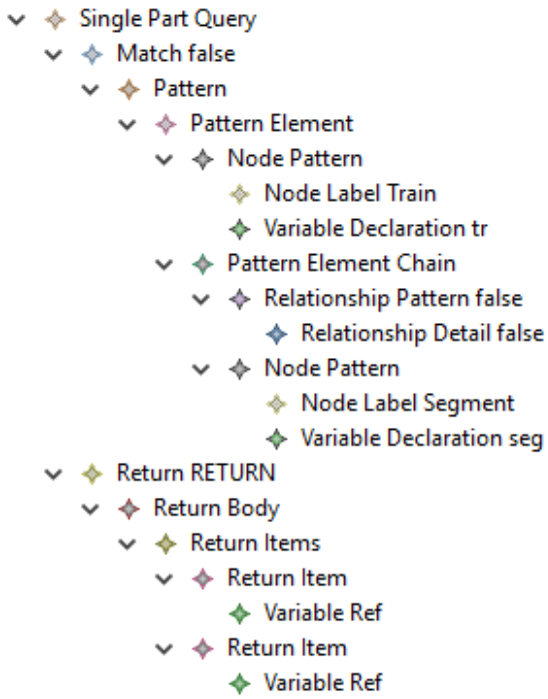
Az Xtext keretrendszer programozási nyelvek, domain-specifikus nyelvek és szöveges editorok fejlesztésére készül. Az Xtext egy erős nyelvtani szabályokkal rendelkező nyelvet használ az egyedi nyelvtanok definiálására. Ezáltal egyszerre biztosít parseolót, linkelőt, helyesírásellenőrzőt szövegkiemelést, hibaüzeneteket és írás során könnyen kiegészíthető fordítókkal, kódgenerátorokkal és egyéb modellezőeszközökkel. És a nyelvtervező mérnök határozhatja meg a nyelvének célformátumát is. Ahhoz, hogy a Cypher nyelven megírt lekérdezéseket értelmezni lehessen a 2.4-ös metamodellen megismert elemek szintjén egy Xtext [3] keretrendszerben íródott nyelvtanra van szükség, amelyből a metamodell automatikusan elkészült . A dolgozatomban a slizaa[13] nevű Xtext alapú Cypher nyelvtant használtam. ASG

```
SinglePartQuery:
(readingClauses+=ReadingClause)* return=Return ;

Return:
(return='RETURN' distinct?='DISTINCT'? body=ReturnBody);

ReadingClause:
LoadCSV | Start | Match | Unwind | InQueryCall;
```

A fenti ábrán a SinglePartQuery elem Xtext nyelvtana látható. Azt mondja ki, hogy amikor egy ilyen elem készül akkor összerak egy vagy több ReadingClause-t (Match elem absztrakt ősosztálya, 3. nyelvtani részlet), és egy returnt. Alatta pedig a Return elem Xtext nyelvtana következik, ami azt mondja ki hogy a return elemet úgy kell sorosítani, hogy



2.5. ábra. A példalekérdezés kibontása Xtext-tel

”`RETURN DISTINCT`(ezt csak akkor kell odaírni ha ezt a tulajdonságot igazra állítottuk) `kifejezes`”. Tehát itt határozza meg hogy a Return elem úgy néz ki mint az alábbi lékérdezésen.

```
MATCH (tr:Train)-[:ON]->(seg:Segment)
RETURN tr, seg
```

Ezt a lekérdezést Xtext segítségével ősösztlyaira lehet bontani, és megnézni, mi micsoda benne. Erről a 2.5 ábrán láthatunk egy példát.

2.3.3. VIATRA jólformáltsági kényszerek

Az Eclipse VIATRA keretrendszer [10] egy modell lekérdező, validáló és transzformációs eszköz. Specifikusan olyan eseményvezérelt és reaktív transzformációkra fókuszál, amelyek a modell változása közben történnek. Hibás modellrészletek hibamintákkal történő megfogalmazásával olyan jólformáltsági kényszereket is megadhatunk általa, amelyek kifejezésére a metamodell önmagában nem lenne alkalmas. A lekérdezés generálás során erre használom.

```
pattern hasMatch (q : SinglePartQuery, m: Match){
  SinglePartQuery.readingClauses(q,m);
}

@Constraint(severity = "error", key ={q}, message ="error")
pattern hasNoMatch(q : SinglePartQuery) {
    neg find hasMatch(q,_);
}
```

A fenti példán egy VIATRA kényszert láthatunk, ami egy segédmintával van meghatározva. A felső segédminta arról szól, hogy egy `SinglePartQuery` -nek van `Match`-e. Az alsó kényszer pedig ellenpéldát keres a fenti lekérdezésre, amikor a `Match` nincs kitöltve. A @Constraint sor azt jelenti, hogy ha az alatta levő mintára példát találunk akkor azt a modellt ott helyben dobhatjuk félre.

2.4. Gráfgenerálás

Munkám alapját mégis leginkább gráfok generálása képezi. A gráfgenerálás célja, hogy egy adott feladatra szintetizáljon gráfokat A Viatra Solver [15] egy korszerű nyílt forráskódú szoftver keretrendszer amely képes diverz szakterület-specifikus gráf modellek automatikus szintézisére, melyek teszt készletként használhatóak gráf alapú modellező eszközök szisztematikus tesztelése során. Bemenetként a megoldó

- a tesztelni kívánt modellező eszköz specifikációját használja fel metamodell formátumban az Eclipse Modeling Framework-öt használva
- jólformáltsági kényszerek egy halmazát a VIATRA keretrendszer használatával
- opcionálisan egy példánymodell részletet

Kimenetként pedig diverz gráfok egy halmazát generálja. Minden kimeneti gráf megfelel a metamodell specifikációinak és kielégíti az összes jólformáltsági kényszert. Struktúrájukban pedig különböznek egymástől biztosítva ezzel tesztkészlet diverzitását. Én ezt a keretrendszert használom.

3. fejezet

Áttekintés

3.1. Funkcionális áttekintés

Munkám célja hogy elkészítsek egy olyan megközelítést, amely képes gráflekérdezések automatikus generálására, gráfmintaillesztő rendszerek tesztelése. Az elképzelt keretrendszer koncepcionális elrendezését a 3.1 ábrán mutatom be. Az ötlet lényege az, hogy a tesztelni kívánt rendszer **nyelvi specifikációjának** és egy **esettanulmány szignatúrájának** (ezalatt egy olyan tulajdonsággráf adatmodell alapú adatbázis szignatúrájára gondolok amely az adott lekérdező rendszert használja) bemenetként való felhasználásával, a kimeneten szöveges és a gráfmintaillesztő rendszer nyelvén íródott lekérdezéseket kapjak.

Amint rendelkeznék egy tesztkészletnyi ilyen lekérdezéssel azokat tudnám futtatni azon az adatbázison amelynek szignatúráját esettanulmányként választottam. Ha az ezzel a módszerrel generált lekérdezésekre adott válaszok válaszidejeit összehasonlítjuk több rendszeren tudnánk **teljesítményben tesztelni** azokat. Illetve ha a generált lekérdezések helyes eredményével is rendelkeznénk, (például több létező implementációt összehasonlítanánk, és egyiket a másik referenciájaként használnánk, akkor a referencia megfelelő tesztorákulumként szolgálhatna, így ellenőrizni tudnánk azt is hogy a teszt alatt álló lekérdező rendszer hogyan funkcionál, helyes válaszokat ad-e? Ha elég bonyolultak a lekérdezések, akkor az is lehetséges, hogy lesznek olyanok amelyek az egyes lekérdező motorokon nem míg másikakon működnek, így azt is **tesztelni** tudnánk hogy mekkora az egyes lekérdező motorok funkcionális lefedettsége.

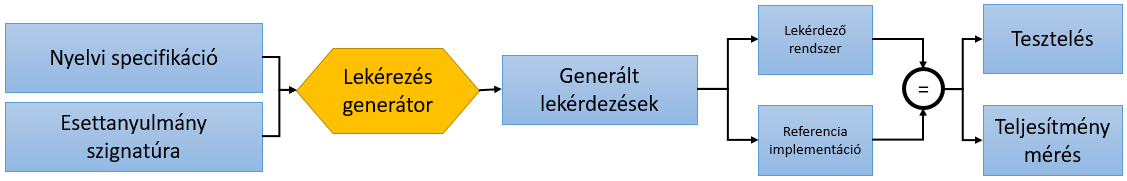
bekezdés ami Oszkárnak kell

Felmerül a kérdés, hogy mégis miért lesz ez jobb, mintha írnánk a lekérdezéseket magunk. Azért, mert a generálás segítségével ki tudunk törni az emberi sematikus gondolkozásból, és olyan lekérdezéseket tudunk készíteni amelyek számításokkal bizonyítottan különböző ekvivalencia osztályba tartoznak. Az ekvivalencia partícionálás, egy bevett technika tesztelésnél [2], mérésekkel bizonyították, hogy a generált modellek szignifikánsan magasabb tesztfedettséget mutattak, mint a manuálisan elkészítettek [12]. Illetve nem korlátoz minket az sem, hogy a teszteléshez írt lekérdezésekből túl kevés van, mert a generátor segítségével megadott számú minta, akár óriási tesztkészlet előállítható automatikusan.

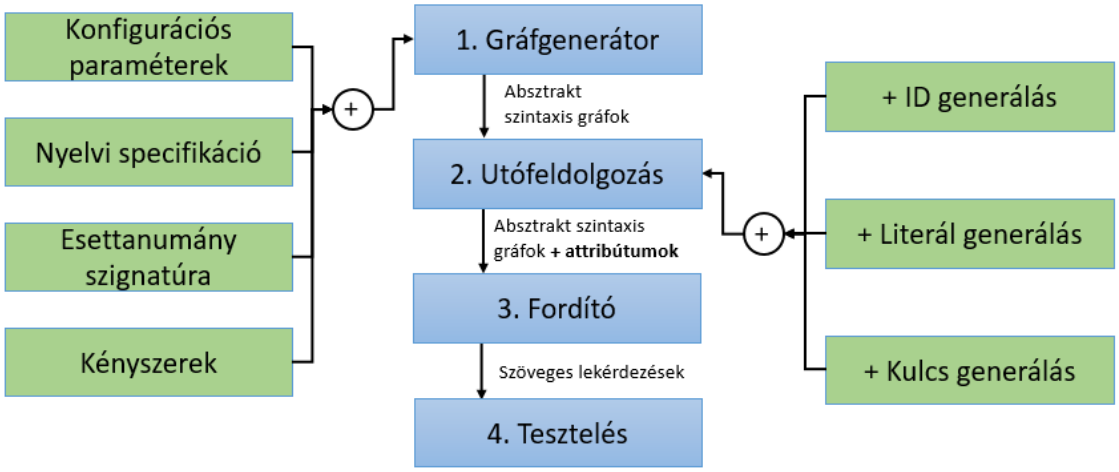
A megközelítésemet egy Neo4j [1] property gráf adatbázison mutatom be, amely a Train benchmark [14] által használt szignatúrával van felszerelve. A lekérdezéseket a Neo4j által kifejlesztet Cypher [8] nyelven generálom, a slizaa [13] által készített openCypher nyelvi specifikáció felhasználásával.

3.2. Lekérdezés generálási folyamat felépítése

A folyamat felépítését a 3.2 -es ábra segítségével ismertetem.



3.1. ábra. Az elképzelés funkcionális áttekintése



3.2. ábra. A lekérdezés generálási folyamat áttekintése

1. **Gráfgenerátor** A gráf generálást a Viatra Solver keretrendszer segítségével végzem. Ehhez sok különböző bemenetet kell megadnom.

   (a) **Nyelvi specifikáció**: A Cypher nyelv specifikációját tartalmazó metamodell a nyelv egészére kiterjed. Így lekérdezéseken kívül sok egyéb műveletet is definiál, mint például létrehozás, törlés. Olyan kényelmi, extrafunkcionális elemeket is tartalmaz amelyek csak bonyolítják a lekérdezéseket, hogy felhasználóbarátabban adhassák vissza a tartalmat, például a visszatérési referencia átnevezése (`x AS "username"`). Ahhoz, hogy egyszerű lekérdezéseket generáljak nem szükséges ezt a hatalmas metamodellt feldolgozni, viszont egyértelműen meg kell határozni egy olyan részmodelljét, amelyből hiánytalanul előáll az egyszerű lekérdezések nyelvi specifikációja, továbbá a lényegtelen elemek szűrésével a generált modellsereg diverzitását is növeljük, mert így lényeges modellelemek között kell hogy különbözzenek.

   (b) **Kényszerek**:Azonban vannak olyan szabályok amelyeket a metamodell nem tud kifejezni, betartásuk nélkül viszont a generált példánygráfok nem értelmezhetőek Cypher nyelvű lekérdezésekként. Például annak meghatározása, hogy milyen változókra lehet, és milyenekre nem lehet hivatkozni a visszatérési értékben. Ezeket a szabályokat jólformáltsági kényszerekkel tartatom be.

   (c) **Konfigurációs paraméterek**: A generátor működéséhez elengedhetetlen a saját nyelvén íródott konfigurációs fájl. Itt határozható meg, hogy milyen megoldóval működjön a generálás, hogy hányat használjon az egyes elemekből a generálás során, hogy mekkora és milyen mennyiségű példányokat generáljon stb.

   (d) **Esettanumány szignatúra** : A generált példánygráfok változók nélkül jönnek létre. Ahhoz, hogy egy értelmes adatbázison végezhessük el őket, fontos hogy

fel legyenek fegyverezve az adatbázisban használt címkékkel, típusokkal. Mivel én a Train Benchmark által használt adatbázison futtattam lekérdezéseimet, ezért az általa használt szignatúrával szereltem fel a rendszert.

2. **Utófeldolgozás** : Az általam generált gráfokban a változóknak nem adok nevet. Megtehetném, hogy a generálás során kitöltöm őket, de csak úgy, hogy a gráfgenerátor a generált szavakat különbségekként kezelje két példánygráf között. A nagyobb diverzitás elérésének érdekében a generátor nem foglalkozik a változók elnevezésével. Az utófeldolgozás során az Esettanulmány szignatúra szavaival töltöm fel az addig még csonka példánygráfokat.

3. **Fordító** : Az utófeldolgozás során sorosíthatóvá vált példány gráfokat a Cypher nyelv XText nyelven íródott nyelvtanának segítségével szöveges lekérdezésekké alakítom.

4. fejezet

Gráflekérdezések automatikus generálása

4.1. Lekérdezések generálása

4.1.1. Tesztelni kívánt résznyelv kiválasztása

A gráfgenerátor egyik bemenete a tesztelni kívánt nyelvi fregmens metamodellje. Ahhoz, hogy hasznos tesztesetek állítsunk elő, szükséges a generátort egy konkrét feladatot ellátó modellek előállítására konfigurálni. (Ha az egész nyelvet használnánk akkor a lekérdezése érdektelen részletekből állna, például kommenteket, felesleges változó átnevezéseket, lekérdezéssel kapcsolatos metainformációkat generálna.) Munkám során tehát az első feladat az volt, hogy kiválasszam a Cyphernek egy olyan résznyelvét amellyel érdekes lekérdezéseket lehet generálni. Dolgozatomban olyan úgynevezett pozitív mintájú lekérdezések generálására koncentráltam, amelyek vizsgálják a csomópontok típusát illetve a csomópontok közötti kapcsolatok típusát. Ugyanis számos lekérdező nyelvben, mint például Cypher[8] és VIATRA [10] ezek képezik a lekérdezések alapját.

Alább azt kívánom megmutatni, hogy egy példalekérdezés alapján hogyan szűkítettem le a slizaa [13] nevű Xtext alapú Cypher metamodellt.

```
MATCH (train : Train {name : "train1"})-[:ON]->(seg : Segment {name : "seg3"})
RETURN seg
```

A szűkítést a fenti lekérdezés alapján végeztem a következő metódussal:

- Felírtam a fenti lekérdezést a query editorban.
- Kibontottam Xtext segítségével.
- Megnyitottam a nyelvtan metamodelljét.
- A konfigurációs fájlban egy metamodell deklarációs mappába összegyűjtöttem a kibontott fában található osztályokat.
- A nyelvtan alapján kiválogattam az osztályok ősosztályait, illetve a használt/használható asszociációkat.
- A nyelvtanban ezen kívül növeltem a muptiplicitását legalább 1-re azoknak az elemeknek amelyek generálása elengedhetetlen az értelmes lekérdezések létrejöttéhez.

4.1.2. Jólformáltsági kényszerek betartása

A gráfgenerátor másik bemenete olyan kényszerek halmaza, amelyek szükségesek ahhoz, hogy értelmes modelleket tudjunk generálni. Ezekre főleg azért van szükség, mert az Xtextben [3] íródott nyelvtan és a nyelvtan metamodellje közötti konverzió nem tökéletes. Ennek több oka is van: (1) A precíz metamodell meghatározása számításilag komoly kihívást jelent és nem is végezték el a keretrendszer megalkotói, (2) A metamodellnek hibás modellek leírására is alkalmasnak kell lennie (hiszen szerkesztés közben általában félkész modellek vannak a rendszerben) (3) A metamodell a modelleknek csak atz alap struktúráját (referencia hova mutathat) írja le bonyolultabb szabályok (összetett logikai kifejezések) meghatározására alkalmatlan.

Jólformálsági kényszerekre azért van szükség, mert nem minden a metamodellel leírható modell sorosítható Cypher nyelvű lekérdezéssé. Az Xtext Cypher nyelvtanban leírt parseolhatósági szabályokat át kell fordítani ASG-n értelmezett struktúrális kényszerekké.

```
pattern hasReference(retI : ReturnItem, variRef : Expression){
  VariableRef(variRef);
  ReturnItem.expression(retI, variRef);
}

@Constraint(severity ="error", key={ri}, message = "error")
pattern hasNoReference(ri : ReturnItem){
  neg find hasReference(ri, _);
}
```

A fenti kényszerre azért volt szükség, mert a metamodellben nem volt megadva, hogy a `RETURN` szó után kötelező legalább egy változóra referálni is. Pedig ha nem referálnánk itt változót a `MATCH` ágon megfogalmazott mintából, akkor nem lenne értelme a minta megírásának, illetve a válasznak sem.

A felső minta meghatározza hogy hogyan néz ki egy olyan `ReturnItem` aminek van `VariableRef` a visszatérési értékében. Az alsó pedig egy kényszer, ami arra kényszeríti a generátort hogy eldobja az összes olyan modellt, amiben nem találja meg a felső mintát.

```
pattern wellLookingPattern (patt : Pattern, patternElement : PatternElement){
  Pattern.patterns(patt,patternElement);
}
@Constraint
pattern notWellLookingPattern(patt : Pattern){
  neg find wellLookingPattern(patt , _);
}
```

A fenti kényszerre azért volt szükség, mert a metamodell alapján egy `Pattern`-be a `.patterns` tulajdonság beállításakor sok féle elem kerülhetett volna, viszont sok akár a mostaninál bonyolultabb példalekérdezés során sem volt precedens arra, hogy nem `PatternElement` -ek kerültek bele. Ezért úgy döntöttem, hogy a konzisztensebb és gyorsabb generálás érdekében megtiltom a generátornak, hogy jóként fogadjon el más megoldást.

```
@Constraint
pattern  patternElementHasVar(pE : PatternElement, vari: VariableDeclaration){
  PatternElement.^var(pE,vari);
}

@Constraint
```

```
pattern  patternElementHasPart(pE : PatternElement, pp : PatternPart){
  PatternElement.part(pE,pp);
}
```

A fenti két kényszerre azért volt szükség, mert a `PatternElement` osztály örököl a `PatternPart` osztálytól egy `var` és egy `part` tulajdonságot, pedig ilyen tulajdonságokkal nem szabadna rendelkeznie, csak ez az információ elveszett a Cypher Xtext nyelvtanának metamodellé generálódása során. Ezért hibásnak nyilvánítom azokat a `PatternElement`-eket, amelyek rendel a két tulajdonság valamelyikével.

```
pattern pe(pe:PatternElement) {
  PatternElement(pe);
}

@Constraint
pattern notPatternElement( pp : PatternPart){
  neg find pe(pp);
}
```

A munkám során vizsgált példalekérdezések között nem volt olyan `PatternPart`, ami nem volt `PatternElement`. Viszont a generátor számára meghatározott metamodellben mindkét osztály szerepel, és egyik sem absztrakt, így a generátor sokat bajlódna `PatternPart`-ok generálásával, ezzel fontos időt veszítve. Ettől a lehetőségtől a fenti kényszer segítségével megfosztom.

```
@Constraint
pattern notWellLookingMapliteral(mapLiteral : MapLiteral, nodeLabel: NodeLabel){
  MapLiteral.nodeLabels(mapLiteral,nodeLabel);
}
```

A `Mapliteral` osztály örököl egy `nodelabels` tulajdonságot, és ezáltal nyelvtanilag helytelenné válik, mivel a `Mapliteral`-ok a következőképpen néznek ki a Cypher nyelven: `name : "seg3"` és egyéb elemek nem kerülhetnek bele.

```
pattern wellDeepMap(mapLiteralEnrty: MapLiteralEntry, string : StringLiteral){
  MapLiteralEntry.value(mapLiteralEnrty,string);
}

@Constraint
pattern notWellDeepMap(mapLiteralEnrty : MapLiteralEntry){
  neg find wellDeepMap(mapLiteralEnrty, _);
}
```

Erre a mintára pedig azért van szükség, mert ha nem lennének akkor a következőhöz hasonló értelmetlen lekérdezések generálása válna lehetővé:

```
MATCH (seg : Segment {name : {name2 : {name3: { name4 : seg4}}}})
RETURN seg
```

És a generátor nagy méretű modelleknél könnyen esne abba a hibába hogy az extra csúcspontokat így pocsékolja el.

4.1.3. Diverzitás biztosítása

A generállt modellszekvencia használhatatlan lenne , ha a lekérdezések között nem , vagy csak lényegtelen különbségek lennének. Dolgozatomban több féle szinten is garantáltam a lekérdezések diverzitását (elkerülve ezzel hogy túlságosan hasonló lekérdezések szülessenek). A diverzitás biztosítása kiemelten fontos tesztgenerálási feladatoknál, mert így hatékony ekvivalencia partícionálást biztosíthatunk. Generálás során az alábbi lényegi különbségeket garantáljuk.

- Egyenértékű változók elnevezésétől függetlenül azonosnak találjuk az alábbi két megoldást, hiszen a válasz a két lekérdezésre azonos értékeket adna vissza.

```
MATCH ( V1 : Segment  )-->( V2 : Segment ) RETURN V1 , V2
MATCH ( Var1 : Segment  )-->( Var2 : Segment) RETURN Var1 , Var2
```

- A változók sorrendezésétől függetlenül azonosnak tekintjük az alábbi két problémát, hiszen a két lekérdezés ugyanazt a táblázatot adná vissza csupán az oszlopokat fordított sorrendben jelenítené meg.

```
MATCH ( V1 : Segment  )-->( V2 : Segment ) RETURN V1 , V2
MATCH ( V1 : Segment  )-->( V2 : Segment ) RETURN V2 , V1
```

- A attribútomok ellenőrzésének sorrendje nem számít, hiszen ugyanannak a csomópontnak attribútumairól van szó, mindegy milyen sorrendben írjuk.

```
MATCH ( Var1 : Segment { signal : "String1", currentPosition : "String2" })
RETURN Var1
MATCH ( Var1 : Segment { currentPosition : "String2", signal : "String1" })
RETURN Var1
```

- Illetve a vesszővel elválasztott minta részek sorrendje sem változtat a lekérdezés eredményén, mert ezek metszetét értékeli ki a lekérdező rendszer.

```
MATCH ( V1 : Segment { signal : "String1"} ) ,
    ( V2 : Route { length : "String2" } )
RETURN V1 ,V2
MATCH ( V2 : Route { length : "String2" } ) ,
    ( V1 : Segment { signal : "String1"} )
RETURN V1 ,V2
```

A fenti példák nem tartalmaznak lényegi szűrést, továbbra is az összes lehetséges lekérdezés generálható marad.

Továbbá az alábbi extra diverzitást is biztosítottam a lekérdezések létrehozása során:

- Két struktúrálisan hasonló lekérdezést nem különböztetünk meg. Struktúrálisan hasonlónak azokat a lekérdezéseket tekintem amelyek a típusnevek átírásával azonossá válnának.

```
MATCH ( V1 : Segment  )-->( V2 : Segment ) RETURN V1 , V2
MATCH ( V1 : Route  )-->( V2 : Switch ) RETURN V1 , V2
```

- A literál értékeket is hasonlónak tekintjük az előző ponthoz hasonlóan ez is a struktúrális különbségek létrehozásnak érdekében történik.

```
MATCH ( Var1 : Segment { signal : "String1" }) RETURN Var1
MATCH ( Var1 : Segment { signal : "String2" }) RETURN Var1
```

- A generátoron belül a diverzitás szintet magasra állítottam. Ez aszerint növeli a különbségeket két gráf között, hogy megvizsgálja a csomópontok szomszédságát (a csomóponttal szomszédos csomópontok halmaza). És nem generál több, csak azonos szomszédságokból álló gráfot.

4.2. Utófeldolgozás

Az előző fejezetben kifejtettem miért jó a változókat csak utólag elnevezni. Ezt a feladatrészt az utófeldolgozás során végzem el. Itt adom hozzá ezen kívül azokat a részleteket is a modellekhez, ami mindegyikben azonos, ezért a generálás során nem foglalkoztam vele.

Ahhoz, hogy értelmesen nevezzem el az egyes változókat, a Train Benchmarkban használt kifejezéseket és értékeket osztom ki, amik láthatóak a 2.3. ábrán. Három féle elemet kell elneveznünk a jelenleg generált modellekben. (1) csomópont címkék `NodeLabel.labelName` , (2) kapcsolat címkék `RelationshipDetail.relTypeNames` és (3) tulajdonság címkék `MapLiteralEntry.key`. Ezek mind megfeleltethetőek az ábrán látható elemeknek. A csomópont címkék az osztályok neveinek, a kapcsolat címék az asszociációk neveinek, míg a tulajdonságcímkék az attribútumoknak. Ezért készítettem három listát a megfelelő nevekkel.

1. csomópont címkék : Region, Route, Segment, Semaphore, Sensor, Switch, SwitchPosition
2. kapcsolat címkék : connectsTo, entry, exit, follows, monitoredBy, monitors, ”requires”, ”target”
3. tulajdonság címkék : id, active, position, currentPosition, length, signal

Ezután szükségem volt egy randomizáló függvényre, amely a megfelelő nevek valamelyikét elhelyezi egy-egy elemen. Majd bekötöttem mindhárom típus minden elemére a megfelelő szavakat. Ezen kívül a lekérdezésekben vannak még változók is amelyeket szisztematikusan elneveztem V1...Vn-el, illetve literálok amiket pedig ”String1”...”Stringn”-el neveztem el.

Ezzel lehetnek olyan problémák, hogy a lekérdezések értelmetlennek bizonyulnak, például:

```
MATCH ( Var1 : Segment { signal : "String1" }) RETURN Var1
```

Hiszen a Train Benchmark metamodellje alapján egy szegmensnek nincsen singnal tulajdonsága, tehát egy metamodell alapján felépített gráf adatbázisban nem találnánk a lekérdezésben szereplő mintát. A Neo4j gráfadatbázisára viszont igaz, hogy csak enyhén típusos, ezért igazából nem kezel metamodelleket adatbázis szinten, ha akarnánk bele tudnánk írni egy olyan csomópontot, ami Segment címkét kap és signal tulajdonsággal rendelkezik, és a felhasználókat semmi nem akadályozza meg abban, hogy esetleg értelmetlen gráfadatbázist hozzanak létre. így amikor keresünk egy ilyen adatbázisban fel kell hogy legyen készülve ara, hogy olyan dolgot keresünk amit nem találhatunk meg.

4.3. Fordítás

A generált gráfokat az utófeldolgozás után kifejezések ASG-je ként értelmezem, majd a nyelvtani szabályok fordított irányú alkalmazásával szöveges dokumentummá alakítom. A fordító a szöveggé alakítás során szóközöket rak a szükséges helyekre, behelyettesíti a változók neveit a rájuk mutató referenciákba, kitölt a nyelvtani szabályokban meghatározott szavakat pl: `MATCH` , `RETURN` , illetve megfelelő helyekre zárójeleket rak. A fordítást az Xtext keretrendszerének segítségével automatizáltam.

5. fejezet

Értékelés

Ebben a fejezetben kiértékelem munkámat mialatt megválaszolom a következő kérdéseket:

- **Kérdés1**: Hogy aránylik egymáshoz az előfeldolgozás, a generálás és az utófeldolgozás időtartama?
- **Kérdés2**: Hogy skálázódik a generálás modell méret szempontjából?
- **Kérdés3**: Hogy skálázódik a generálás modell darabszám szempontjából?
- **Kérdés4**: Mennyire diverzek a lekérdezések? (egymás utáni 50 illetve 50 független)

5.1. Mérési környezet felállítása

A méréseket eclipse fejlesztői környezetben végeztem. Ahhoz, hogy bemelegítsem a modell generátort memóriakezelés és optimalizálás szempontjából 5 extra futást adtam hozzá minden kiértékelt mérés előtt. A mérésekhez a generátor számára 4000 MB memóriát biztosítottam, és ez mindig elegendőnek bizonyult. Az összes mérést egy egyszerű asztali számítógépen végeztem (Intel Core i7-3520M CPU, 2.90GHz, Windows 10 Pro). A generáláshoz részmodellként egy 22 elemből álló ASG-t adtam meg, de csak az esszenciális részletek specifikálásával.

5.2. K1

5.2.1. Specifikáció

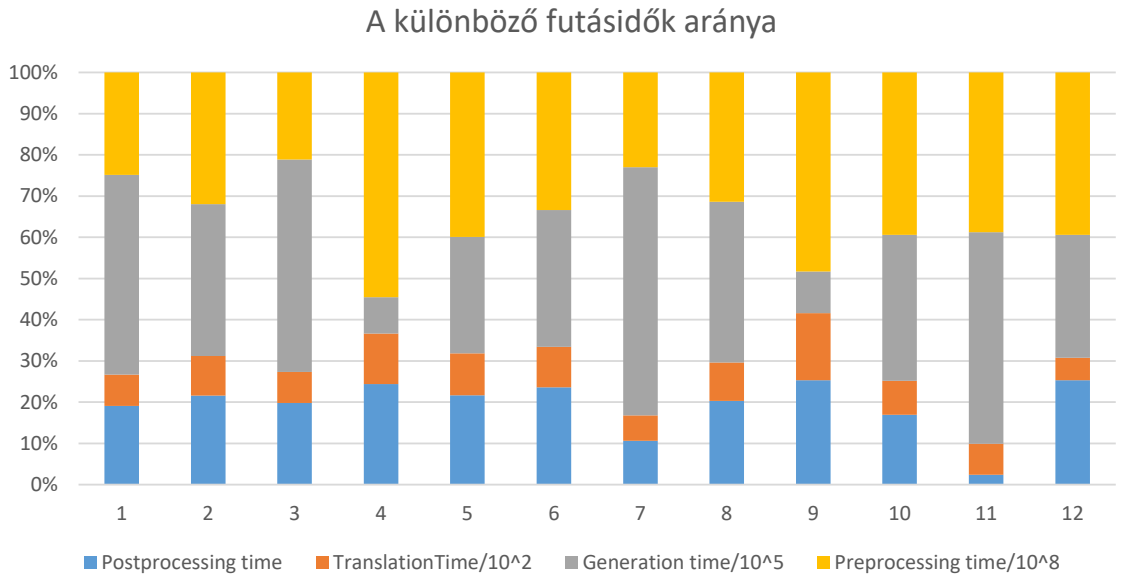
A kérdés megválaszolásához 12 mérést végeztem, 50 példány gráfot generáltam 10 hozzáadott elemmel. A kölünböző futásidőket egy helyre összegyűjtöttem és az adatokat transzformáltam befogadható formátumra.

5.2.2. Eredmények

A mérés eredményei az 5.1. ábrán láthatóak. Az ábra segítségével azt szeretném bemutatni, hogy a különböző futásidők nagyságrendileg eltérnek egymástól. A grafikon vízszintes tengelyén pedig az ábrázolt négy érték össszegét tekintjük 100 százaléknak.

A négy ábrázolt érték pedig rendre: Az utófeldogozás ideje(kék), a fordítás idejének század része (narancs), a generálás idejének tízezrede (szürke), és az előfeldolgozás idejének százmilliomoda (sárga). Tehát ezek szerint a különböző tényezők aránya a következő:

- utófeldolgozási idő * 10 fordítási idő * 2



5.1. ábra. A K1 mérés eredményei

- fordítási idő * 3 * 1000 generációs idő
- generációs idő * 1000 előfeldolgozási idő

5.3. K2

5.3.1. Specifikáció

A kérdés megválaszolásához 3 al mérést végeztem. A három mérés egymástól csak a generált modellek minimális méretében különbözött, az elsőben 10 a másodikban 20 a harmadikban pedig 30 elemet generáltam a részmodellben megadottak mellé. Mindhárom almérést 12 alkalommal végeztem el, és minden alkalommal 50 példányt generáltam.

5.3.2. Eredmények

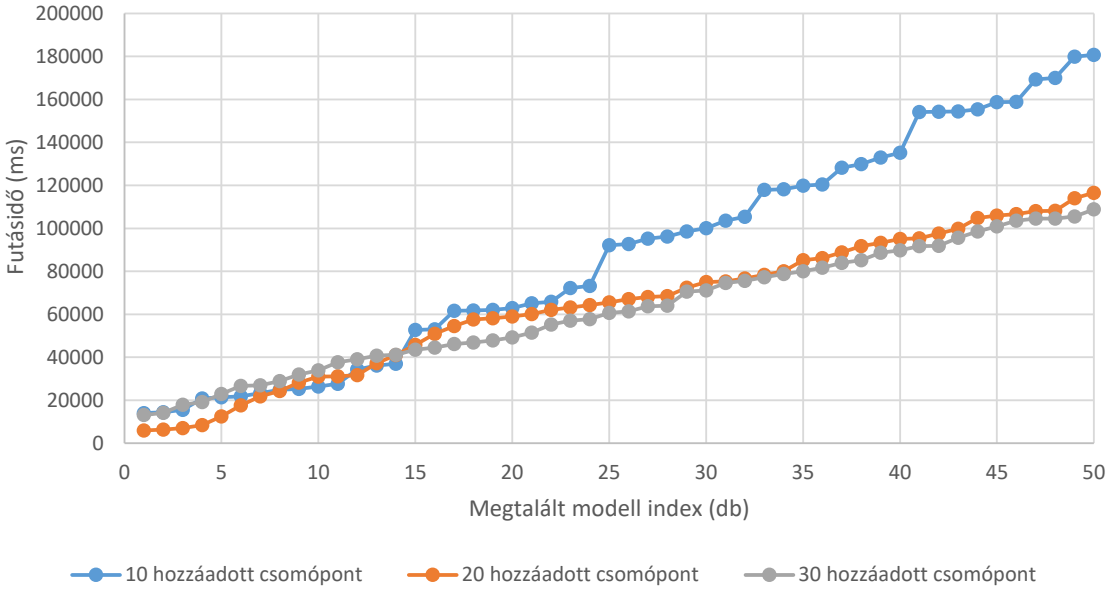
A mérés eredményei az 5.2. ábrán láthatóak.

Az 50 darab generált modell megtalálásának ideje nem volt azonos a 12 alkalommal, ezért a 12 érték mediánját kiválasztottam, az ábrán ezen medián értékek táthatóak. A 3 almérés eredményeit 3 színnel jelöltem. Az ábrán a vízszintes tengelyen a modellek láthatóak a megtalálás sorrendjében, a függőleges tengelyen pedig a futásidők mediánja milliszekundumban.

A három mérés eredményei külön-külön megfeleltethetőek egy egyenes trendvonalnak. Az egyre későbbi modellek megtalálása a diverzitás biztosítása miatt növekszik, egyre nehezebb ugyanis szignifikánsan különböző modelleket találni.

Az ábrán látható, hogy a generátor nehezebben boldogult a kis modellek megtalálásával mint a nagyokéval. Ez azzal magyarázható, hogy a megadott részmodell elég nagy,(22 elem) emiatt tovább kell keresgélni, hogy 10 hozzáadott elemből ki tudja tölteni az összes szükséges helyet és még különböző is legyen.

K3 : B(12x, 1db, 5-10-15-...50), B+(12x, 10, 5-10-15-...50) , B++=(12x, 30, 5-10-15-...50)



5.2. ábra. A K2 mérés eredményei

5.4. K3

5.4.1. Specifikáció

A kérdés megválaszolásához 13 al mérést végeztem. Minden al mérést 10 alkalommal végeztem el, és minden alkalommal 10 példány modellt generáltam. A 13 mérés során rendre 5, 10 ,15, ...,50, 100, 150, 200 elemet adtam hozzá a részmodellhez.

5.4.2. Eredmények

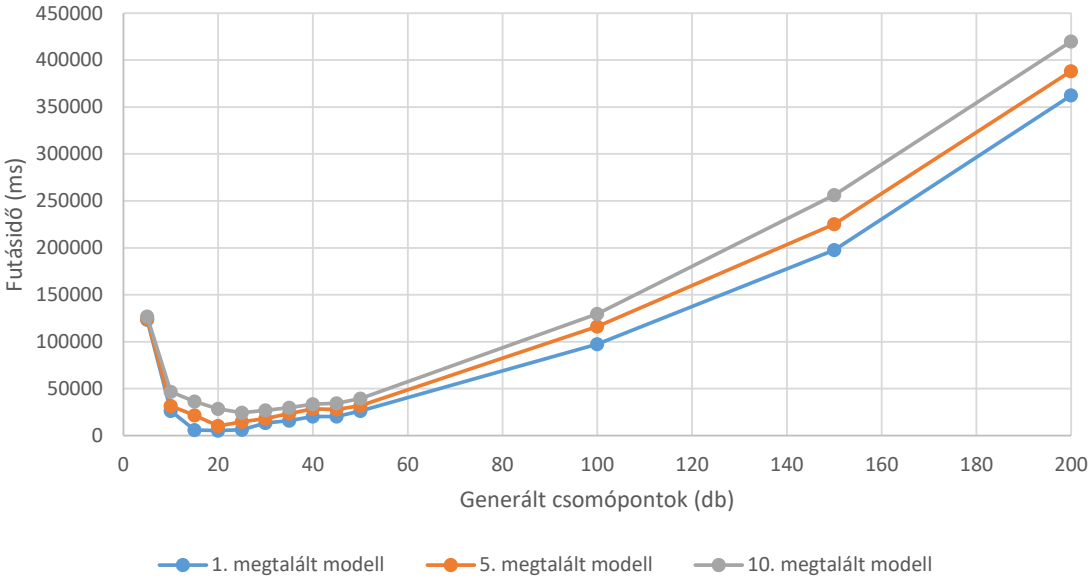
A mérés eredményei az 5.3. ábrán láthatóak.

Mind a 13 alkalommal vizsgáltam az 1., az 5. és a 10. modell megtalálásához szükséges időt. Az előző méréshez hasonlóan most is rendre a 12 identikus mérés futásidejének mediánját ábrázoltam. A vízszintes tengelyen a hozzáadott elemek darabszáma, míg a függőleges tengelyen az adott modell megtalálásához szükséges futásidő látható.

Itt is látható, hogy a kis darabszámú generált elem hozzáadásával nehezebben boldogult a generátor, mint a közepes elemszámokkal, ám a hatalmas modellek megtalálása is egyre nehezebb feladatnak bizonyult. Ha a 3 görbéhez illeszkedő trendvonalat keresünk és az elejét vagy a végét levágjuk, akkor egy köbös függvényt tudunk illeszteni a szakaszokra. Mit a legtöbb np teljes problémára erre is igaz, hogy a megfelelő egyszerűsítésekkel és megoldásokkal exponenciális helyet köbösre egyszerűsítődik le a probléma.

A három vonal ezen kívül azt is mutatja, hogy az első modellt gyorsabb megtalálni mint az 5-et, és az 5.-et gyorsabb mint a 10.-et. Ez továbbra is a diverzitás fenntartásának nehézségéből fakad.

5.5. Diverzitás mérése



5.3. ábra. A K3 mérés eredményei

6. fejezet

Összefoglalás

6.1. Elméleti eredmények

6.2. Gyakorlati eredmények

6.3. Jövőbeli tervek

Köszönetnyilvánítás

Ez nem kötelező, akár törölhető is. Ha a szerző szükségét érzi, itt lehet köszönetet nyilvánítani azoknak, akik hozzájárultak munkájukkal ahhoz, hogy a hallgató a szakdolgozatban vagy diplomamunkában leírt feladatokat sikeresen elvégezze. A konzulensnek való köszönetnyilvánítás sem kötelező, a konzulensnek hivatalosan is dolga, hogy a hallgatót konzultálja.

Irodalomjegyzék

[1] 2018 October 16 – 2018 October 15: The neo4j graph platform – the #1 platform for connected data. URL `https://neo4j.com/`.

[2] Ilene Burnstein: *Practical software testing: a process-oriented approach*. 2006, Springer Science & Business Media.

[3] Sven Efftinge – Miro Spoenemann: Why xtext?
URL `https://www.eclipse.org/Xtext/#feature-overview`.

[4] Graph database | multi-model database. URL `https://orientdb.com/`.

[5] Richard Gronback: Eclipse modeling framework (emf).
URL `https://www.eclipse.org/modeling/emf/`.

[6] High-performance human solutions for extreme data.
URL `http://sparsity-technologies.com/#sparksee`.

[7] József Marton – Gábor Szárnyas – Márton Búr: Model-driven engineering of an opencypher engine: Using graph queries to compile graph queries. In *International SDL Forum* (konferenciaanyag). 2017, Springer, 80–98. p.

[8] Neo4j's graph query language: An introduction to cypher.
URL `https://neo4j.com/developer/cypher-query-language/`.

[9] Oracle database technologies.
URL `https://www.oracle.com/database/technologies/index.html`.

[10] Scalable reactive model transformations. URL `https://www.eclipse.org/viatra/`.

[11] Oszkár Semeráth – Ágnes Barta – Ákos Horváth – Zoltán Szatmári – Dániel Varró: Formal validation of domain-specific languages with derived features and well-formedness constraints. *Software & Systems Modeling*, 16. évf. (2017) 2. sz., 357–392. p.

[12] Oszkár Semeráth – Dániel Varró: Iterative generation of diverse models for testing specifications of dsl tools. In *International Conference on Fundamental Approaches to Software Engineering* (konferenciaanyag). 2018, Springer, 227–245. p.

[13] Slizaa: slizaa/slizaa-opencypher-xtext, 2018. Aug.
URL `https://github.com/slizaa/slizaa-opencypher-xtext`.

[14] Gábor Szárnyas – Benedek Izsó – István Ráth – Dániel Varró: The train benchmark: cross-technology performance evaluation of continuous model queries. *Software & Systems Modeling*, 17. évf. (2018) 4. sz., 1365–1393. p.

[15] Viatra: viatra/viatra-generator, 2018. Oct.
URL `https://github.com/viatra/VIATRA-Generator`.