



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	i
Abstract	ii
1. Bevezetés	1
2. Előismeretek	2
2.1. Esettanulmány	2
2.2. Gráflekérdező rendszerek	3
2.2.1. Tulajdonság gráfok	3
2.2.2. Neo4j	3
2.3. Modellezés és metamodellezés	4
2.3.1. Cypher query-k metamodellje	4
2.3.2. Xtext	5
2.3.3. VIATRA jólfarmáltsági kényszerek	6
2.4. Gráfgenerálás	7
3. Áttekintés	8
3.1. Funkcionális áttekintés	8
3.2. Lekérdezés generálási folyamat felépítése	9
4. Gráflekérdezések automatikus generálása	11
4.1. Lekérdezések generálása	11
4.1.1. Tesztelni kívánt résznyelv kiválasztása	11
4.1.2. Jólformáltsági kényszerek betartása	12
4.1.3. Diverzitás biztosítása	13
4.2. Post processing	13
4.3. Fordítás	14
5. Értékelés	15
5.1. Mérési környezet felállítása	15
5.2. Benchmark eredmények	15
5.3. Feature fedés eredmények	15
5.4. Diverzitás mérése	15
6. Összefoglalás	16
6.1. Elméleti eredmények	16
6.2. Gyakorlati eredmények	16
6.3. Jövőbeli tervek	16
Köszönetnyilvánítás	17

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 lekérdezőként 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 lekérdezéseként, 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 válaszüzeneteket 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 ...

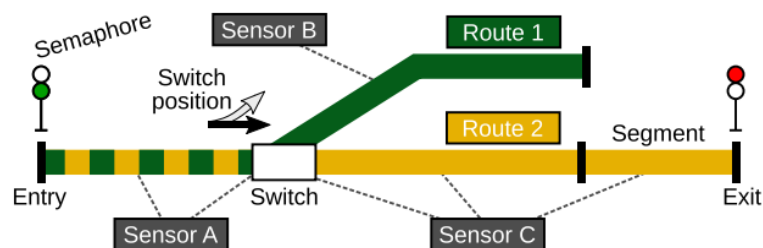
2. fejezet

Előismeretek

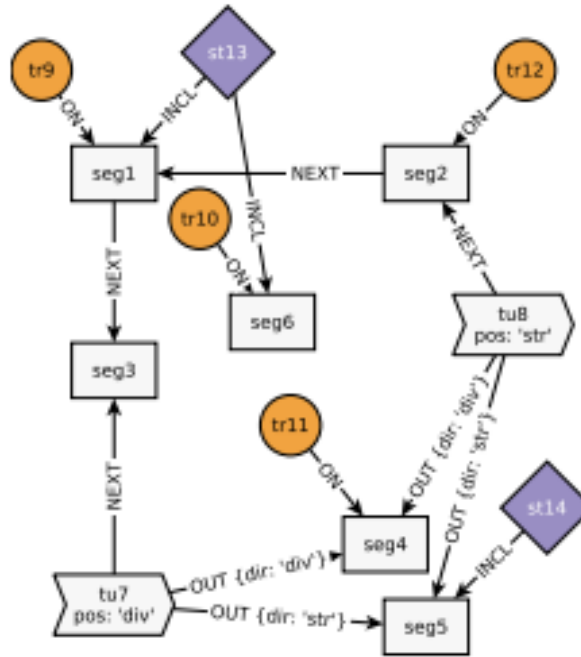
2.1. Esettanulmány

A dolgozatban elért eredményeket a Train Benchmark [12] esettanulmány segítségével fogom bemutatni. Ez a benchmark azért jött létre, hogy össze tudjuk hasonlítani különböző gráflekérdező rendszerek teljesítményét, beleértve gráfadatbázisokat, mint Neo4j [1], SparkSee [5], nagy teljesítményű modellezőeszközöket, mint VIATRA [9] és szokványos relációs adatbázisokat, mint az Oracle [8], 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: [12])



2.2. ábra. Tulajdonsággráf példa(forrás: [6])

2.2. Gráflekérdező rendszerek

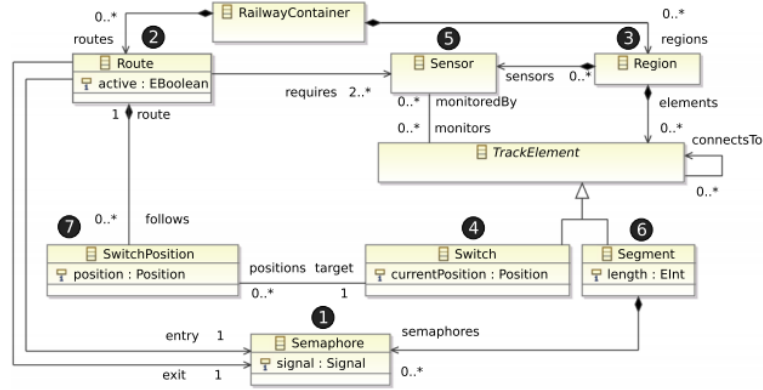
2.2.1. Tulajdonság gráfok

A gráfok intuitív formalizációt nyújtanak modellezési szempontból arra, hogy úgy írhasuk le a világot ahogy az ember gondolkozik róla. Tehát mint dolgok (csomópontok) és köztük lévő kapcsolatok (élek)[6]. 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 [3] és a SparkSee [5].

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 választva a lekérdező rendszerről, ezért az képes a Cypher nyelven írt lekérde-



2.3. ábra. A Train Benchmark metamodellje.(forrás: [12])

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 címüket 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.

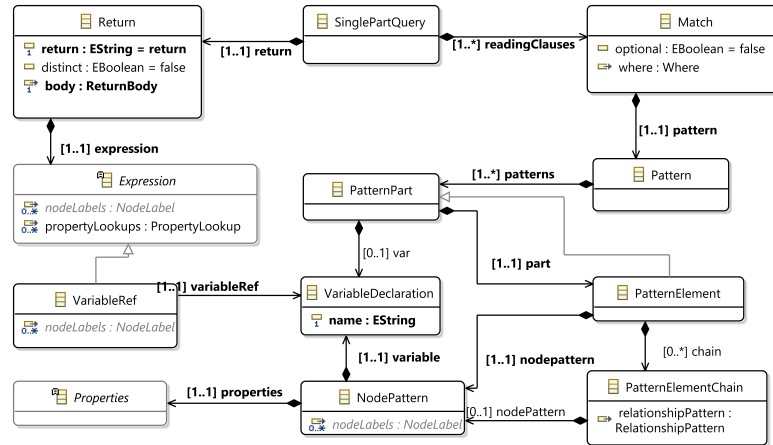
2.3. Modellezés és metamodellezés

A metamodellezés egy technika 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 [10]. Dolgozatomban az Eclipse Modeling Framework (EMF) [4] -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 metamodel

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 nyelvnek célformátumát is. Ahhoz, hogy a Cypher nyelven megírt lekérdezéseket értelmezni lehessen a 2.4-ös metamodellel megismert elemek szintjén egy Xtext [2] keretrendszerben íródott nyelvtanra van szükség, amelyből a metamodellel automatikusan elkészült. A dolgozatomban a slizaa[11] nevű Xtext alapú Cypher nyelvtant használtam.

ASG

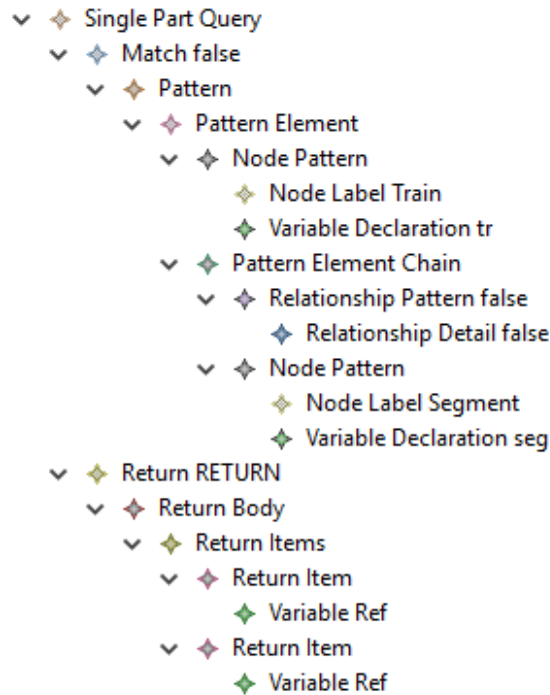
```
SinglePartQuery:
  (readingClauses+=ReadingClause)* return=Return ;

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 **"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 lekérdezésen.

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



2.5. ábra. A példa lekérdezés kibontása Xtext-tel

```
RETURN tr, seg
```

Ezt a lekérdezést Xtext segítségével ősosztlyaira lehet bontani, és megnézni, mi mi-csoda benne. Erről a 2.5 ábrán láthatunk egy példát.

2.3.3. VIATRA jólformáltsági kényszerek

Az Eclipse VIATRA keretrendszer [9] 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 metamodel önmagában nem lenne alkalmas. A lekérdezés generálás során erre használok.

```

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 [13] 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 metamodellel 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 metamodellel specifikációinak és kielégíti az összes jólformáltsági kényszert. Struktúrájukban pedig különböznek egymástól biztosítva ezzel tesztkészlet diverzitását. Én ezt a keretrendszert használom.

3. fejezet

Áttekintés

3.1. Funkcionális áttekintés

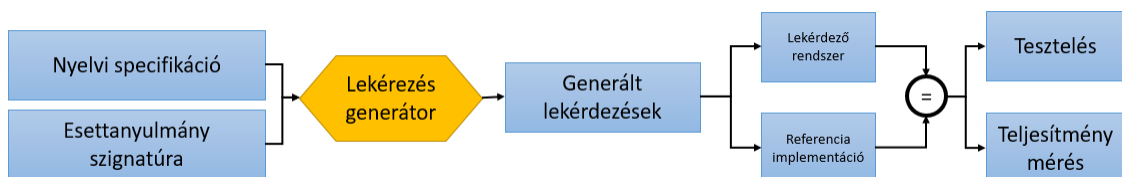
Munkám célja hogy mutasson egy olyan megközelítést, amelynek segítségével lehetséges gráfmintaillesztő rendszerek tesztelése. Az elképzelést a 3.1 ábrán mutatom be.

Az ötlet lényege az, hogy a tesztelni kívánt rendszer nyelvi specifikációjának és egy esettanulmány szignatúrájának (ezalatt egy olyan property gráf 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.

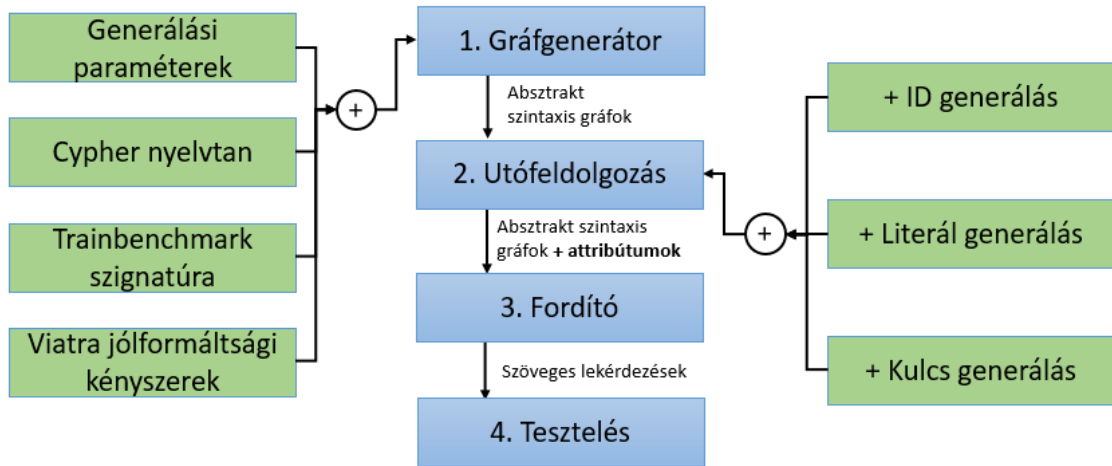
Amint rendelkeznék egy tesztkészletnyi ilyen lekérdezéssel azokat tudnám futtatni azon az adatbázison amelynek szignatúráját esettanulmányként választottam. Ha az ezzel a módszerrel generált lekérdezésekre adott válaszok válaszüzeit összehasonlítjuk több rendszeren tudnánk teljesítményben tesztelni azokat. Illetve ha a generált lekérdezések helyes eredményével is rendelkeznék akkor 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 másikon működnek, így azt is tesztelni tudnánk hogy mekkora az egyes lekérdező motorok funkcionális lefedettsége.

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. //TODO mi az? 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.

A megközelítésemet egy Neo4j [1] property gráf adatbázison mutatom be, amely a Train benchmark [12] által használt szignatúrával van felszerelve. A lekérdezéseket a Neo4j által kifejlesztet Cypher [7] nyelven generálom, a slizaa [11] által készített openCypher nyelvi specifikáció felhasználásával.



3.1. ábra. Az elképzelés funkcionális áttekintése



3.2. ábra. A lekérdezés generálási folyamat áttekintése

3.2. Lekérdezés generálási folyamat felépítése

A folyamat felépítését a 3.2 -es ábra segítségével ismertetem.

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 metamodel a nyelv egészére kiterjed. Így lekérdezéseken kívül sok egyéb műveletet is definiál, mint például létrehozás, törlés. Olyan elemeket is tartalmaz amelyek csak bonyolítják a lekérdezéseket, hogy felhasználó centrikusabban adhassák vissza a tartalmat, például a visszatérési referencia átnevezése, az adatok csökkenő sorrendbe rendezése. Ahhoz, hogy egyszerű lekérdezéseket generáljak nem szükséges ezt a hatalmas metamodelt feldolgozni, viszont egyértelműen meg kell határozni egy olyan részmodelljét, amelyből hiánytalanul előáll az egyszerű lekérdezések nyelvi specifikációja.
 - (b) **Kényszerek:** Azonban vannak olyan szabályok amelyeket a metamodel nem tud kifejezni, betartásuk nélkül viszont a generált példánygráfok nem értelmezhetők Cypher nyelvű lekérdezéseként. 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 példányokat generáljon stb.
 - (d) **Esettanumány szignatúra :** A generált példánygráfok változók nélkül jönnek létre. Ahhoz, hogy egy értelmes adatbázison végezhesük el őket, fontos hogy legyenek fegyverezve annak az adatbázisnak a címékével, típusaival. Ezért hát össze készítettem több halmaznyi szót, ami a Train benchmark által használt adatbázis címkéiből áll.
2. **Utófeldolgozás :** Az általam generált gráfokban a változóknak nem adok értéket. 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 döntöttem úgy hogy üresen hagyom az értékeket. 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[7] és VIATRA [9] ezek képezik a lekérdezések alapját.

Alább azt kívánom megmutatni, hogy egy példalekérdezés alapján hogyan szűkítettem le a slizaa [11] 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 módszerrel:

- Felírtam a fenti lekérdezést a query editorban.
- Kibontottam Xtext segítségével.
- Megnyitottam a nyelvtan metamodelljét.
- A konfigurációs fájlban egy metamodellek deklarációs mappába összegyűjtöttem a kibontott fájlban található osztályokat.
- A nyelvtan alapján kiválogattam az osztályok őssztályait, illetve a használt/használható asszociációkat.
- A nyelvtanban ezen kívül növeltem a multiplicitását 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 Xtext-ben [2] íródott nyelvtan és a nyelvtan metamodellje közötti konverzió nem tökéletes. Ennek több oka is van: (1) A precíz metamodell meghatározása számításlag 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 az alap struktúráját (referencia hova mutathat) írja le bonyolultabb szabályok (összetett logikai kifejezések) meghatározására alkalmatlan.

Jólformáltsági kényszerekre azért van szükség, mert nem minden a metamodelllel 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 (~pattern : Pattern, patternElement : PatternElement){
    Pattern.patterns(~pattern,patternElement);
}

@Constraint pattern notWellLookingPattern(~pattern : Pattern){
    neg find wellLookingPattern(~pattern , _);
}
```

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.

```
pattern wellLookingPatternElement(patternElement : PatternElement, nodePattern : NodePattern){
    PatternElement.nodepattern(patternElement, nodePattern);
}

@Constraint pattern notWellLookingPatternElement(patternElement : PatternElement){
    neg find wellLookingPatternElement(patternElement, _);
}
```

```

@Constraint(severity = "error", key={patternElement}, message = "error")
pattern patternElementHasVar(patternElement : PatternElement, variable : VariableDeclaration){
PatternElement.^var(patternElement,variable);
}

@Constraint(severity = "error", key={patternElement}, message = "error")
pattern patternElementHasPart(patternElement : PatternElement, part : PatternPart){
PatternElement.part(patternElement,part);
}

```

```

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

@Constraint(severity = "error", key={patternPart}, message = "error")
pattern notPatternElement( patternPart : PatternPart){
neg find pe(patternPart);
}

```

```

@Constraint(severity = "error", key={mapLiteral}, message = "error")
pattern notWellLookingMapliteral(mapLiteral : MapLiteral, nodeLabel: NodeLabel){
MapLiteral.nodeLabels(mapLiteral,nodeLabel);
}

```

```

//ahhoz kell hogy ne ott legyenek 2 lekérezés hogy milyen mélyen címkézett
pattern wellDeepMap(mapLiteralEntry: MapLiteralEntry, string : StringLiteral){
MapLiteralEntry.value(mapLiteralEntry,string);
}

@Constraint(severity = "error", key={mapLiteralEntry}, message = "error")
pattern notWellDeepMap(mapLiteralEntry : MapLiteralEntry){
neg find wellDeepMap(mapLiteralEntry, _);
}

```

A fenti két kényszer azt biztosítja, hogy a lekérdezésnek, amit generálunk mindenképpen meglegyen a két fő része. A match részben határozzuk meg azt a bizonyos mintát, amit keresünk, a return részben pedig visszatérünk az erre a mintára illeszkedő gráfrészletekkel.

Ezen belül ki kell kössük szintén, hogy a két fő rész ne csak csonkként vagy helytelenül generálódjon, hanem az egész elengedhetetlen részfa megszülessen.

Ezért van szükség a return ágon az alábbi kényszerekre:

4.1.3. Diverzitás biztosítása

pár mondat

4.2. Post processing

itt egy összefoglaló pár mondatban, ha esetleg valaki kihagyta volna az előző fejezetet, hogy mit jelent az alcím.

esettanulmány címkehalmoz megadása

hogy mondom meg mi hova kerülhet

mi alapján tettem azokat ide amiket

4.3. Fordítás

itt egy összefoglaló pár mondatban, ha esetleg valaki kihagyta volna az előző fejezetet, hogy mit jelent az alcím.

kód
hogyan olvasok be
kód
hogyan írok ki
kód
nyelvtan ami alapján dolgozok
lekérdezés
ilyen a végtermék

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

K2 : A : (12x, 50db, 20 db elem), A+(12x, 50db, 10 db elem), A-(12x, 50db, 30 db elem)
K3 : B(12x, 1db, 5-10-15-...50), B+(12x, 10, 5-10-15-...50) , B++=(12x, 30, 5-10-15-...50)

5.2. Benchmark eredmények

5.3. Feature fedés eredmények

5.4. Diverzitás mérése

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] Sven Efftinge–Miro Spoenemann: Why xtext?
URL <https://www.eclipse.org/Xtext/#feature-overview>.
- [3] Graph database | multi-model database. URL <https://orientdb.com/>.
- [4] Richard Gronback: Eclipse modeling framework (emf).
URL <https://www.eclipse.org/modeling/emf/>.
- [5] High-performance human solutions for extreme data.
URL <http://sparsity-technologies.com/#sparksee>.
- [6] 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.
- [7] Neo4j’s graph query language: An introduction to cypher.
URL <https://neo4j.com/developer/cypher-query-language/>.
- [8] Oracle database technologies.
URL <https://www.oracle.com/database/technologies/index.html>.
- [9] Scalable reactive model transformations. URL <https://www.eclipse.org/viatra/>.
- [10] 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.
- [11] Slizaa: slizaa/slizaa-opencypher-xtext, 2018. Aug.
URL <https://github.com/slizaa/slizaa-opencypher-xtext>.
- [12] 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.
- [13] Viatra: viatra/viatra-generator, 2018. Oct.
URL <https://github.com/viatra/VIATRA-Generator>.