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

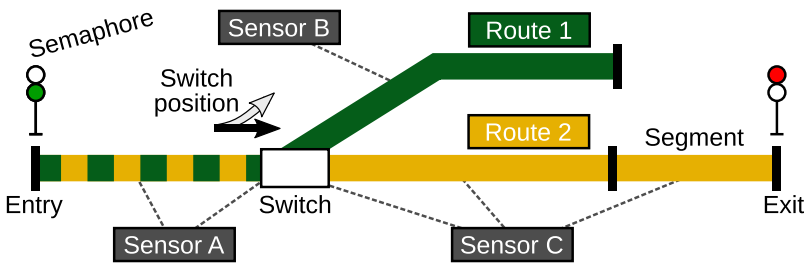
A dolgozatban elért eredményeket a Train Benchmark [29] esettanulmány segítségével fogom bemutatni. Ez a benchmark azért jött létre, hogy össze tudjuk hasonlítani különböző gráflekérdező rendszerek teljesítményét, beleértve gráfadatbázisokat, mint Neo4j [1], SparkSee [16], nagy teljesítményű modellezőeszközöket, mint VIATRA [24] és szokványos relációs adatbázisokat, mint az Oracle [23], 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.

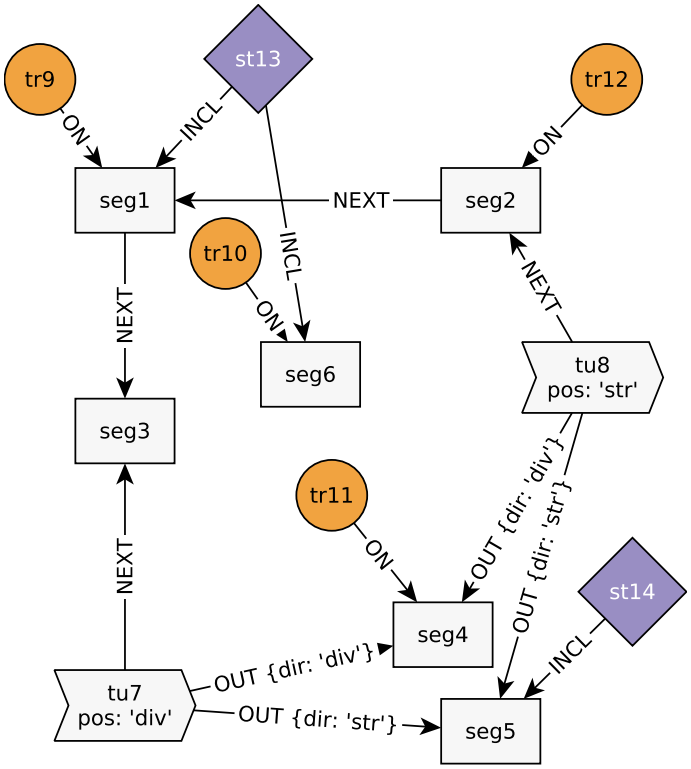
2.2. Gráflekérdező rendszerek

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



2.1. ábra. Vasúti útvonal részlet (forrás: [29])



2.2. ábra. Tulajdonsággráf példa(forrás: [19])

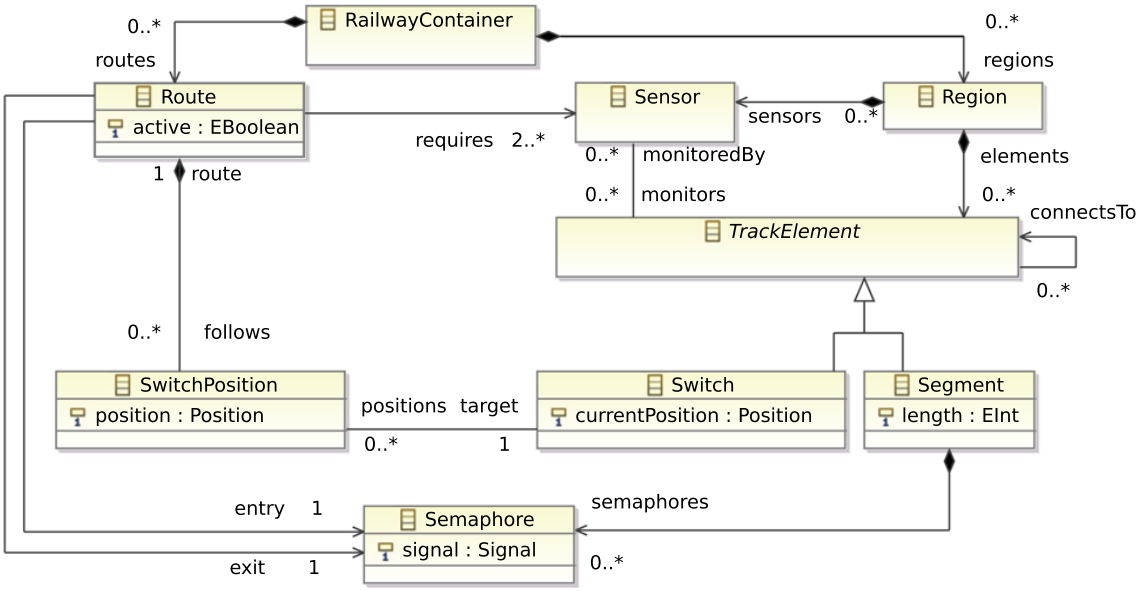
köztük lévő kapcsolatok (élek)[19]. A tulajdonsággráf (property graph) adatmodell kiterjeszti a gráfokat úgy, hogy címkéket/típusokat, illetve tulajdonságokat ad a csúcsoknak és az éleknek. A gráf adatbázisok alkalmasak tulajdonsággráfok tárolására, és az abban lévő adatok lekérdezésére komplex gráf minták használatával. Ilyen rendszerek például a Neo4j [1], OrientDB [14] és a SparkSee [16].

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

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



2.3. ábra. A Train Benchmark metamodellje.(forrás: [29])

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.

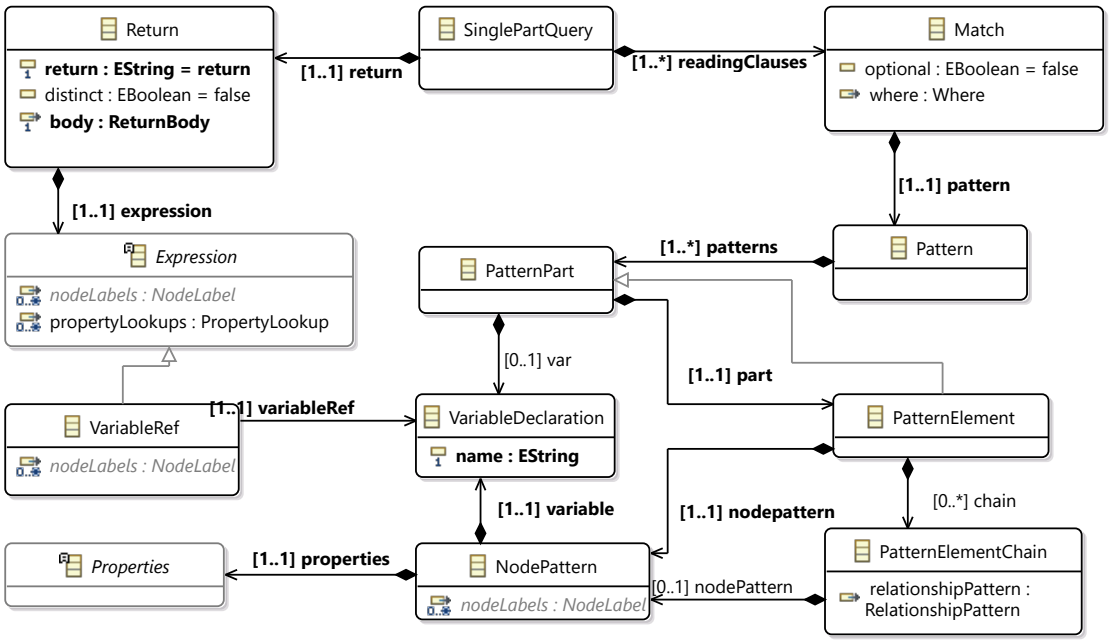
2.3. Modellezés és metamodellezés

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 [25]. Dolgozatomban az Eclipse Modeling Framework (EMF) [15] -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 deklarációt, vagy egy belső részt, ami tartalmaz változó deklarációt. (`VariableDeclaration`).



2.4. ábra. Cypher metamodell

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

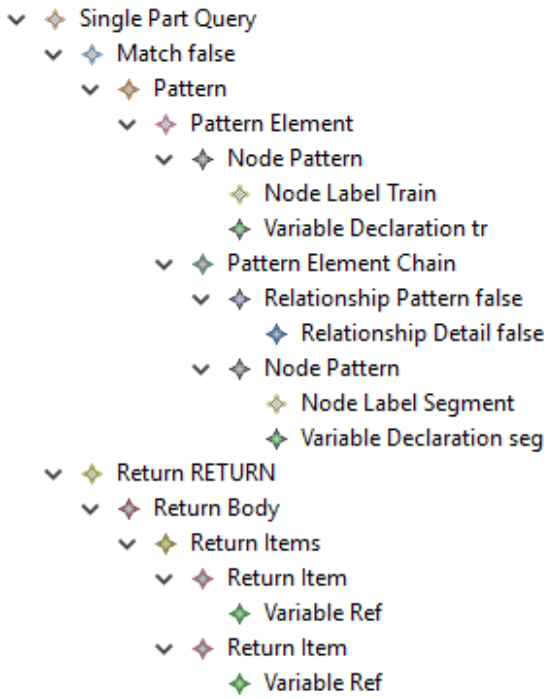
Az Xtext keretrendszer programozási nyelvek, domain-specifikus nyelvek és szöveges editorok fejlesztésére készül. Az Xtext egy erős nyelvtani szabályokkal rendelkező nyelvet használ az egyedi nyelvtanok definiálására. Ezáltal egyszerre biztosít parseolót, linkelőt, helyesírásellenőrzőt szövegkiemelést, 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 [10] keretrendszerben íródott nyelvtanra van szükség, amelyből a metamodell automatikusan elkészült . A dolgozatomban a slizaa [27] nevű Xtext alapú Cypher nyelvtant használtam.

```
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)



2.5. ábra. A példalekérdezés kibontása Xtext-tel

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 egy úgynevezett absztrakt szintaxis gráfra (ASG) lehet bontani. Egy ASG egy olyan gráf, amelyben a csomópontok nyelvtani szbaályokkal vannak címkézve az éleket pedig a részszabályok és aszabályok egymásra hivatkozásai adják. Xtext segítségével ezek az ASG-k objaktum gráfként reprezentálhatóak így programozottan könnyen hozzáférhetőekké válnak. A a 2.5. ábrán egy slizaa nyelvtannal készült ASG-t láthatunk.

2.3.3. VIATRA jólformáltsági kényszerek

Az Eclipse VIATRA keretrendszer [24] 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.

Az alábbi példán egy VIATRA mintát láthatunk. A minta arról szól, hogy egy SinglePartQuery-nek van Match-e. A mintát **pattern** szó után lehet deklarálni, ahol a zárójelek között megadjuk, hogy a mintában milyen elemekkel foglalkozunk, a zárójelek után pedig meghatározzuk a megadott minták egymáshoz való viszonyát.

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

Az alábbi kényszer pedig ellenpéldát keres a fenti mintára, 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. a **neg find** kifejezés pedig azt jelenti hogy a mögötte meghatározott minta ellentéteit keressük.

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

2.4. Gráfgenerálás

Munkám alapját mégis leginkább gráfok generálása képezi. A gráfgenerálás célja, hogy egy adott feladatra szintetizáljon gráfokat A VIATRA Solver [32] 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

Egy adatbázisnak sok része tesztelhető, ezek közül az alábbi három az ami a kutatásom témájába vág:

- **A lekérdezéseket beolvasó editor tesztelése**: Az editor sok szempontból lehet hibás. Funkcionális szempontból az editor által elfogadott lekérdezések halmaza eltérhetnek a specifikációban leírtaktól például egyes a specifikáció alapján felírt nyelvi konstrukciókat az editor nem fogad el. A dologozatban bemutatott módszerrel például felfedeztem, hogy a Neo4j editora csak olyan lekérdezéseket hajlandó beparszolni, amelyekbe, egy referencia csak egyszer szerepel a visszatérési értékben. (`RETURN V1, V1` nem fordul). Ezen kívül az editorok nagy és bonyolult programok, ezért egy-egy fejlesztési lépés után drasztikusan lelassulhatnak.[1][2]
- **A lekérdezések feldolgozása**: Lekérdezés optimalizáló, lekérdezési terv készítő vagy típuskövetkeztető tesztelése. Ilyenkor egy lekérdezés esetén azt vizsgáljuk, hogy a lekérdezésnek megfelelő/optimális/köztes struktúra jön-e létre.
- **Lekérdező motor tesztelése**: Célja az, hogy meghatározzunk a lekérdező motor helyes eredményt ad-e vissza, egy lekérdezésre. Többféle megközelítésben lehet összehasonlítani a helyességet : (1) Más adatbázis implementációkkal (2) A lekérdező rendszer korábbi verziójával (Regressziós tesztelés) [33] (3)Egy összetettebb tesztorákulummal [8]. Munkám elsődleges célja ez.

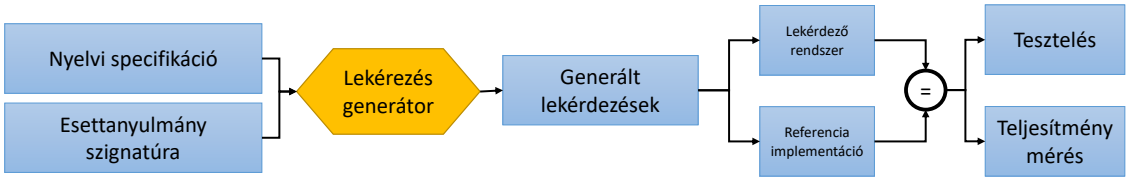
3.1. Funkcionális áttekintés

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.

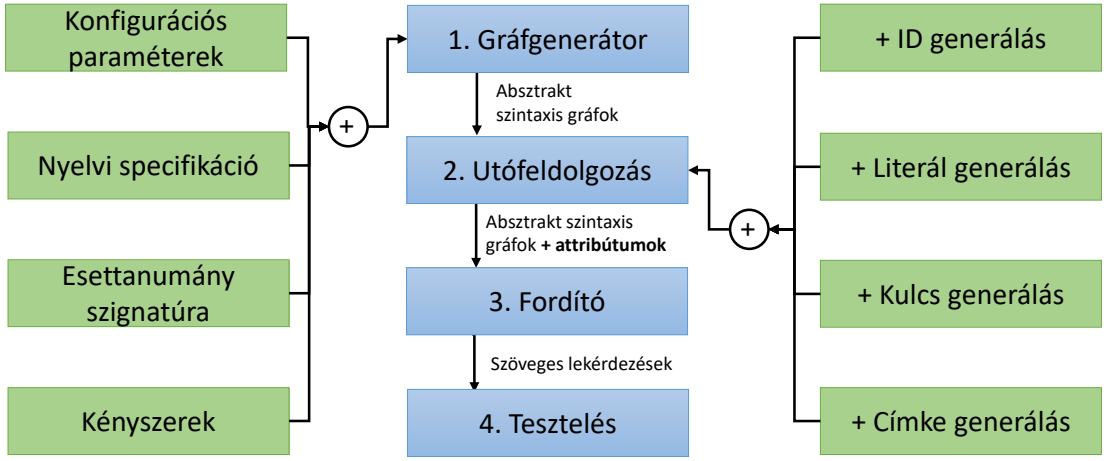
Amint rendelkeznék egy tesztkészletnyi ilyen lekérdezéssel azokat tudnám futtatni azon az adatbázison amelynek szignatúráját esettanulmányként választottam. Ha az ezzel a módszerrel generált lekérdezésekre adott válaszok válaszidejeit összehasonlítjuk több rendszeren tudnánk **teljesítményben tesztelni** azokat. Illetve ha a generált lekérdezések

[1] Editor terribly slow - `https://www.eclipse.org/lists/viatra-dev/msg00501.html`

[2] Type inferrer is slow because of repeated calculations of single compatible parent type - `https://bugs.eclipse.org/bugs/show_bug.cgi?id=534807`



3.1. ábra. Az elképzelés funkcionális áttekintése



3.2. ábra. A lekérdezés generálási folyamat áttekintése

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.

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 [6], mérésekkel bizonyították, hogy a generált modellek szignifikánsan magasabb tesztfedettséget mutattak, mint a manuálisan elkészítettek [26]. 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 [29] által használt szignatúrával van felszerelve. A lekérdezéseket a Neo4j által kifejlesztet Cypher [22] nyelven generálom, a slizaa [27] által készített openCypher nyelvi specifikáció felhasználásával.

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 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. (a) **ID-kel**, (b) **Literálokkal**, (c) **Kulcsokkal** és (d) **Címkékkel**.

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

4. **Tesztelés**: Az így elkészült lekérdezések már használhatóak teljesítménymérésre és tesztelésre.

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 releváns résznyelvét amellyel érdekes lekérdezéseket lehet generálni. Dolgozatomban olyan úgynevezett pozitív mintájú lekérdezések generálására koncentráltam, amelyek vizsgálják a csomópontok illetve a csomópontok közötti kapcsolatok típusát és attribútumait. Ugyanis számos lekérdező nyelvben, mint például Cypher [22] és VIATRA [24] ezek képezik a lekérdezések alapját.

Alább azt kívánom megmutatni, hogy milyen módszerrel lehet le szűkíteni egy Xtext alapú Cypher metamodellt.

1. Példa lekérdezések kiválasztása, amelyekhez hasonló tesztkészletet szeretnénk generálni.
2. Példákból ASG-k készítése.
3. ASG-kból legnagyobb közös részgráf előállítása: ez fogja alkotni a kiindulási részmodellt.
4. ASG-kben használt összes nyelvi konstrukció (típusok, referenciák, literálok) összegyűjtése: ez adja az effektív metamodellt.

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 [10] íródott nyelvtan és a nyelvtan metamodellje közötti konverzió nem tökéletes. Jólformálsági kényszerekre azért van szükség, mert nem minden a metamodellel leírható modell sorosítható Cypher nyelvű lekérdezéssé. 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 az alap struktúráját (referencia hova mutathat) írja le bonyolultabb szabályok (összetett logikai kifejezések) meghatározására alkalmatlan. Az Xtext Cypher nyelvtanban leírt parseolhatósági szabályokat át kell fordítani ASG-n értelmezett struktúrális kényszerekké.

A felső minta meghatározza, hogy hogyan néz ki egy olyan `ReturnItem` aminek van `VariableRef` a visszatérési értékében. Az alsó pedig egy kényszer, aminek felírásával garantálom, hogy csak olyan eredmény áll elő, ahol a minta ez teljesül.

A lenti kényszer azt határozza meg, hogy a lekérdezésekben a `RETURN` szó után kötelező legalább egy változóra referálni.

```
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, _);
}
```

Minden `Pattern`-nek kell hogy legyen legalább egy `PatternElement`-je. Ezt a lenti kényszerben fogalmazom meg. A kényszerre a konzisztensebb és gyorsabb generálás érdekében azért volt szükség. (Egy `Pattern`-be a `.patterns` tulajdonság beállításakor a metamodell alapján más elem is kerülhetne, de a példalekérdezések között egyéb elemre nem volt példa.)

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

A lenti két kényszer megtiltja, hogy egy `PatternElement` rendelkezzen `var` és `part` tulajdonságokkal. Ilyen tulajdonsága csak a `PatternPart` osztálynak lehetnek, de a nyelvtan metamodellje alapján helytelenül a `PatternElement` osztály is megörökli őket.

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

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

Az alábbi kényszerben megfogalmazom, hogy ne legyen `PatternPart` a generált csomópontok között, hanem mindig `PatternElement`-ek jöjjenek létre. Erre azért volt szükség, mert a metamodellben mindkét osztály szerepel, és egyik sem absztrakt. A kényszer által jelentősen csökken a generátor futásideje.

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

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

A `Mapliteral`-ok a következőképpen néznek ki a Cypher nyelven: `name : "seg3"`, egyéb elemek nem kerülhetnek bele. A `Mapliteral` osztály ennek ellenére helytelenül örököl egy `nodelabels` tulajdonságot is. A lenti kényszer biztosítja hogy ne töltse ki azt.

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

Erre a mintára pedig azért van szükség, mert ha nem lennének akkor a következőhöz hasonló értelmetlen lekérdezések generálása válna lehetővé és a generátor nagy méretű modelleknél könnyen esne abba a hibába hogy az extra csúcspontokat így pocsékolja el.

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

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

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

4.1.3. Diverzitás biztosítása

A generált 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 megoldottam a modellek lényegi részeinek generálását, amely biztosította hogy a modellek lényegi különbségekkel rendelkezzenek. 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 a 2.3. 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 változók is amelyeket szisztematikusan elneveztem `V1`...`Vn`-el, illetve literálok amiket pedig `"String1"`...`"Stringn"`-el neveztem el.

Fontos megjegyezni, hogy a generált lekérdezéseknek a konkrét Train Benchmark modelleben nem mindig lesz megoldása. Nézzük meg az alábbi példát:

```
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 nem típusos, így nem készülhet fel a modell alapvető struktúrájára. 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-jeként 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 lekérdezés méretének szempontjából?
- **Kérdés3**: Hogy skálázódik a generálás lekérdezés darabszámának szempontjából?
- **Kérdés4**: Mennyire diverzek a lekérdezések a tesztelés minőségének szempontjából?
- **Kérdés5**: Milyen eredménnyel futtathatóak a lekérdezések egy Neo4j gráfadatbázison?

5.1. Mérési környezet felállítása

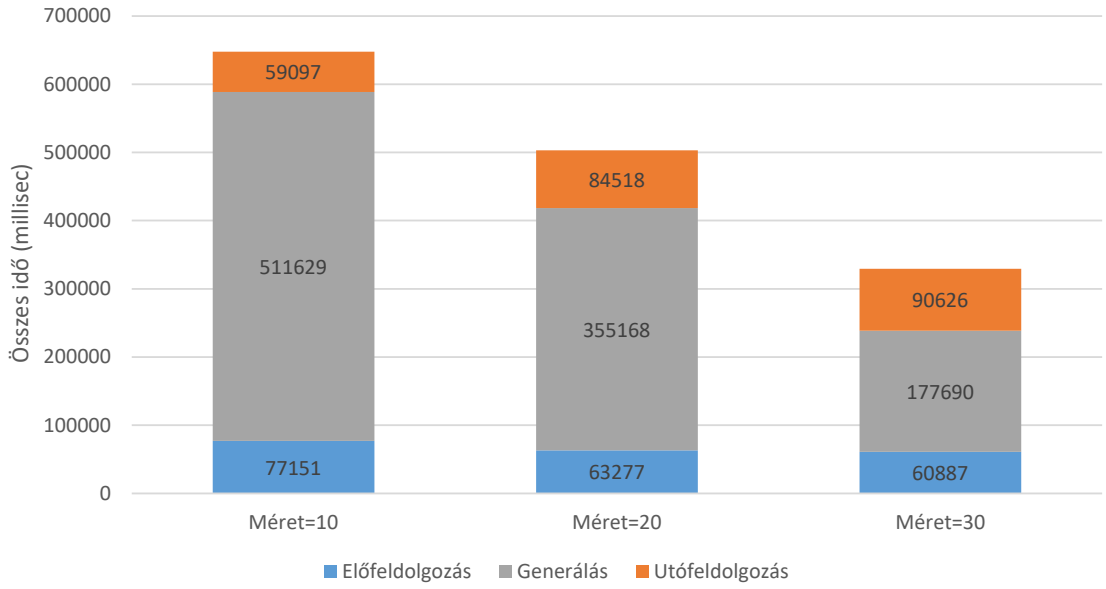
A méréseket eclipse fejlesztői környezetben végeztem. Ahhoz, hogy bemelegítsem a modell generátort memóriakezelés és optimalizálás szempontjából 5 extra futást adtam hozzá minden kiértékelt mérés előtt, melyek futásidejét a mérés során figyelmen kívül hagytam. A mérésekhez a generátor számára 4000 MB 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. Erre az alapra építve két különböző mérési környezetet implementáltam, hogy mind a 4 kérdésre választ tudjak adni. A két környezetben generált gráfokat elkészültük után egy utófeldolgozás fázison mennek keresztül, ahol cypher nyelvű lekérdezésekké fordítom le őket.

5.1.1. M1: Skálázódás modellkészlet méretében.

Az első mérési környezetben 50-50 eltérő lekérdezést generáltam, úgy, hogy 10 20 és 30 elemmel egészítettem ki a kiindulási állapotot. Minden mérést 10-szer megismételtem. (a belemelegedést leszámolva) Az elrendezés célja, hogy a generátor skálázhatóságát vizsgálja modellkészlet méretének szempontjából.

5.1.2. M2: Skálázódás modellkészlet méretében.

A második mérési környezetben 10-10 eltérő lekérdezést generáltam, 5, 10 ,15, ...,50, 100, 150, 200 elemmel egészítettem ki a kiindulási állapotot. Minden mérést 10x megismételtem. (a belemelegedést leszámolva) Az elrendezés célja, hogy a generátor skálázhatóságát vizsgálja modellek méretének szempontjából.



5.1. ábra. A K1 mérés eredményei

5.2. A futásidő összetétele

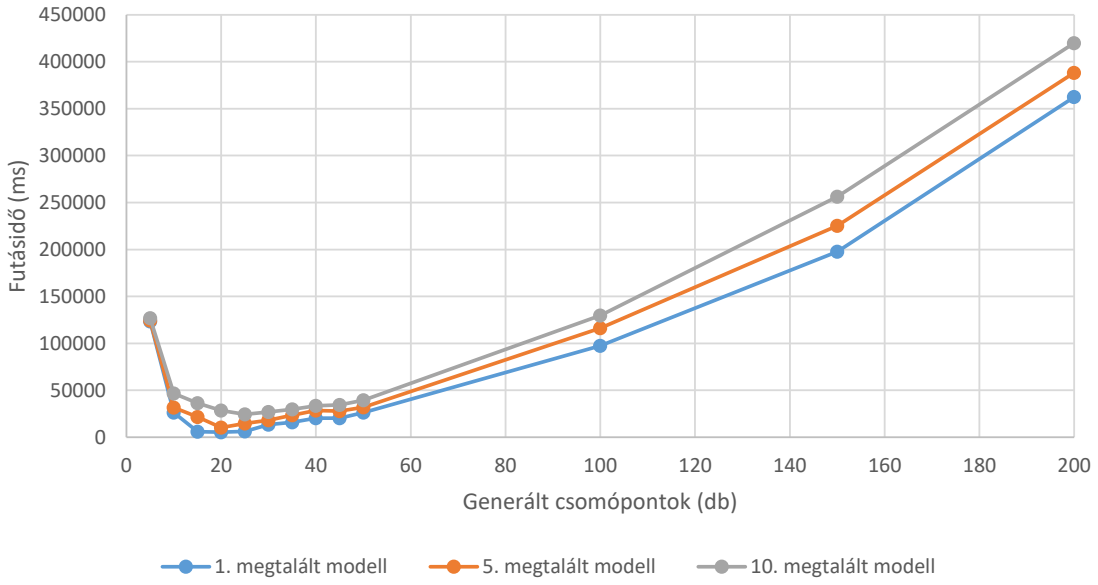
A futásidő összetételének vizsgálatát az M1 mérési környezetben végeztem el, de M2 esetén is hasonló karakterisztikát mutat. A mérés eredményei az 5.1. ábrán láthatóak. Az ábra segítségével azt szeretném bemutatni, hogy a különböző futásidők nagyságrendileg mekkora részét képezik a futásidő egészének. A három szín három értéket reprezentál: az előfeldolgozás idejét (ide tartozik a modellezési nyelv beolvasása, leképezése logikai feltételekké, a logikai probléma megfelelő formára hozása) kék színnel, a generálás idejét szürke (ide a VIATRA Solver futása tartozik) színnel, az utófeldogozás idejét (logikai probléma értelmezése, gráfként való ábrázolása, utófeldolgozás, változók és literálok elnevezése, végül a megoldás gráf ASG -ként való értelmezése és szöveggé alakítása) pedig narancssárga színnel jelöltem. A grafikon vízszintes tengelye a teljes futásidőt mutatja meg. Az első oszlop a 10 a második a 20 a harmadik pedig a 30 hozzáadott csomópontban minimalizált al-környezetben végzett mérések futásidejének összegét ábrázolja.

Az ábrán látható, hogy míg az elő és az utófeldolgozás összes ideje nem mutat jelentős különbséget a három esetben, addig a generálás időtartama szignifikánsan különbözik. Láthatóan a legkisebb méretnél a leghosszabb és a legnagyobbnál a legrövidebb. (Ennek az okát következő szekcióban fejtem ki.)

Végeredményben levonhatom a következtetést, hogy a futásidő leghosszabb részét a generálás ideje teszi ki, míg az előfeldolgozás és az utófeldolgozás arányaiban megegyező időtartamú, kisebb modellek esetében az előfeldolgozás, nagyobbak esetében pedig az utófeldolgozás tart tovább.

5.3. Skálázódás a modellek méretének függvényében

A második kérdésben felállított problémára az M2-es környezetben kerestem a választ. A mérés eredményei az 5.2. ábrán láthatóak. A következtetéseimet az 1., az 5. és a 10.modell megtalálásának időpontjából vontam le. A 10 mérés során számolt futásidők mediánját ábrázoltam. A vízszintes tengelyen a hozzáadott csomópontok minimális elemszáma, míg a függőleges tengelyen az adott modell megtalálásához szükséges futásidő látható.



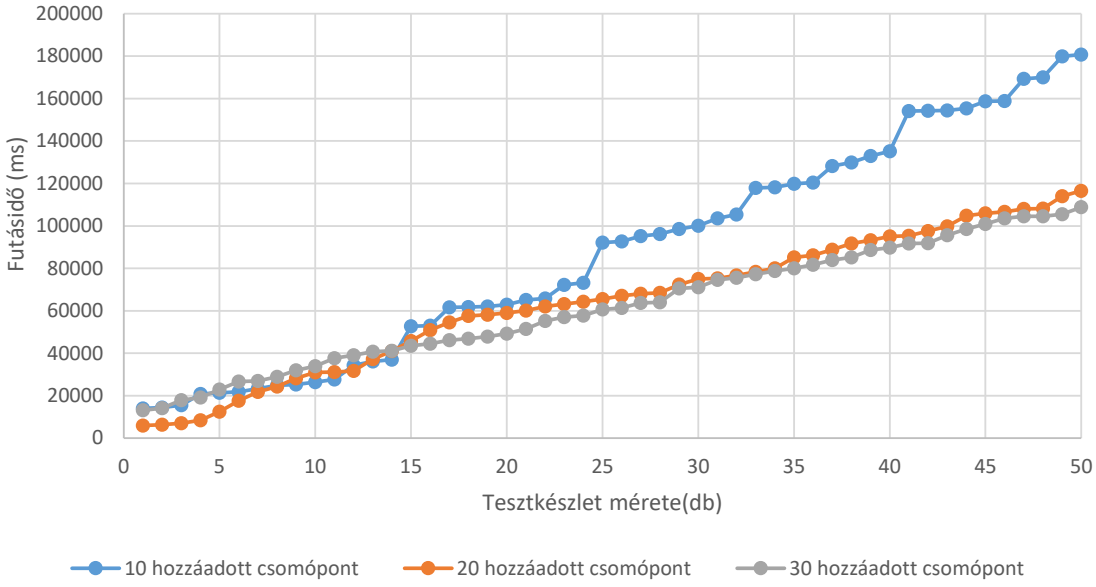
5.2. ábra. A K2 mérés eredményei

Látható, hogy kis minimum darabszámú generált elem hozzáadásával nehezebben boldogult a generátor, mint a közepes elemszámokkal, ám a hatalmas modellek megtalálása is egyre nehezebb feladatnak bizonyult. A kis modelleknél mért lassúság azzal magyarázható, hogy a megadott részmodell elég nagy,(22 elem) emiatt tovább kell keresgélni, hogy 10 hozzáadott elemből ki tudja-e tölteni az összes szükséges helyet,illetve ha nem tudja akkor növelnie kell a hozzáadott csomópontok számát, és újrapróbálkoznia. Nagyobb elemek esetén ez a probléma nem áll fenn, azonban a gráfgenerálás problémája egyenletesen növekszik.

A vizsgált alkalmazás nagyobb modelleknél négyzetes karakterisztikát mutat. A gráfgenerálás egy NP-nehéz probléma, így a négyzetes karakterisztika remek eredmény, de tetszőleges méretű lekérdezések elkészítésére nem biztos hogy alkalmas.

Tehát végeredményben állíthatom, hogy a generátor szépen oldja meg a problémát és 200 generált csomópontra még elfogadható futásidővel működik. (Alább egy 200 csomópontból álló kifejezés látható illusztrációként). A lenti lekérdezés benchmarkolásra és tesztelésre is alkalmas hosszúságú.

```
MATCH ( V1 : Semaphore { signal : "String1" } ) ,
  ( V2 : Semaphore { position : "String2" , length : "String3" } ) ,
  ( V3 : Region { id : "String4" , length : "String5" } ) ,
  ( V4 : Semaphore { length : "String6" , active : "String7" } ) ,
  ( V5 : Route { signal : "String8" , currentPosition : "String9" } )
   - [ V6 : requires { length : "String10" , position : "String11" } ] -
  ( V7 : Region { currentPosition : "String12" , length : "String13" } )
   - [ V8 : entry { signal : "String14" , active : "String15" } ] -
  ( V9 : Route { active : "String16" , currentPosition : "String17" } ) ,
  ( V10 : Segment { length : "String18" , signal : "String19" } ) ,
  ( V11 : Semaphore { id : "String20" , active : "String21" } ) ,
  ( V12 : Route { signal : "String22" } ) ,
  ( V13 : Region { position : "String23" , signal : "String24" } ) ,
  ( V14 : Region { currentPosition : "String25" , length : "String26" } ) ,
  ( V15 : Route { currentPosition : "String27" , currentPosition : "String28" }),
  ( V16 : Segment { currentPosition : "String29" , position : "String30" } ) ,
  ( V17 : Switch { length : "String31" , currentPosition : "String32" } ) ,
```



5.3. ábra. A K2 mérés eredményei

```
  ( V18 : Semaphore { id : "String33" , signal : "String34" } )
RETURN V5 , V10 , V7 , V14 , V18 , V9 , V6 , V16
```

5.4. Skálázódás a modellek darabszámának függvényében

A harmadik kérdésre az M1-es környezetben kerestem a választ. A mérés eredményei az 5.3. ábrán láthatóak. Az 50 darab generált modell megtalálásának ideje nem volt azonos a 10 alkalommal, ezért a 10 érték mediánját kiválasztottam, az ábrán ezeket a medián értékeket jelenítettem meg. A 3 méret mérési eredményeit 3 színnel jelöltem. Az ábrán a függőleges tengelyen a futásidő látható milliszekundumban, a vízszintes tengelyen pedig a létrejött tesztkészlet mérete.

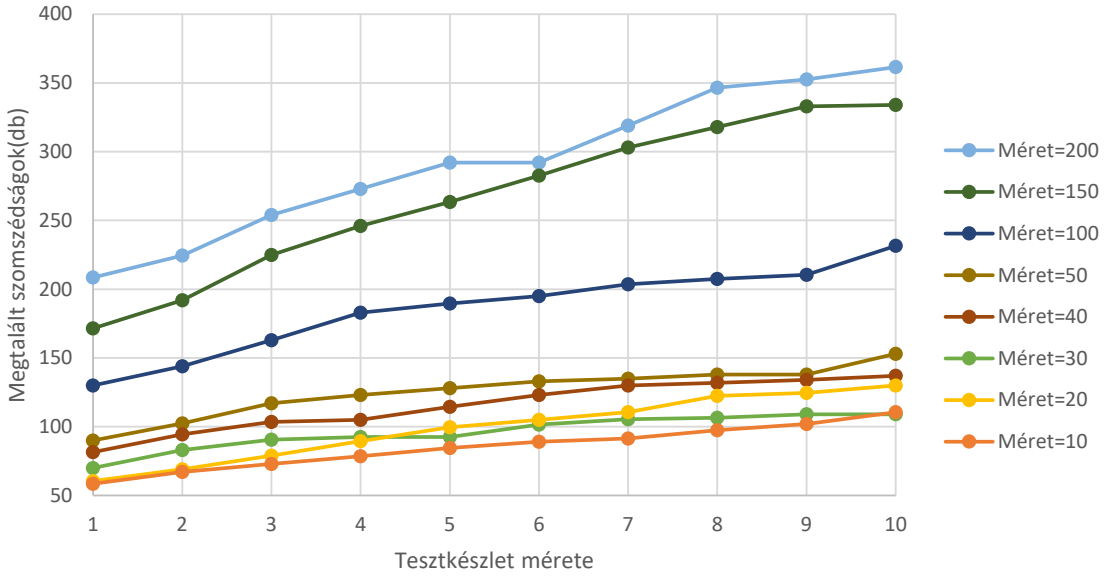
A három mérés eredményei lineárisak különböző meredekséggel, ez azt jelenti, hogy az egyes modellek megtalálása körülbelül ugyanannyi időt vesz idénybe. Ezen az ábrán is látható, hogy a generátor lassabban végzett a nagy modellek megtalálásával mint a kicsikével.

Mivel közel lineáris egyenesek születtek a mérés eredményeként, levonható a következtetés, hogy az egyes modelleket körülbelül azonos időközönként találja meg a generátor, tehát a modellek darabszámának függvényében remekül skálázódik a módszer.

5.5. Diverzitás mérése

Az M1 es környezetben végzett mérések során generált lekérdezések diverzitását is kimértem. Ezt szomszédsági formák segítségével tettem. A mérés eredményei az 5.4. ábrán láthatóak. Az ábra vízszintes tengelyén a tesztkészlet mérete látható, míg a függőleges tengelyen a megtalált szomszédságok darabszáma 3-as távolságban mérve. Az ábrázolt értékek a különböző minimális hozzáadott elemszámok esetén a 10 mérés mediánját mutatják.

Az ábrán látható, hogy a szomszédságok száma körülbelül lineárisan növekszik az különböző esetekben. A nagyobb lekérdezések esetén azért sokkal több a szomszédság mint a kisebbek esetén, mert ezen az ábrán már a cypher nyelvre lefordított, nevekkel kitöltött lekérdezések szomszédságait ábrázoltam.



5.4. ábra. A K2-es mérés diverzitása

Levonható tehát a következtetés, hogy a diverzitás kezdetben folyamatosan közel lineárisan növekszik és nagyobb modellek esetén mindig nagyobb diverzitást érünk el.

Az M2-es környezetben a lekérdezések és nyers kitöltetlen gráfok diverzitását egyaránt kimértem. A mérés eredményei az 5.5. ábrán láthatóak. Az ábra vízszintes tengelyén a tesztkészlet mérete látható, a függőleges tengelyen pedig a megtalált szomszédságok darabszáma. Az ábrázolt értékek a 10 elvégzett mérés során kapott értékek mediánjai.

Az ábrán látható, hogy a kitöltetlen modellek sokkal kisebb diverzitást mutatnak mint a kitöltöttek. Illetve az is látható, hogy a kevés modellnél még lineárissal közelíthető növekedés sok modellnél logaritmikussá alakul.

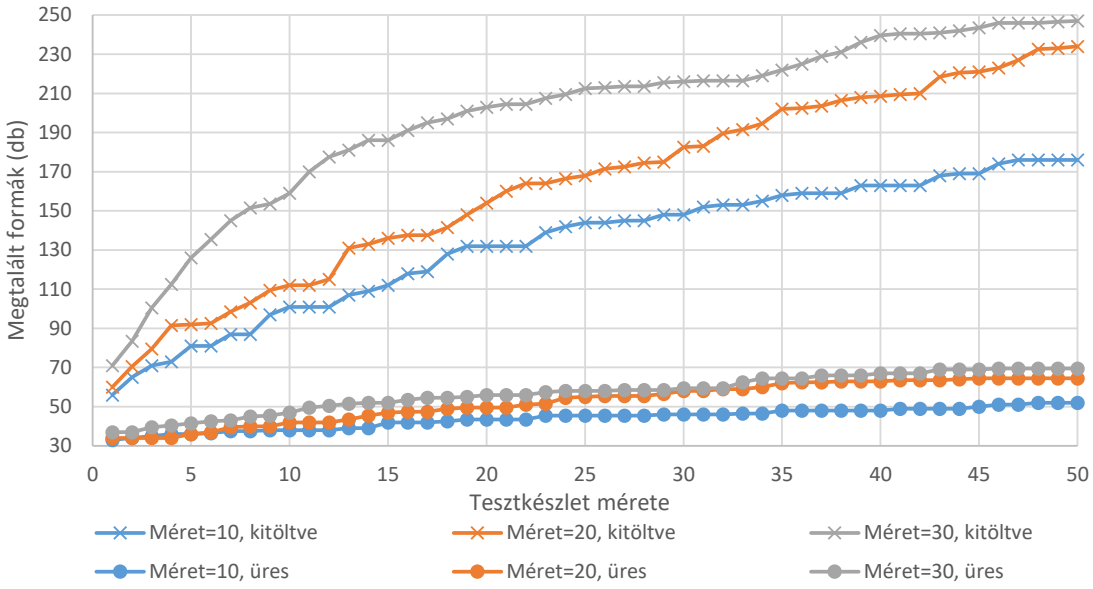
Az általam bevitt diverzitás növelését segítő módszerek tehát hatékonyak,illetve a generálás során a diverzitás folyamatosan növekszik.

5.6. Lekérdezések futásidejének mérése Neo4j adatbázison

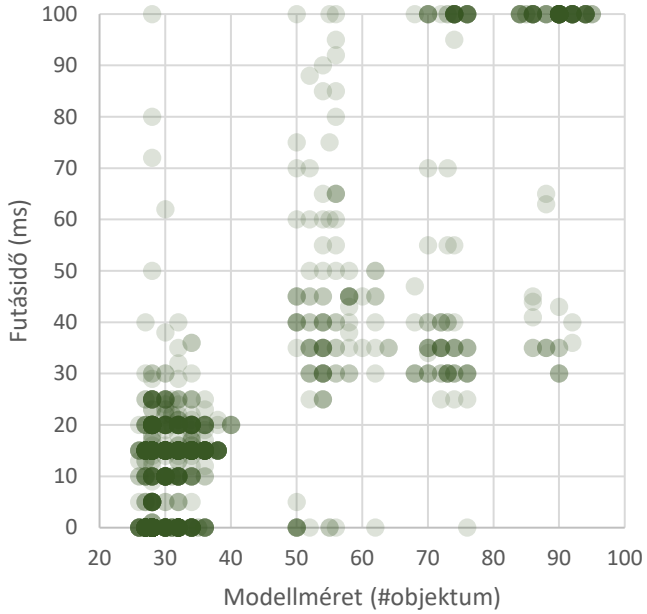
Az M2es környezetben generált lekérdezéseket egy a Train Benchmark által összeállított adatbázison futtattam, amelyben 2024 csomópont és ezek között 5878 kapcsolat volt. Az első futás alkalmával a legtöbb lekérdezés 0 ms-os eredménnyel nem futott le, ez azért volt, mert a Neo4j az attribútumok alapján indexel, és az általam generált lekérdezések ebből a szempontból nem voltak helyesek, ezért a lekérdezéseket megszűrtem az utófeldolgozás során, és attribútum mentesen is lefuttattam őket az adatbázison. Ennek a mérésnek az eredményei az 5.6. ábrán láthatóak.

Az ábrán a pontok az egyes modellekhez tartozó futásidőket jelenítik meg. A vízszintes tengelyen a modellek mérete látható, az alapján számolva hogy hány csomópontból áll az ASG-jük, a függőleges tengelyen pedig a hozzájuk tartozó futásidő. A sötétebb pontok azt jelentik, hogy ott sok egyforma érték született, a halványabbaknál kevesebb. A futásidőt 100-ban maximalizáltam, azok a lekérdezések amelynek a futásidejét 100-al jelenítem meg azok 5 másodpercen keresztül nem futottak le, de nem jött rá a rendszer, hogy nem jók.

Az ábrán látható, hogy a kisebb modellek kisebb, a közepesek közepes, a nagyok pedig nagyon nagy futásidőt mutatnak, az is látható, hogy míg a kisebb modelleknél volt kevés olyan lekérdezés ami nagyon lassan futott le, addig a nagyobbaknál nem volt olyan ami nagyon gyorsan futott volna le. A futásidő és a modell méret között kiszámoltam a



5.5. ábra. K3-as mérés diverzitása



5.6. ábra. Lekérdezéseim futásidejének eredménye

korrelációs együttható értékét, 0.85 -öt kaptam. Ez azt jelenti, hogy a modellméret és a futásidő között nagy korreláció van jelen.

Következtetésként levonhatjuk,

5.7. Lehetséges mérési hibák

- Egyetlen esettanulmányon futtattam a méréseket, de ez egy reprezentatív, korszerű, aktívan fejlesztett teljesítmény benchmark, amelyet már több más esettanulmányban is alkalmaztak.[13, 5]
- Csak pozitív mintájú lekérdezések generálásával foglalkoztam. Mivel ezek adják az összes lekérdezés alapját a tőlem függetlenül fejlesztett nyelvtanban, jó reprezentatív esetnek számítanak.
- A mérések során mindent csak 10-szer futtattam, de a futásidők és a diverzitások nem mutattak nagy eltéréseket egymástól. A mérési zajt medián számítással mértem ki. A bemelegedés hatását pedig bemelegítő mérések hozzáadásával küszöböltem ki az éles mérések előtt, megvárva a futásidő stabilizálódását.

6. fejezet

Kapcsolódó munkák

Az alábbiakban összefoglalom, a munkámhoz kapcsolódó szakirodalmat. A modellgenerálás fejezet ennek [26] a cikknek a felépítését követi.

6.1. Modell és gráfgenerálás

A diverz modell generálás kulcsfontosságú szerepet játszik modell transzformációk, kód generátorok és komplett fejlesztőrendszerek tesztelése során. A mutáció alapú megközelítések [2], [9], létező modelleken hajtanak végre véletlenszerű változtatásokat mutációs szabályok alkalmazásával. Más automatizált technikák [4], [11], olyan modelleket generálnak amelyek csak a metamodellhez alkalmazkodnak. Míg ez az utóbbi megoldás jól skálázódik nagyobb modellekre a mutációs módszerrel generált modelleknél nincs arra garancia hogy jólformáltak lesznek.

A modellgeneráló technikák egy nagyobb halmaza bizonyos ígéreteket ad a teszt hatékonyságára. Az ilyen white-box alapú megközelítések [2], [3], implementáción és transzformáción alapulnak, és általában back-end logikai megoldókat használnak, amelyek nem jól skálázhatóak gráfmodellekre.

A black-box megközelítések [7], [12], csak a specifikációját használják fel a nyelvnek, vagy a transzformációnak, így alapvetően kényszereken illetve részmodelleken alapulnak. Ezekben a megközelítésekben közös, hogy néha egyszerű modelleket generálnak, amelyeknek növelhető ugyan diverzitása szimmertia-törő predikátumokkal, de nagyobb modellekre nem skálázódnak. Sőt, a modellek effektív diverzitása is megkérdőjelezhető, mert a jelenlegi modell- transzformációkat tesztelő módszerekben sokkal enyhébb vizsgálatoknak kell csak megfelelni, mint a különböző szoftverek tesztelése során.

6.2. Adatbázis tesztelés

Az adatbázisok tesztelése nehéz, aktívan kutatott feladat, több hasonló megközelítés is található a szakirodalomban.

A miénkhez leghasonlóbb megközelítés a [18] cikkben leírt ADUSA keretrendszer, ami szintén SAT alapú megoldókkal dolgozik, ők Alloy-t [17] , [30] használnak. Az ő módszerük a miénkkel ellentétben SQL specifikus és nyelv alapján úgy tűnik hogy csak fix méretű modellek generálására alkalmas. A Viatra Solver is támoatja az Alloy-t mint mögöttes megoldót cypher lekérdezések generálására viszon azt tapasztaltam, hogy nem skálázódik eléggé. Másrészt a korábbi tapasztalataink alapján [26] gyenge minőségű tesztkészletet állít elő diverzitás szempontjából.

Az [21] cikk páronkénti fedettség tesztelési módszert javasol adatbázisok tesztelésére. Ez azt jelenti, hogy első lépésben részekre bont egy lekérdezést, majd a másodikban min-

den részletre felsorol lehetséges részkifejezéseket, majd ezekből a részkifejezésekből készít helyes lekérdezéseket úgy, hogy minden lehetséges részlet pár legalább egyszer szerepeljen. (Anélkül, hogy explicit az összes kombinációt elő kelljen állítani.) Ehhez képest a mi megközelítésünk során nem szükséges explicit felsorolni lekérdezés részleteket, hanem ezeket automatikusan állítjuk elő. Továbbá az általun biztosított fedettség hasonló a páronkénti fedettséghez, hiszen a generálás során az összes különböző szomszédságú gráfcsomópont felfedezését biztosítjuk.

Az [33] cikkben egy felhő alapú adatbázisban végeznek regressziós tesztelést, a miénkhez hasonló felépítés segítségével, csak itt a teszt lekérdezések halmazát a felhasználók által korábban írt és futtatott lekérdezésekből válogatják össze így próbálva biztosítani a diverzitást. Ezzel szemben mi automatikusan generálunk lekérdezéseket, ami ezáltal remekül egészíti ki ezt a megközelítést olyan esetben amikor a lekérdezések nem hozzáférhetőek.

A [31] cikkben alkalmazott módszer során relációs adatbázisok teszteléséhez generálnak SQL lekérdezéseket mutációs módszerrel. Ennek a megközelítésnek több hátránya is van amit az én megközelítésem kiküszöböl. Egyrészt szükség van egy nagyobb meglévő tesztkészletre, másrészt a mutánsok hasonlítanak az eredeti teszt készletre ezért rossz minőségű tesztkészletet alkotnak diverzitás szempontjából.

A [28] cikkben SQL specifikus struktúrális metrikákat javasolnak a nagyobb tesztfedettség eléréséhez. Mi is ehhez hasonlóakat használunk általánosan.

A [8] cikkben a módszerünkkel ellentétben white-box alapú adatbázis tesztelési módszert javasolnak, mégpedig úgy, hogy a lekérdezéseket végrehajtható forráskóddá alakítják és így teszt orákulumot képeznek.

6.3. Adatbázis benchmarkolás

Sok adatbázis Benchmark létezik, egy válogatást a [29] cikkből emeltem ki az alábbi táblázatba. A táblázat alsó sorában az látható, hogy a benchmarkolásra hány lekérdezést használtak fel. Ezzel szemben munkám során én megközelítőleg 3000 lekérdezést generáltam. Az eddigi benchmarkoklási technikák hiányosságnak a lekérdezések mennyisége tűnik. Ezáltal a megközelítésem alkalmazásával forradalmasítani lehetne az adatbázisok teljesítmény tesztelését.

LUBM	Barton	SP2Bench	BSBM	DBpedia	LDBC SNB	Train Benchmark
14	7	12	12	25	14	6

Tudomásom szerint nincs olyan adatbázis benchmark ami szintetikus lekérdezéseket futtatna.

7. fejezet

Összefoglaló és jövőbeli munkák

A dolgozatomban sikerrel megvalósítottam egy többlépéses gráflekérdezés generálási folyamatot, amely képes volt Neo4j adatbázis által beolvasható és feldolgozható lekérdezéseket előállítására. A generátor paraméterezhető a vizsgálni kívánt adatbázis tartalmával (címkekészlet), valamint a lekérdezések mennyiségével és bonyolultságával. Ezen felül biztosítja lekérdezések diverzitását, így minden kiadott lekérdezés különbözik a többitől. Az általam készített keretrendszert teljesítmény és diverzitás szempontjából is kiértékeltem egy ipari esettanulmányon, valamint korrelációt is találtunk a lekérdezések bonyolultsága és futásideje között. Továbbá kiemelendő, hogy dolgozatomban számos technológiát alkalmaztam:

- Gráfadatbázisok területén: Cypher gráfadatbázisnyelv, Neo4j adatbázis, Train Benchmark esettanulmány.
- Modellezés területén: EMF modellező keretrendszer, Xtext technológia lekérdezések beolvasáshoz és sorosításhoz (konkrétan a slizaa nyelvtant használva), VIATRA gráfmintaillesztő rendszer jólformáltsági kényszerek meghatározására, és Xtend-et modellek utófeldolgozására.
- Matematikai eszközök:VIATRA Solver gráfgeneráláshoz, illetve a prototipizáláshoz Alloy gráfgenerátor Sat4j mögöttes SAT megoldóval (amely skálázódási okok miatt nem volt alkalmas komolyabb mérések elvégzéséhez). Elméleti eredményként elmondható, hogy sikerrel alkalmaztam egy fejlett gráfgenerálási algoritmust

Elméleti eredményként elmondható, hogy Cypher nyelv slizaa nyelvtanának nyelvtani szabályait absztrakt szintaxis gráfon értelmezhető gráfmintákként formalizáltam VIATRA nyelven. Ezáltal a nyelv feldolgozhatóvá vált logikai következtetőkkel, amit sikerrel alkalmaztam lekérdezések szintetizálásához. Ezen felül a Cypher nyelv szimmetrikus megfogalmazásait modell-szimmetriaként fogalmaztam meg, így javítva a lekérdezések diverzitását.

Sikerként mondható el, hogy már sikerült is találni szemantikus eltérést a tanszékemen fejlesztett InGraph [20] és a referenciaimplementáció Neo4j rendszerekben. Ezen kívül felkerestük a Neo4j fejlesztőit is, akiknek felkeltette az érdeklődését a fejlesztett eszköz.

Jövőbeli munkaként az általam készített lekérdezés generátor kiegészíthető lehet gráfadatbázis-tartalmak előállításával is. Az így előálló rendszer akár egy komplett környezetet biztosíthatna gráfadatbázisok tesztelésére és teljesítménymérésére, előállítva a teljes bemenetet.

Irodalomjegyzék

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

[2] Vincent Aranega – Jean-Marie Mottu – Anne Etien – Thomas Degueule – Benoit Baudry – Jean-Luc Dekeyser: Towards an automation of the mutation analysis dedicated to model transformation. *Software Testing, Verification and Reliability*, 25. évf. (2015) 5-7. sz., 653–683. p.

[3] Behzad Bordbar – Kyriakos Anastasakis: Uml2alloy: A tool for lightweight modelling of discrete event systems. In *IADIS AC* (konferenciaanyag). 2005, 209–216. p.

[4] Erwan Brottier – Franck Fleurey – Jim Steel – Benoit Baudry – Yves Le Traon: Metamodel-based test generation for model transformations: an algorithm and a tool. In *Software Reliability Engineering, 2006. ISSRE'06. 17th International Symposium on* (konferenciaanyag). 2006, IEEE, 85–94. p.

[5] Márton Búr – Gábor Szilágyi – András Vörös – Dániel Varró: Distributed graph queries for runtime monitoring of cyber-physical systems. In *International Conference on Fundamental Approaches to Software Engineering* (konferenciaanyag). 2018, Springer, 111–128. p.

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

[7] Fabian Büttner – Marina Egea – Jordi Cabot – Martin Gogolla: Verification of atl transformations using transformation models and model finders. In *International Conference on Formal Engineering Methods* (konferenciaanyag). 2012, Springer, 198–213. p.

[8] Man-yee Chan – Shing-Chi Cheung: Testing database applications with SQL semantics. In *CODAS* (konferenciaanyag). 1999, 364–376. p.

[9] Andrea Darabos – András Pataricza – Dániel Varró: Towards testing the implementation of graph transformations. *Electronic Notes in Theoretical Computer Science*, 211. évf. (2008), 75–85. p.

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

[11] Karsten Ehrig – Jochen Malte Küster – Gabriele Taentzer: Generating instance models from meta models. *Software & Systems Modeling*, 8. évf. (2009) 4. sz., 479–500. p.

[12] Franck Fleurey – Benoit Baudry – Pierre-Alain Muller – Yves Le Traon: Towards dependable model transformations: Qualifying input test data. *Journal of Software and Systems Modeling (SoSyM)*, 2007.

[13] Antonio Garcia-Dominguez – Konstantinos Barmpis – Dimitrios S Kolovos – Ran Wei – Richard F Paige: Stress-testing remote model querying apis for relational and graph-based stores. *Software & Systems Modeling*, 2017., 1–29. p.

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

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

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

[17] Daniel Jackson: Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11. évf. (2002) 2. sz., 256–290. p.

[18] Shadi Abdul Khalek – Sarfraz Khurshid: Automated SQL query generation for systematic testing of database engines. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010* (konferenciaanyag). 2010, 329–332. p.
URL `http://doi.acm.org/10.1145/1858996.1859063`.

[19] 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.

[20] József Marton – Gábor Szárnyas – Márton Búr: Model-driven engineering of an opencypher engine: Using graph queries to compile graph queries. In *SDL Forum*, Lecture Notes in Computer Science konferenciasorozat, 10567. köt. 2017, Springer, 80–98. p.

[21] Yuper Lay Myint – Hironori Washizaki – Yoshiaki Fukazawa – Hideyuki Kanuka – Hiroki Ohbayashi: Test case reduction based on the join condition in pairwise coverage-based database testing. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (konferenciaanyag). 2018, IEEE, 239–243. p.

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

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

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

[25] 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.

[26] 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.

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

[28] María José Suárez-Cabal – Javier Tuya: Using an sql coverage measurement for testing database applications. *SIGSOFT Softw. Eng. Notes*, 29. évf. (2004. október) 6. sz., 253–262. p. ISSN 0163-5948.
URL `http://doi.acm.org/10.1145/1041685.1029929`. 10 p.

[29] 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.

[30] Emina Torlak – Daniel Jackson: Kodkod: A relational model finder. In *Tools and Algorithms for the Construction and Analysis of Systems*. 2007, Springer, 632–647. p.

[31] Javier Tuya – Ma Jose Suarez-Cabal – Claudio De La Riva: Sqlmutation: A tool to generate mutants of sql database queries. In *Mutation Analysis, 2006. Second Workshop on* (konferenciaanyag). 2006, IEEE, 1–1. p.

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

[33] Jiaqi Yan – Qiuye Jin – Shrainik Jain – Stratis D Viglas – Allison Lee: Snowtrail: Testing with production queries on a cloud database. In *Proceedings of the Workshop on Testing Database Systems* (konferenciaanyag). 2018, ACM, 4. p.