Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Evaluation of openCypher graph queries with a local search-based algorithm

Master's Thesis

Author
Dávid Szakállas

Advisor
Gábor Szárnyas
Márton Búr

Monday 18th December, 2017

Contents

Kivonat **i**

Abstract **ii**

1 Introduction **1**

1.1 Motivation 1

1.2 Structure of the thesis 2

2 Preliminaries **3**

2.1 Knowledge graph of a railway system 3

2.2 Train Benchmark: a railway metamodel & benchmark suite 5

2.2.1 Metamodel 6

2.2.2 Queries 6

2.3 Property graph data model 6

2.3.1 Definition 6

2.3.2 Our running example as a property graph 8

2.4 Cypher: a query language for property graphs 8

2.4.1 Syntax 8

2.4.2 openCypher 10

2.4.3 Train Benchmark queries 10

2.5 Relational Graph Algebra 11

3 A dynamic programming based search plan generation algorithm **14**

3.1 Planning 14

3.2 Search plan evaluation 18

3.3 Strengths, shortcomings & adaptation barriers 19

3.3.1 Strength: Polynomial complexity for planning 19

3.3.2 Shortcomings: Missing initial case & special handling of check operations 19

3.3.3 Limitation: Applicability for property graphs 21

4 Overview of the approach **22**
4.1 General Concepts . . . 22
4.2 Search planning . . . 24
4.2.1 The `compile_search_plan` procedure . . . 25
4.2.2 The `step` procedure . . . 26
4.2.3 The `insert_cell` procedure . . . 27
4.3 Extending the search planner with patterns . . . 28

5 Adapting the search engine to ingraph **31**
5.1 The ingraph query engine . . . 31
5.2 Clojure . . . 32
5.3 Mapping relational algebra operators to constraints . . . 32
5.4 Platform operations . . . 34
5.5 Evaluation . . . 35
5.6 Conclusion . . . 36

6 Related work **37**
6.1 Queries on property graphs . . . 37
6.1.1 Query languages . . . 37
6.1.2 Query engines . . . 39
6.2 Search-based graph queries . . . 39
6.2.1 Database technologies . . . 39
6.2.2 Model-driven technologies . . . 40
6.3 Train Benchmark in the Transformation Tool Contest . . . 40

7 Summary and future work **41**

Acknowledgements **43**

Bibliography **48**

Appendix **49**
A.1 Generalized cost function . . . 49
A.2 ActiveRoute evaluation with costs . . . 49
A.3 Source code of the traversal . . . 51

HALLGATÓI NYILATKOZAT

Alulírott *Szakállas Dávid*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2017. december 18.

Szakállas Dávid
hallgató

Kivonat

Az 1970-es évektől az elmúlt évtizedig az adatbázis-kezelés szinte kizárólag relációs technológiák alkalmazását jelentette. Az elmúlt évtizedben azonban több tucat olyan rendszer jelent meg és terjedt el, amelyek nemrelációs adatmodellt használnak. Ezek az ún. NoSQL rendszerek egyik csoportja a gráfadatbázis-kezelők, amelyek a tulajdonsággráf (property graph) adatmodellt támogatják. Napjaink egyik legnépszerűbb gráfadatbázis-kezelő rendszere, a Neo4j, a Cypher lekérdezőnyelvet hozta létre arra, hogy a felhasználók és fejlesztők gráfmintákat definiálhassanak. Az openCypher kezdeményezés 2015. végén indult azzal a küldetéssel, hogy – a relációs adatbáziskezelés világának SQL nyelvéhez hasonlóan – egy szabványos nyelvet definiáljon gráflekérdezések megvalósítására. Ennek célja, hogy a felhasználók szabványos lekérdezéseket fogalmazhassanak meg, amelyek különböző gráfadatbázis rendszereken kiértékelhetők, így csökkentve azt a kockázatot, hogy csak egy adott adatbázis rendszer képes a lekérdezéseik kiértékelésére (vendor lock-in).

A tanszéken fejlesztett ingraph rendszer célja, hogy openCypher nyelven definiált gráflekérdezések hatékony kiértékelését tegye lehetővé. Az ingraph rendszerben jelenleg már működik egy inkrementális lekérdezőmotor, amely képes arra, hogy az előre megadott, folyamatosan kiértékelt lekérdezések eredményhalmazát hatékonyan karbantartasa. Ez a megközelítés azonban rendkívül pazarlóan bánik a memóriával, így nem hatékony komplex, egyszer lefuttatott lekérdezések kiértékelésére. Erre a problémára a lokális keresést alkalmazó algoritmusok használata célravezetőbbnek bizonyulhat. Munkám során egy meglévő ilyen, a keresést kényszerkielégítési problémára visszavezető algoritmust módosítottam és illesztettem az ingraph rendszerhez. A motor az openCypher nyelvben fellelhető minták nagy részét támogatja, az egyszerű csomópont-, útvonal- és típuskritériumokon túl képes negatív kondíciók kiértékelésére is, valamint kiterjeszthető további kényszerekkel, mintákkal. A kiértékelés hatékonyságát és skálázhatóságát teljesítménymérésekkel igazoltam.

Abstract

Since the 1970s database management had been almost exclusively based on relational technologies. However, the last decade saw the birth and rise of dozens of systems facilitating nonrelational models. One particular branch of these so-called NoSQL systems are graph databases, which usually employ the distinctive "property graph" model. Neo4j, which is arguably the most popular such graph database provides its own language called Cypher. Following the heritage of SQL, the openCypher initiative - established in 2015 - set on a mission to standardize the language, with the incentive to reach general adoption and compatibility between implementations.

The ingraph system, developed at the Department of Measurement and Information Systems, features incremental query evaluation. This approach supports effective execution of repeated queries by caching interim results, however it is not suitable for running complex one-off queries. For these, algorithms using graph exploration may prove far more suitable. In my work, I extend and adapt a dynamic programming based polynomial time algorithm for ingraph. My engine is already able to handle a wide range of patterns defined on openCypher, such as vertex, edge and type criteria, negative patterns and can be extended to support more constraints and patterns. I support the performance and scalability properties of the system with benchmarks.

Chapter 1

Introduction

Once regarded as an obscure notion of mathemathics and scientific research, graphs are rapidly gaining ground in mainstream software development. The abstraction that left a major footprint on many natural and applied sciences, such as biology, physics, electrical engineering, computer science, computer networking, software engineering and social sciences is expected by many to be the foundational data model of a growing range of non-scientific applications in the coming years. In fact, graphs can be leveraged in at least four current disruptive trends [30] in software technology. In the field of financial automation, graph-based transaction histories can revolutionize micropayment transaction systems [42], knowledge graphs might finally bring about semantic web search [**?**], and the Internet of Things [**?**] along with mobile networks [26] are in essence highly dynamic and volatile graphs.

1.1 Motivation

Naturally, the broad range of specific application requirements juxtaposed with the abstract mathematical definition of graphs sprouted solutions tailored to the given domain and seldom applicable anywhere else. Aiming at closing this gap, it is no surprise the software industry is pushing forward to deliver general graph processing platforms. This "revolution" is similar to that of relational database systems (RDBMS) and the SQL language for OLTP[1] in the 1970s, or Hadoop and MapReduce for big data ten years ago, successfully pioneering direly needed workflows and eventually becoming industry standards. Notwithstanding the potential the current graph processing initiatives have, admittedly they are not expected to be capable of providing a silver bullet for all graph-related problems in the foreseeable future. If we take a closer look on the spectrum of these projects, we find that they are almost exclusively centered around so called knowledge graphs[2], a model for encoding domain information in graph structures. Then, the developers of these platforms are focusing on either

(a) creating a distributed system for running knowledge graph specific workloads or

(b) porting such processes to existing architecture.

[1] Online transacion processing

[2] also called semantic graphs in some jargon

Neo4j, Inc., falls into the first category, and develops its own graph database technology[3]. A building block of this database is "Cypher", a SQL-like query language for knowledge graphs. A standardized subset of this language is developed under the name openCypher as an open specification with the goal of creating the de-facto query language for knowledge graphs. openCypher already reached some success throughout its two-years-old infancy by being adopted by a handful of commercial systems, most notably SAP HANA Graph and AgensGraph.

The subject of this thesis is to create a supplementary execution engine for ingraph, an openCypher backend developed by the Fault Tolerant Systems Research Group (FTSRG) of Budapest University of Technology and Economics, with the mission statement of "providing a horizontally scalable graph query engine, which is able to perform complex graph queries with 100M+ elements". Although the main focus of the project is incremental evaluation[4] (hence the name *in*graph), the requirement to run conventional one-shot queries arose recently. As ingraph is a completely in-memory database we felt that a graph exploration based approach would overperform a relational engine.

1.2 Structure of the thesis

The document follows a conventional structure and introduces the most important preliminary subjects through an example. This is done in Chapter 2, opening with a particular knowledge graph domain; followed by property graphs; Cypher, and a relational algebra extension for it. We interrupt the string of property-graph related sections with the introduction of a search-driven model querying algorithm developed by Gergely Varró et al. in Chapter 3.

Next, we present the main contributions of this thesis. In Chapter 4, we discuss our extensions to the search-driven algorithm to lay down the theoretical background for our engine. Building on these extensions, in Chapter 5, we discuss the adaptation of the search plan generation algorithm for property graphs, and present SRE, a search engine for the *ingraph* engine. We also show preliminary performance experiments of the engine.

Finally, we discuss existing research and implementations on related topics in Chapter 6 and summarize the results in Chapter 7.

[3]The company originally named Neo Technology has been rebranded after its sole product in 2017 as Neo4j, Inc.

[4]this is closely related to to event stream processing as here a continuous stream of changes are run through a constant – or in other terms, a standing – query, which generates a further stream as its output.

Chapter 2

Preliminaries

2.1 Knowledge graph of a railway system

Knowledge graphs (abbrev. KG) have been in the focus of research since 2012 resulting in a wide variety of published descriptions and definitions, some which are contradicting [13]. Here we use the most widely accepted one given by Paulheim [40]: "A knowledge graph *(i)* mainly describes real world [*sic*] entities and their interrelations, organized in a graph, *(ii)* defines possible classes and relations of entities in a schema, *(iii)* allows for potentially interrelating arbitrary entities with each other and *(iv)* covers various topical domains."[1]

It is useful to present an example with which the preliminary concepts and the body of the work can be illustrated throughout the rest of the document. A data set is also essential for testing and benchmarking the implementation itself. We derive both the illustrative example and the benchmark data sets from the *Train Benchmark* (abbrev. TB), a benchmark suite for continuous model queries on *KG*s [53]. Moreover, as Cypher is a large and complex language, we must cherry-pick a limited set of features in the scope of this thesis project. We chose our goals in terms of queries provided by Train Benchmark, i.e., we demarcated the set of language features for inclusion that are simple, but are able to express some TB queries.

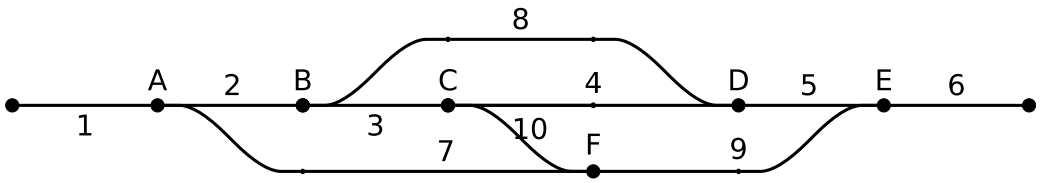
As the name implies, Train Benchmark uses a domain-specific model of a safety-critical system: railways. Our toy example features *track elements*, such as

1. Segments, interleaved with
2. Switches, together constituting a physical topology of railway tracks;
3. Routes, logical paths that require
4. Sensors for safe operation.

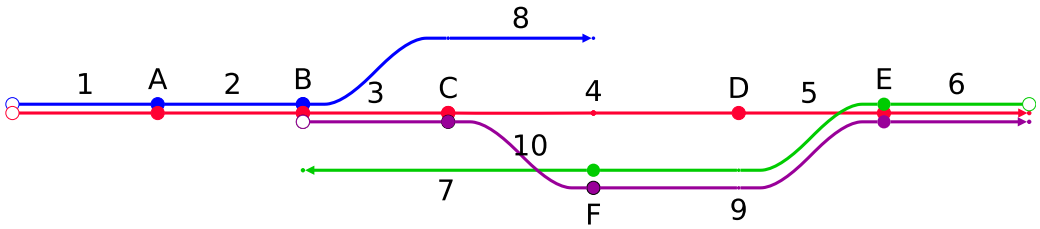
A Route can be understood as an ordered list consisting of the pairs (Switch, position), telling whether the switch should be in straight or diverging position.

We have ten Segments and six Switches as illustrated by Figure 2.1a. Furthermore, we define four logical Routes for trains with switch positions as seen on Figure 2.1b. The direction is determined by the arrow on the final segment. Finally, Segments are monitored by two Sensors as shown in Figure 2.1c.

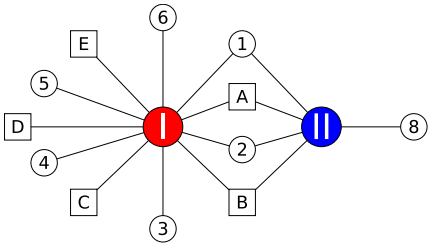
[1] We avoid the related term *semantic graph* on purpose, as it has the connotation of W3C's semantic web specification, which is not a subject of this writing.



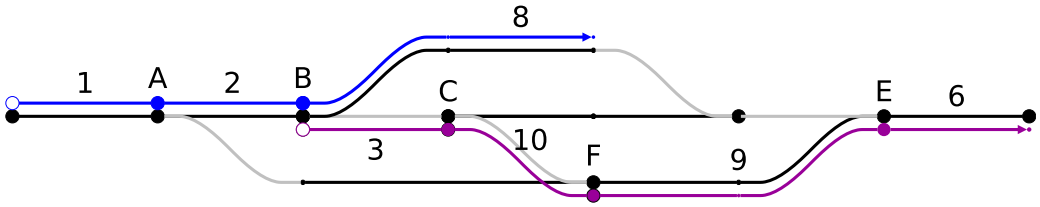
(a) Plan of the track elements comprising the railroad, with arabic digits assigned to segments, letters to switches.



(b) Four Routes in different colors, with the Segments and Switches they visit.



(c) Sensors with roman numerals, colored by the routes that require them and connected to the track elements they monitor.



(d) On Monday 18th December, 2017 at 03:09, these were the active routes and switch positions.

Figure 2.1: Illustrations for the running example.

The upper three figures share a common property: they only capture static (time-invariant) aspects of the railway system. Separation of dynamic (time-varying) and static properties is a common modeling practice. We provide a piece of dynamic information as well: Figure 2.1d is a snapshot illustrating the switch positions and active routes at a given point in time.

We ask the following four questions, included as well-formedness validation queries in the TB:

PosLength	Is there a segment whose length is not positive?
RouteSensor	Is there route that has one or more unmonitored switches?
ConnectedSegments	Is there a six-segment-long chain with a common sensor?
ActiveRoute	Is there an active route with one or more switches in the wrong position w.r.t. the route (possible derailment)[2]?

[2]this is the only query which relies on dynamic properties as well

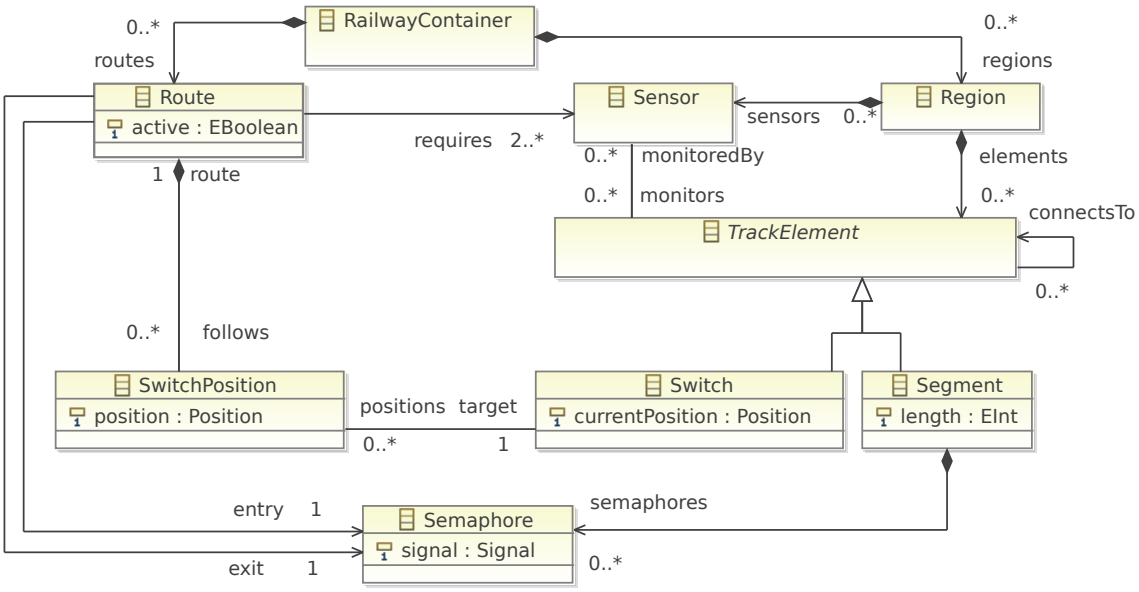


Figure 2.2: The Train Benchmark metamodel shown in an Ecore diagram.

Looking at the figures once again, one can spot that the answers to some of these questions are indeed positive. There is no segment with non-positive length of course, and let's assume that the rest of the routes have attached sensors as well in addition to those shown in Figure 2.1c, as these were omitted to keep the example small. Thus, the first two answers are negative. However, we can easily identify one six-segment-long chain with a common sensor (segments 1 to 6). As for the last question, switches C & F are in the wrong position for the purple route, detaching segment 10 from the track[3].

In the following section we procede with formalizing this model and queries in the context of the Train Benchmark.

2.2 Train Benchmark: a railway metamodel & benchmark suite

The Train Benchmark (TB), already mentioned in Section 2.1 is a macrobenchmark suite developed by the Fault Tolerant Systems Research Group. The ultimate goal of TB is to provide a cross-technology toolkit to measure the performance of continuous model validation with graph-based models and queries. It defines a metamodel[4] of a railway, well-formedness constraints as queries, and transformations. The execution is performed against typical workflow scenarios w.r.t. model-driven engineering. The framework also provides instance model generators that generate models (graphs) scalable in size.

2.2.1 Metamodel

We use the TB metamodel to describe our running example. The benchmark suite uses Ecore as its modeling language, which is one of the *de facto* standard industrial metamodeling environments, used for defining several domain-specific languages and editors [50]. The TB metamodel is presented as an Ecore diagram on Figure 2.2. Although the diagram only shows a partial metamodel (as containment hierarchy is omitted), it is sufficient for the current discussion. In fact, collating with Figure 2.1 and the track element enumeration in Section 2.1, the reader may notice that only a subset of the classes and references shown in the figure is required. Also we would like to bring to attention those familiar with object-oriented programming might already have noticed, that the Ecore diagram closely resembles an UML class diagram; in fact Ecore's metamodel is a simplified version of UML's MOF metamodel [2]. Moreover, Ecore is based on Java and its model storage is only a thin layer above the target platform's native mechanism, which includes the way references are handled.

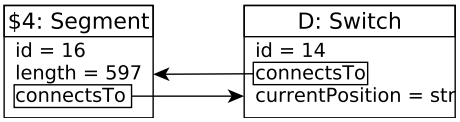


Figure 2.3: References are stored as pointers to other objects.

An outline of mapping Ecore concepts to OO and property graphs is included on Table 2.1 for clarity. We omit charting the Ecore instance model[5] of the example, as it contains more than 20 objects. Instead, we provide a property graph representation later.

2.2.2 Queries

Pattern matching, i.e., detecting patterns in a graph corresponds to finding a copy of some predefined smaller graph, thus solving a subgraph isomorphism problem with a fixed size subraph. We picked four of the TB's pattern matching problems, the already mentioned PosLength, RouteSensor, ConnectedSegments and ActiveRoute queries, shown in Figure 2.4.

In order to formalize these graph queries, we have to introduce *property graphs*, the underlying data model of the Cypher language.

2.3 Property graph data model

Property graph is a formal model for KGs, able to capture rich semantic information in separate graph elements.

2.3.1 Definition

As provided in [28], a *property graph* is defined as $G = (V, E, st, L, T, \mathcal{L}, \mathcal{T}, P_v, P_e)$ where:

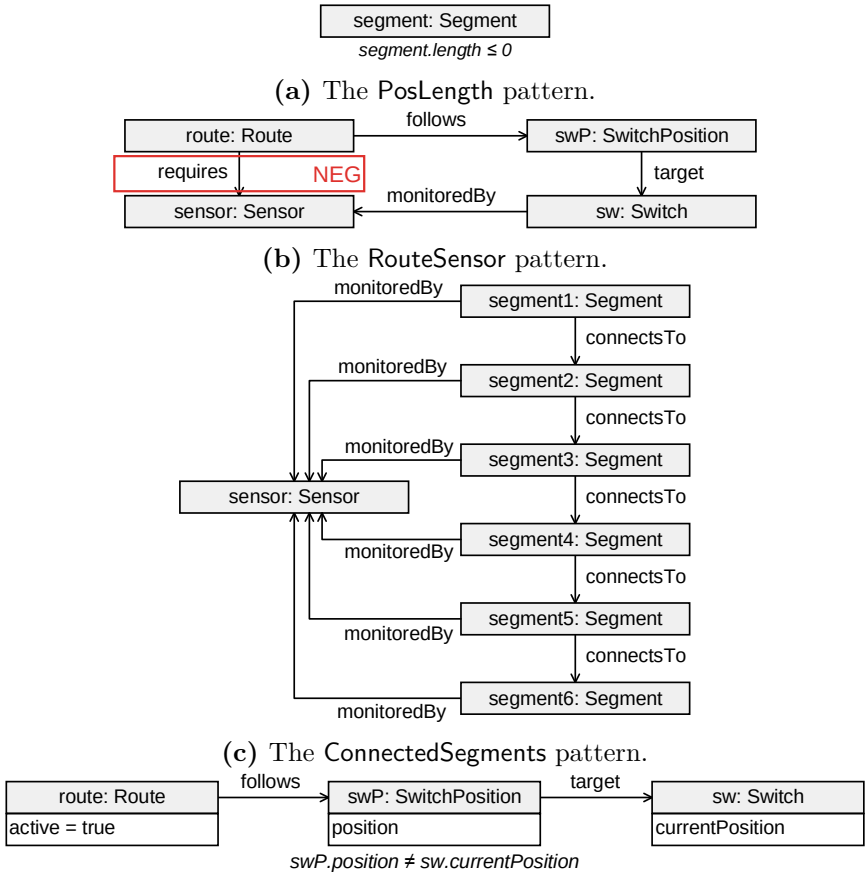
- V: the set of vertices[6]

[3]Note that the color of the segments in Figure 2.1d indicate the position of the adjacent switch from which the segment forked; and not a property of the segments themselves.

[4]in model driven engineering terminology, we usually refer to instances as *(instance) model*s, while their "domain" is called "metamodel", e.g., the Hungarian railway network model defined by the Train Benchmark railway metamodel.

[5]we can also draw an analagy between the Ecore instance diagram and UML object diagram

[6]sometimes referred to as *nodes*



(a) The PosLength pattern.

(b) The RouteSensor pattern.

(c) The ConnectedSegments pattern.

(d) The ActiveRoute pattern.

Figure 2.4: Implemented patterns from Train Benchmark [53].

- E: the set of edges[7]
- $st : E \to V \times V$: the mapping function which assigns the source, and target vertices of an edge.
- L : the set of possible vertex *labels*
- T : the set of possible edge *types*
- $\mathcal{L} = V \to 2^L$: the mapping function which assigns each vertex a set of labels.
- $\mathcal{T} = E \to T$: the mapping function which assigns each edge a type.

Note that edges have *types*, while vertices have *labels*. The difference is that edges have exactly one type, and vertices have a *set* of labels. To define the vertex properties, let $D = \bigcup_i D_i$, where D is the union of all the atomic domains D_i. Additionally, let ε represent $\perp$ (the NULL value).

- P_v is the set of vertex properties. A vertex property $p_i \in P_v$ is a (total) function $p_i : V \to D_i$ which assigns a property value from domain $D_i \in D$ to a vertex $v \in V$ if v has property p_i, otherwise $p_i(v)$ returns ε.

[7] sometimes referred to as *relationships*

OO	Ecore	Property graphs
class definition	EClass instance	vertex label or type property
reference definition	EReference instance	edge label
attribute definition	EAttribute instance	property name
type	EDataType instance	(only primitives)

Table 2.1: Mapping object-oriented concepts to Ecore and property graphs [53].

- P_e is the set of edge properties. An edge property $p_j \in P_e$ is a (total) function $p_j : E \to D_j$ which assigns a property value from domain $D_j \in D$ to a vertex $e \in E$ if e has property p_j, otherwise $p_j(v)$ returns ε.

2.3.2 Our running example as a property graph

We use the mappings shown in Table 2.1 to derive a property graph instance from our previous Ecore instance model. We transform EClasses to vertex labels as opposed to values of the type property. As property graphs typically have no schema, we discard metamodeling elements (EClasses, EReferences, etc.) without instances. On top of that, as there is no clear indication how supertype relations should be handled in the benchmark specification [28], and as all of our queries use direct (concrete) types only, we omit supertypes as well. Consequently, every vertex has a single label.

A gist of the property graph representation can be found in Figure 2.5. Most of is straightforward, however, the mapping of SwitchPositions requires some elaboration. As the TB's metamodel is rooted in object-oriented programming, some of the concepts are quite unintuitively in a property graph, namely SwitchPositions, where each instance corresponds to each unique switch, position and route tuple in the TB metamodel. In the context of the example, this means that there are different vertices for e.g., the straight position of the A switch as required by the red route (A_{str}), and for the same position of the same switch as required by the blue route (A_{str}).[8] In fact, A and D is a rather strange switches, as only one position is assigned to them this way.

2.4 Cypher: a query language for property graphs

Cypher is a declarative query language, originally created for the Neo4j graph database [31]. Since then, it was adopted by other graph DB vendors, including SAP HANA [3], AgensGraph [1], and most relevantly our project, *ingraph*.

2.4.1 Syntax

Cypher's syntax is similar to that of SQL. Among the most common clauses are: `MATCH` and `WHERE`, which are only slightly different than in SQL. `MATCH` is used for describing the structure of the pattern searched for, primarily based on edges. `WHERE` is used to add additional constraints to patterns. A simple example is shown in Source 2.4.1a. Cypher

[8] The savvy reader can notice that the original SwitchPosition objects could be easily mapped to edges between Switches and Routes instead as edges can have properties, and we can use the edge type instead of the vertex label. We decided we do not want to further complicate the discussion by including this extra rule.

$$
\begin{aligned}
L &= \{\text{Segment}, \text{Route}, \text{SwitchPosition}, \text{Switch}, \text{Sensor}\} \\
T &= \{\text{monitoredBy}, \text{target}, \text{requires}, \text{follows}, \text{positions}, \ldots\} \\
P_v &= \{\text{length}, \text{currentPosition}, \text{position}, \text{active}\} \\
P_e &= \emptyset \\
V &= \{1, 2, \ldots, 9, A, B, \ldots, F, A_{\text{str}}, A_{\text{str}}, \ldots, E_{\text{str}}, E_{\text{div}}, E_{\text{div}}, \ldots, \text{Red}, \text{Blue}, \text{Green}, \text{Purple}, \text{I}, \text{II}\} \\
E &= \{\text{Red\_requires\_I}, \ldots, \text{Blue\_requires\_II}, \text{I\_monitors\_1}, \text{1\_monitoredBy\_I}, \ldots, \\
&\quad \text{II\_monitors\_F}, \text{F\_monitoredBy\_I}, \text{1\_connectsTo\_A}, \text{A\_connectsTo\_1}, \ldots\} \\
st &: \text{1\_connectsTo\_A} \to \langle 1, A\rangle, \text{A\_connectsTo\_1} \to \langle A, 1\rangle, \ldots \\
\mathcal{L} &: 1 \to \{\text{Segment}\}, \ldots, A \to \{\text{Switch}\}, \ldots, \text{Red} \to \{\text{Route}\}, \ldots, \text{I} \to \{\text{Sensor}\}, \ldots \\
\mathcal{T} &: \{\text{Red\_requires\_I} \to \text{requires}, \ldots, \text{1\_monitoredBy\_I} \to \text{monitoredBy}, \ldots \\
\hline
\text{length} &: 1 \to 750, 2 \to 126, 3 \to 42, \ldots \\
&\vdots \\
\text{position} &: A_{\text{str}} \to \text{str}, A_{\text{str}} \to \text{str}, \ldots, E_{\text{str}} \to \text{str}, E_{\text{div}} \to \text{div}, E_{\text{div}} \to \text{div}, \ldots \\
\text{currentPosition} &: A \to \text{str}, B \to \text{div}, C \to \text{str}, \ldots
\end{aligned}
$$

Figure 2.5: The running example as a property graph.

```
MATCH (a:Acc)<-[:OWNS]-(p:Pers)
WHERE p.name = "Szakállas"
RETURN a, p.name
```

a: A simple pattern with a `WHERE` clause.

```
MATCH (a)-[:LTE]->(b),
      (a)-[:LTE]->(c)
RETURN a, b, c
```

b: Vertex matching is homomorphic.

```
MATCH (d:Dog)
WHERE d.name = "Lucy"
CREATE (a:Dog {name: "Sam"}),
       (a)-[:KNOWS]->(d),
       (d)-[:KNOWS]->(a)
```

c: Queries can mutate the graph.

Source 2.4.2: Simple Cypher examples.

additionally contains clauses for mutating data. `CREATE` and `DELETE` are used to create and delete vertices and edges. `SET` and `REMOVE` are used to set values to properties and add labels on vertices. It is important to note that we are only going to support non-mutating query operations such as `MATCH` and `WHERE` in this work.

Graph patterns in Cypher use *edge isomorphic matching*, which means that for matches of a pattern within a single `MATCH` clause, edges are required to be unique. This criteria does not apply for vertices. This is illustrated in Source 2.4.1b, where vertices are natural numbers, and the `LTE` edges are created from *less than or equal to* relations according to the usual total ordering on natural numbers (see Figure 2.6 for an example). This query can return a result containing a tuple with the same `a` and `b` values, but not with the same `b` and `c` values.

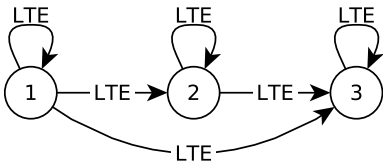


Figure 2.6: Example graph with natural numbers as vertices and *less than or equal to* (LTE) edges.

2.4.2 openCypher

The openCypher project [34] is a community effort started in 2015 under the stewardship of Neo4j, Inc., the company behind Neo4j. It aims to provide an open grammar and textual semantics specification for Cypher. While there have been attempts to formalize a subset of the language [28], defining precise formal semantics is a future goal of the openCypher project. To compensate for lack of formal semantics, the Cypher Technology Compatibility Kit (TCK) [33] project defines acceptance test covering major features of the openCypher language that prospective implementors should meet.

2.4.3 Train Benchmark queries

```
MATCH (segment:Segment)
WHERE segment.length <= 0
RETURN
  segment,
  segment.length AS length
```

a: PosLength

```
MATCH (r:Route)-[:follows]->(swP:SwitchPosition),
      (swP:SwitchPosition)-[:target]->(sw:Switch),
      (sw:Switch)-[:monitoredBy]->(sensor:Sensor)
WHERE NOT (r)-[:requires]->(sensor)
RETURN r, sensor, swP, sw
```

b: RouteSensor

```
MATCH (route:Route {active: true})-[:follows]->(swP:SwitchPosition)
      (swP:SwitchPosition)-[:target]->(sw:Switch)
WHERE swP.position <> sw.currentPosition
RETURN route, swP, sw
```

c: ActiveRoute

```
MATCH
  (sensor:Sensor)<-[:monitoredBy]-(segment1:Segment),
  (segment1:Segment)-[:connectsTo]->
  (segment2:Segment)-[:connectsTo]->
  (segment3:Segment)-[:connectsTo]->
  (segment4:Segment)-[:connectsTo]->
  (segment5:Segment)-[:connectsTo]->(segment6:Segment),
  (segment2:Segment)-[:monitoredBy]->(sensor:Sensor),
  (segment3:Segment)-[:monitoredBy]->(sensor:Sensor),
  (segment4:Segment)-[:monitoredBy]->(sensor:Sensor),
  (segment5:Segment)-[:monitoredBy]->(sensor:Sensor),
  (segment6:Segment)-[:monitoredBy]->(sensor:Sensor)
RETURN sensor, segment1, segment2, segment3,
       segment4, segment5, segment6
```

d: ConnectedSegments

Source 2.4.3: Cypher queries for the four TB patterns used in this thesis.

Source 2.4.3 shows the four TB patterns written in Cypher. The `MATCH` clause contains graph patterns, with vertices enclosed in parentheses, edges in square brackets. Both vertices and edges can be bound to fresh variables contrary to `WHERE` clauses that can only use previously bound symbols, thus they always filter the results (i.e., the result of a `WHERE` clause cannot contain more tuples than its input). In the `WHERE` clause one can

compose arbitrary boolean formulae from patterns, equality, relational operations, nullity checks, etc., with the usual boolean operators. The `RETURN` clause is used to return the results in a table (more formally a *graph relation* [28]) containing the specified variables as columns.

Cypher is a complex and expressive language with advanced features. However, this thesis focuses on a search driven approach for simpler pattern matching (navigational queries according to the classification of [6]). Consequently, and also for the sake of conciseness, some advanced language features were omitted from this thesis. Just to mention some, Cypher supports transitive navigation along edges, aggregation, unwinding lists and paths, optional matches and calling user-defined procedures [35]. The interested reader is referred to [32] for a comprehensive specification.

It is important to note that Cypher along with the majority of other property graph languages – such as Gremlin [44], G-CORE [5], GraphScript [39], and PGQL [57] (see Section 6.1.1) – requires not only vertices, but edges to be effectively retrievable on their own. For instance a query like `MATCH ()-[m:monitoredBy]->() RETURN m` might look strange from an OO point of view, but is totally valid. This is contrary to EMF or object-oriented databases (and their corresponding query languages), which focus on objects (cf. vertices), and references (cf. edges) are only second-class citizens.

2.5 Relational Graph Algebra

Relational graph algebra is an extension of relational algebra with operations specific to PGs, i.e., vertex and edge operations. The reason we include an overview on this subject is because ingraph's QPlan IR[9], from which we construct our patterns is based on this.

#ops.	**notation**	**name**	**schema**
0	$\bigcirc_{(\texttt{v}:\mathsf{L})}$	GetVertices	$\langle \texttt{v} \rangle$
	$\bigcirc\!\!\leftrightarrow\!\!\bigcirc^{(\texttt{w}:\mathsf{W})}_{(\texttt{v}:\mathsf{V})}$ [e:E]	GetEdgesUndirected	$\langle \texttt{v}, \texttt{e}, \texttt{w} \rangle$
	$\bigcirc\!\!\rightarrow\!\!\bigcirc^{(\texttt{w}:\mathsf{W})}_{(\texttt{v}:\mathsf{V})}$ [e:E]	GetEdgesDirected	$\langle \texttt{v}, \texttt{e}, \texttt{w} \rangle$
1	$\updownarrow^{(\texttt{w}:\mathsf{W})}_{(\texttt{v})}$ [e:E] (r)	ExpandBoth	$\mathrm{sch}(r) \parallel \langle \texttt{e}, \texttt{w} \rangle$
	$\uparrow^{(\texttt{w}:\mathsf{W})}_{(\texttt{v})}$ [e:E] (r)	ExpandOut	$\mathrm{sch}(r) \parallel \langle \texttt{e}, \texttt{w} \rangle$
	$\downarrow^{(\texttt{w}:\mathsf{W})}_{(\texttt{v})}$ [e:E] (r)	ExpandIn	$\mathrm{sch}(r) \parallel \langle \texttt{e}, \texttt{w} \rangle$
	$\not\equiv_{\mathsf{variables}}\ (r)$	AllDifferent	$\mathrm{sch}(r)$

Table 2.2: Graph extensions for relational algebra from [28]. $\mathrm{sch}(r)$ denotes the *schema* of relation r, a list containing r's attribute names. Appending schemas is denoted by $\parallel$.

Relational graph algebra (abbrev. RGA) was proposed by Hölsch and Grossniklaus [20] and further refined by Marton, Szárnyas and Varró [28] to establish formal semantics for Cypher based on relational algebra. In order to express graph-specific operations, the fundamental relational algebra operators such as *projection* ($\pi_{a_1,a_2,\cdots,a_n} A$), *selection* ($\sigma_{p_1,p_2,\cdots,p_n} A$), *natual join* ($A \bowtie B$), *left outer join* ($A \rtimes\!\!\!\!\!\!\bowtie B$), *antijoin* ($A \triangleright B$), etc.[10], have been extended with the ones shown in Table 2.2.

[9] intermediate representation

[10] To cover elementary relational algebra is out of scope here. Consult [14] for reference.

The most important new operators are ones that allow formulas to express navigations in the graph. Namely, the ExpandOut unary operator $\uparrow^{(\mathtt{w}:\mathsf{l}_1\wedge\ldots\wedge\mathsf{l}_n)}_{(\mathtt{v})}[\mathtt{e}:\mathsf{t}_1\vee\ldots\vee\mathsf{t}_k]\,(r)$ adds new attributes e and w to each tuple iff there is an edge e from v to w, where e has *any* of types $t_1,\ldots,t_k$, while w has *all* labels $l_1,\ldots,l_n$. More formally, this operator appends $\langle e,w\rangle$ to a tuple iff $st(e)=\langle v,w\rangle$, $l_1,\ldots,l_n\in\mathcal{L}(w)$ and $\mathcal{T}(e)\in\{t_1,\ldots,t_k\}$.

Similarly to the ExpandOut operator, the ExpandIn operator $\downarrow$ appends $\langle e,w\rangle$ iff $st(e)=\langle w,v\rangle$, while the ExpandBoth operator $\updownarrow$ uses edge e iff either $st(e)=\langle v,w\rangle$ or $st(e)=\langle w,v\rangle$.

As a starting point for navigation, instead of primitive relations (tables), nullary operators such as GetVertices (○), GetEdgesUndirected (○↔○), and GetEdgesDirected (○→○) serve as terminals that return a relation representing vertices or edges. GetVertices can optionally receive a label predicate, which can be expressed in terms of σ and ○:

$$\bigcirc_{(\mathtt{v}:\mathsf{L})}\equiv\sigma_{L\subseteq l(\mathtt{v})}\bigcirc_{(\mathtt{v})}$$

Similarly, the type predicate of GetEdgesDirected is

$$\bigcirc\!\!\rightarrow\!\!\bigcirc^{(\mathtt{w}:\mathsf{W})}_{(\mathtt{v}:\mathsf{V})}[\mathtt{e}:\mathsf{E}]\equiv\sigma_{t(\mathtt{e})\in E\,\wedge\,V\subseteq l(\mathtt{v})\,\wedge\,W\subseteq l(\mathtt{w})}\bigcirc\!\!\rightarrow\!\!\bigcirc^{(\mathtt{w})}_{(\mathtt{v})}[\mathtt{e}]$$

GetEdgesBoth is analogous. Furthermore, ○→○ can be expressed in terms of $\uparrow$ and ○:

$$\bigcirc\!\!\rightarrow\!\!\bigcirc^{(\mathtt{w}:\mathsf{W})}_{(\mathtt{v}:\mathsf{V})}[\mathtt{e}:\mathsf{E}]\equiv\uparrow^{(\mathtt{w}:\mathsf{W})}_{(\mathtt{v})}[\mathtt{e}:\mathsf{E}]\,(\bigcirc_{(\mathtt{v}:\mathsf{V})})$$

Again, ○↔○ is analogous. Equivalences such as these play an important role in logical query optimization just like in traditional relational database systems [56, Chapter IV].[11]

The AllDifferent operator $\not\equiv_{\mathtt{e}_1,\ldots,\mathtt{e}_n}$ operator – with $\mathtt{e}_1,\ldots,\mathtt{e}_n$ representing edge variables – was created specifically for the purpose of representing the edge uniqueness constraint of Cypher (discussed in Section 2.4.1) in a concise manner:[12]

$$\not\equiv_{\mathtt{e}_1,\ldots,\mathtt{e}_n}(r)=\sigma_{\bigwedge\limits_{\substack{1\le i,j\le n\\ i\neq j}} r.e_i\neq r.e_j}(r) \tag{2.1}$$

The ConnectedSegments query is a fine example for using this operator as it has many edges with the same type within a single `MATCH` clause. However, for the sake of conciseness, we use a simpler example instead, presented in Source 2.4.1b.

$$\pi_{\mathtt{a},\mathtt{b},\mathtt{c}}\not\equiv_{\mathtt{lte}_1,\mathtt{lte}_2}\left(\bigcirc\!\!\rightarrow\!\!\bigcirc^{(\mathtt{b})}_{(\mathtt{a})}[\mathtt{lte}_1{:}\mathsf{LTE}]\bowtie\bigcirc\!\!\rightarrow\!\!\bigcirc^{(\mathtt{c})}_{(\mathtt{a})}[\mathtt{lte}_2{:}\mathsf{LTE}]\right) \tag{2.2}$$

RGA expressions for the PosLength, RouteSensor and ActiveRoute patterns are shown below.

$$\pi_{\mathtt{s},\mathtt{s}.\mathtt{length}}\left(\sigma_{\mathtt{s}.\mathtt{length}\le 0}\left(\bigcirc_{(\mathtt{s}:\mathsf{Segment})}\right)\right) \qquad \text{(PosLength)}$$

[11] These equivalence rules also highlight that GetEdges∗ operators are essentially *syntax sugar* on top of the GetVertices and Expand∗ operators.

[12] Note that for edges of different types, adding the criteria is superfluous as they would not match regardless.

$$\pi_{\texttt{sw},\texttt{swP},\texttt{r},\texttt{s}}\Big(\Big(\uparrow^{(\texttt{s}:\text{Sensor})}_{(\texttt{sw})}[:\text{monitoredBy}]\uparrow^{(\texttt{sw}:\text{Switch})}_{(\texttt{swP})}[:\text{target}]\uparrow^{(\texttt{swP}:\text{SwitchPosition})}_{(\texttt{r})}[:\text{follows}]\,\circ_{(\texttt{r}:\text{Route})}\Big)\ \triangleright\ \Big(\circ\!\!\rightarrow\!\!\circ^{(\texttt{s}:\text{Sensor})}_{(\texttt{r}:\text{Route})}[:\text{requires}]\Big)\Big)\quad(\text{RouteSensor})$$

$$\pi_{\texttt{swP},\texttt{sw},\texttt{r}}\Big(\sigma_{\texttt{r.active}=\texttt{true}}\downarrow^{(\texttt{r}:\text{Route})}_{(\texttt{swP})}[:\text{follows}]\,\sigma_{\texttt{swP.position}\neq\texttt{sw.currentPosition}}\downarrow^{(\texttt{swP}:\text{SwitchPosition})}_{(\texttt{sw})}[:\text{target}]\,\circ_{(\texttt{sw}:\text{Switch})}\Big)\quad(\text{ActiveRoute})$$

Chapter 3

A dynamic programming based search plan generation algorithm

The main contribution of this thesis is the adaptation of a dynamic programming based search plan generation algorithm published in [61], proposed by Gergely Varró et al. to speed up *pattern matching* on EMF (Eclipse Modeling Framework) models. Pattern matching [60] is the process of binding variables so that they fulfill a finite set of constraints. Thus, it belongs to the family of *constraint satisfaction problems* (CSP) which often present a high complexity[1]. Therefore, in practical cases, they are tackled using some kind of heuristic or combinatorial search methods so that they can be solved in a reasonable time.

This chapter introduces the original approach to the reader in detail, outlines its strengths and shortcomings w.r.t. pattern matching on top of property graphs, and the corollary adaptation barriers. We also propose some further improvements to the algorithm.

3.1 Planning

The algorithm described in the paper comprises two-phases: *planning* and *execution* is carried out separately. The goal of the planning phase is to create an efficient search plan. *Search plan cost* is estimated in terms of the approx. size of the pattern matching state space. As full optimization of the problem would be impracticable even for smaller patterns because of exponentially growing planning state space[2], authors employ a method based on iterative dynamic programming, similar to the ones proposed by Kossman and Stocker for optimizing joins in relational query engines [24].

The algorithm goes as follows. In each iteration a set of applicable *operation*s are selected that advance the search by binding a set of variables. Operations are characterized by the following aspects:

type either check or extend

[1] In general, the CSP problem is NP-complete [58].

[2] The *planning state space* should not be confused with *pattern-matching (execution) state space.* As the name indicates, the first one corresponds to planning itself, i.e., depicts operation sequences with which a pattern can be satisfied, while the latter represents a concrete pattern matching sequence being carried out at runtime.

applicability *applicability* constrains branching (thus largely filtering planning state space) by forbidding operations to be selected at certain states. The applicability criterion is simple:

- check operations can be only considered when every one of their referenced variables are bound;
- extend operations can only be considered with one referenced variable set to bound, the rest being free.

If less than the required amount of variables are bound, the operation is called *"future"*; if more, *"past"*; if applicable, *"present"*.

weight extend operations are assigned a cost based on their approximate cardinality. check operations are not considered for cost based optimization, rather they are selected greedily. For extends, their cardinality (branching factor) is retrieved from the instance model, leveraging the reflective[3] abilities of EMF.

adornment the set of variables the operations sets to bound. For extend operations this set normally contains one variable. It is an empty set for check operations.

The substance of the approach is to choose extend operations iteratively, and only retain in each step those subplans that are in the k-best among those with the same amount of bound variables. States with the same amount of free variables are totally ordered by a cost property. The cost is calculated using an iteratively computable monotone cost function shown on Section 3.1, which takes into account the cost of the previous state c_{i-1}, the cumulative branching factor p_{i-1} and the weight (estimated branching factor) of the operation that produces the new state. Because of this, it is practical only to keep the very best of any two *variable adornment disjoint*[4] (abbrev. VAD) states, as they have identical closure w.r.t. future operations so the better subplan's descendants will always outperform those of the other. This elimination of duplicates is sometimes referred to as *pruning*. A generalized version of the cost function will be introduced later.

$$c_n = \sum_{i=1}^{n} \prod_{j=1}^{i} w_j, \quad \text{iteratively} \quad c_0 = 0, \quad p_0 = 1$$
$$c_i = c_{i-1} + p_i, \quad p_i = p_{i-1} w_i$$

Because only a constant amount is kept from the VAD states, we get a space bound guarantee depending on the number of variables and the constant factor k, avoiding exponential complexity. Furthermore, as check operations are applied instantly upon becoming available, only extend operations are creating branches.

The outline of the algorithm is shown on Algorithm 3.1.1. In the

outer loop (3–17) we progress until there are no free variables left. The final transition's target state corresponds to a complete match. Note that because of pruning, there can be only one final state.

middle loop (5–16) we iterate over each VAD state having the same amount of free variables.

[3] In software engineering, *reflection* is a property of a computer program which can be introspected and modified at runtime [12].

[4] By *variable adornment*, we refer to a function that maps variables to either free or bound symbols.

```
1   n := |a_{S_0}|_f; // number of free variables in S_0
2   T[n][1] := S_0;
3   for(i := n down to 1) {
4     S := T[i][j];
5     for(j := 1 to k) {
6       for(o ∈ O_S^{pe}) { //git for each present extension operation
7         S' := nextState(S, o);
8         i' := |a_{S'}|_f;
9         position, duplicatePosition := determinePositionByCostOrder(T[i'], S')
10        if(shouldInsert(S', T[i'], position, duplicatePosition)) {
11          updateOperations(S', S, o);
12          eliminateDuplicate(T[i'], duplicatePosition);
13          insert(T[i'], S', position)
14        }
15      }
16    }
17  }
18  return T[0][1]
```

Algorithm 3.1.1: The procedure `calculateSearchPlan`(S_0, k).

inner loop (6–15) we inspect each present extend operation for the state.

seq (7–9) we produce the next state from the operation. We determine its position based on its cost, and any duplicates for pruning.

branch (10–14) if the cost is among the k lowest and it is not a duplicate or the existing duplicate has greater cost; do

(i) update the operations by potentially moving some future operations to present ones, and discarding previously present ones,

(ii) eliminate the duplicate if exists,

(iii) insert the item.

The resulting algorithm has

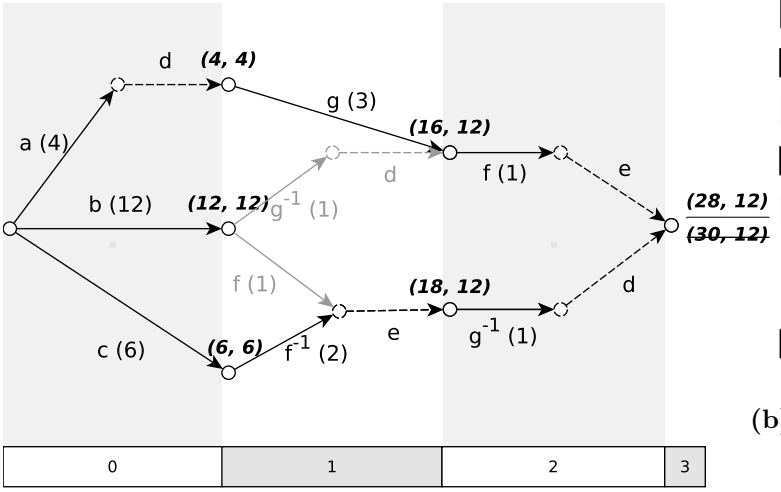
$$\mathcal{O}\left(|V|^2 \cdot |O|^2\right)$$

runtime complexity, where $|V|$ is the number of free variables, $|O|$ is the approx. number of branches (applicable operations at a step).

The complete planning state space of the ActiveRoute pattern is shown in Figure 3.1a. We were not able to find any indication in the paper on binding the initial (vertex) variable; on the contrary, the original algorithm starts out with exactly one pre-existing binding. To provide a more comprehensive example[5], we include operations for binding the first variable as well, called $s_{\mathsf{Route}}(r)$, $s_{\mathsf{SwitchPosition}}(swP)$ and $s_{\mathsf{Switch}}(sw)$ shown. The costs of extend operations have been estimated for the instance model of the running example. We also included costs for the start operations. Although inverse traversals on Ecore models are not inherently feasible due to technical reasons[6], we make a simplification in the following discussion for illustration purposes and assume it's perfectly alright to navigate backwards. This omission is even more justified by the fact that we not have such limitation with our PG indexer implementation.

[5] If we remove the root node and say, select a random starting vertex type the illustration becomes too simplistic to be worthwhile.

[6] As mentioned in Section 2.3.2, Ecore has its roots in OOP, where references are only navigable forward.



(a) Search space of the original algorithm. For each edge, the operation's (*weight*) is shown.

[a] $s_{\mathsf{Route}}(r)$ (4)

[b] $s_{\mathsf{SwitchPosition}}(swP)$ (12)

[c] $s_{\mathsf{Switch}}(sw)$ (6)

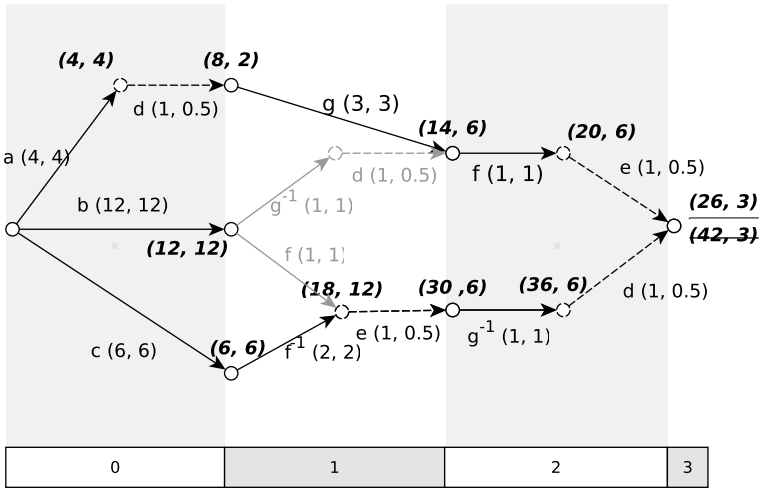
[d] $r.\mathsf{active} = \mathsf{true}$

[e] $swP.\mathsf{position} \neq sw.\mathsf{currentPosition}$

[f] $e_{\mathsf{target}}(swP, sw)$ (1 | 2)

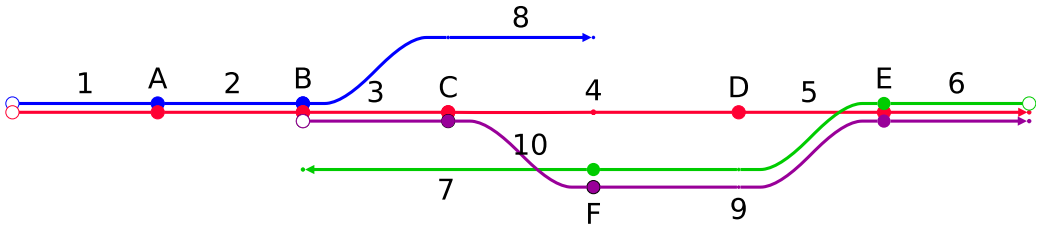
[g] $e_{\mathsf{follows}}(r, swP)$ (3 | 1)

(b) Operations for ActiveRoute. Estimated weight in parentheses for (normal | inverse) traversal.



(c) Search space of the modified algorithm. For each edge, the operation's (*cost*, *weight*) is shown.

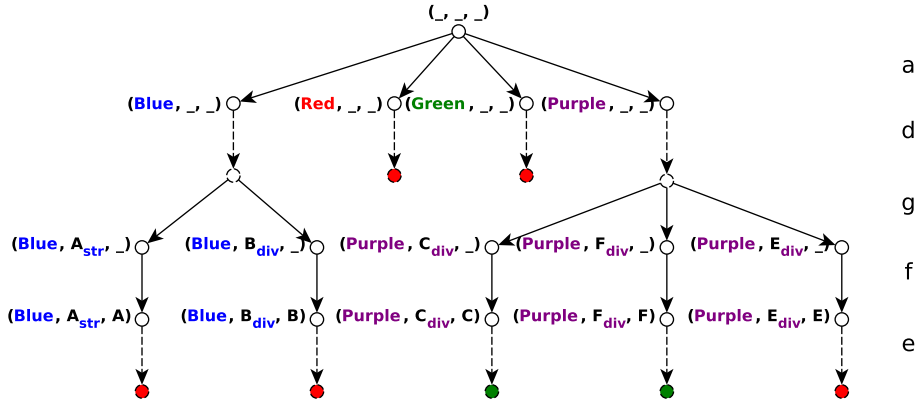
Figure 3.1: Search space of ActiveRoute during planning. Each $(0 \rightarrow 3)$ path describes the steps comprising a (complete) pattern matching sequence. The x^{-1} notation corresponds to an inverse traversal. *For each node*, the number in parentheses shows the cost of the subplan according to Section 3.1. The values with striketrough (e.g., ~~(30, 12)~~ denote cost *without pruning*. Unexplored parts of the search state graph are represented with grey nodes and edges. Dashed lines denote check operations, solid lines denote extend operations.



(a) Routes in the railway example (repeat of Figure 2.1d to make the pattern matching example easier to follow).

[a] $s_{\mathsf{Route}}(r)$, [b] $s_{\mathsf{SwitchPosition}}(swP)$, [c] $s_{\mathsf{Switch}}(sw)$, [d] $r.\mathsf{active} = \mathsf{true}$,
[e] $swP.\mathsf{position} \neq sw.\mathsf{currentPosition}$, [f] $e_{\mathsf{target}}(swP, sw)$, [g] $e_{\mathsf{follows}}(r, swP)$

(b) Operation costs (based on Figure 3.1b).



(c) The complete pattern matching state space of pattern ActiveRoute for sequence $adgfe$.

Figure 3.2: Pattern matching sequence of pattern ActiveRoute.

An important feature of the original algorithm is the *model-sensitive* cost estimation. This means that the operation weights are obtained using statistical data collected from the Ecore instance, viz. aggregate object and link counters (grouped by their corresponding types) from which average branching factors are calculated. Namely, forward navigation (cf. ExpandOut $\uparrow^{(w)}_{(v)}$[e:E]) cardinality is calculated by dividing the link count by the source object count, backward navigation by dividing the link count by the target object count.

3.2 Search plan evaluation

Search plan evaluation is touched by paper [61] exclusively when the resulting pattern matching state space is compared to that of other search plan generation approaches. The output of the planning phase is an ordered list of operations which are mapped to platform-dependent constraint evaluation operations, which we refer to as *tasks* henceforth in order to disambiguate the term. Each task binds zero (check) or one (extend) variable; and as a single variable can have multiple substitutions, the list naturally unfolds a rose tree[7] of primitive constraint evaluation steps during execution. An existing adaptation of the algorithm has been implemented by Búr et al. [9] for the VIATRA query engine (formerly called EMF-INCQUERY), facilitating depth-first traversal. More sophisticated methods could be examined, but are out of scope of our current discussion.

[7] A tree data structure with a variable and unbounded number of branches per node; a term commonly used in functional programming.

Figure 3.2 illustrates the execution part of the search plan of Figure 3.1a for the most efficient operation sequence found by the planner, $adgfe$. We traverse the search tree using depth first search, maintaining an initially empty result bag and a task stack. Popping the top of the task stack (last column) and applying to the context of the current submatch results in 0 or more children contexts. If a tree has no more tasks left, the substitution tuple contained in the current context is appended to the results.

3.3 Strengths, shortcomings & adaptation barriers

3.3.1 Strength: Polynomial complexity for planning

The original paper gives a quantitative evaluation in which the approach is compared with two others, viz. a domain-sensitive and a graph-based one[8], and indicates significantly better results for the model-sensitive approach. Furthermore, the complexity of the algorithm, $\mathcal{O}\left(|V|^2 \cdot |O|^2\right)$, is favorable taking into account that the greedy algorithm for database join optimization would require $\mathcal{O}\left(n^3\right)$ where n is the number of relations involved [24]. Note that the two problems are very similar as links can be mapped to relations.

3.3.2 Shortcomings: Missing initial case & special handling of check operations

We identified a few shortfalls or trade-offs that are not explicitly mentioned in the paper, and list them hereby.

Missing initial case. The algorithm starts with exactly one bound variable, and the authors do not state the reasons why this is the case; we can only guess that they want to run pattern matching process from a given variable. However as such, it hinders the method's applicability in scenarios where there is no specific entry point for the search, as in our case. The simplest resolution of the problem is to introduce a start operation type with its adornment consisting of a single free variable.

Check operations are ignored from estimation. As we have shown, these operations are applied immediately (greedily) upon becoming available. This is a trade-off between optimality and performance, and is justified by the fact that these operations:

- can be applied in any order and will never conflict with any other operations, i.e., there is no state and check operation which lead to a state where some operations (except the chosen one) are moved to the past set.
- are selective, i.e., they filter the set of candidates. Thus their cardinality is $0 \leq w_{o_{check}} \leq 1$. Most of the times applying them as early as possible yields very good results.

However, the fact that *these operations do not have weights assigned to them and in turn do not take part in the cost estimation* results in less accurate search plans while at the

[8] The domain-specific approach uses the same algorithm, except that cost-estimation is exchanged to one that has no information on the underlying model, instead it is based on the metamodel, thus is less accurate. The graph-based approach uses a different algorithm. Results indicate that both are outperformed w.r.t. the PM state space generated. Sadly, planning performance is not measured in paper [61].

same time improving nothing in planning performance. Furthermore, we fail to understand why the states generated by check operations are ignored from the state space. We argue that the execution time of most check operations is comparable to that of an extend operation. While equality and other relational checks are significantly faster than a hash-table lookup usually required by navigation, property value checks can also result in the latter on certain implementations; and some assertions like matching regular expressions can take significantly longer. However, to maintain a simpler model, we assume that every operation's cost is 1 for each search step generated.

Taking into account the arguments of the previous paragraph, one of the problems that arise can be immediately seen collating the plan's final cumulative branching factor in Figure 3.1a with that of the concrete execution in Figure 3.2, which shows overestimation by a factor of 6 (estimated: 12, actual: 2). This is significant for such a small query. The estimated and actual state space counting every arrow are 28 and 23, respectively. The reader can notice that this is cheating, because in the original definition, checks do not count towards the cost. Actually if we remove them, the difference becomes significant (28 to 14); so the cost is not a very good estimation according to the original definition either.

Introducing weights for check operations practically require some information which can be obtained from the model similarly to extends if feasible. Another obstacle is that the cost of these operations are not the same as their branching factor. In fact their cost is uniformly 1 as they need to be checked once for each existing branch, however their branching factor is $0 \leq w_{o_{check}} \leq 1$ as mentioned earlier. It is important to note that immediate application and cost estimation are orthogonal to each other. In particular, we can immediately apply an operation, refresh the cost, apply another, refresh the cost, and so on, without altering anything else in the algorithm body.

We generalized the iterative cost calculation function, with the original motive to enable the iterative calculation of arbitrary long operation sequence chunks (see Section A.1). An unexpected collateral benefit of using the generalized version is that is also makes it possible to have separate cost and weight estimation for atomic operations.

$$c_i = c_{i-1} + p_i \sigma_i, \quad p_i = p_{i-1} w_i$$

In the equations above, w and σ denote the operation's weight and cost, respectively. Setting $\sigma = w$ for every extend operation results in the original method. With this, we can introduce costs and weights to check operations:

(f) $r.\mathsf{active} = \mathsf{true}$ (0.5)

(g) $swP.\mathsf{position} \neq sw.\mathsf{currentPosition}$ (0.5)

Figure 3.1c shows the plan created by the modified algorithm and its execution is shown in Figure 3.2. We can see that the estimation of the branching factor is much better (1.5× overestimation). The estimated and actual state space – counting every arrow, as this required by the modified definition – is 26 to 23. An evaluation tree of the runner-up sequence $cf^{-1}eg^{-1}d$ is shown in Figure A.2.1.

We would like to stress, however, that this is only an illustration, and a quantitative evaluation would be required to prove the superiority of this method over the original. This is not discussed in this thesis and is left for future work.

	$adgfe$		$cf^{-1}eg^{-1}d$		
	c	p	c	p	
w/o check, original definition	28.00	12.00	30.00	12.00	
	14.00	2.00	22.00	2.00	
	2.00	6.00	1.36	6.00	overestimation
w/o check, new definition	28.00	12.00	30.00	12.00	
	23.00	2.00	38.00	2.00	
	1.22	6.00	0.79	6.00	overestimation
w/ check	26.00	3.00	42.00	3.00	
	23.00	2.00	38.00	2.00	
	1.13	1.50	1.11	1.50	overestimation

Table 3.1: Estimated weights for the operations and the degree of overestimation.

3.3.3 Limitation: Applicability for property graphs

The authors proposed the method to generate search plans specifically for Ecore models. The implementation by Búr, which has the closest resemblance to our approach [8], comes from this domain as well. We have already seen that there are several differences between PGs and Ecore models in Table 2.1. In this section, we summarize the obstacles raised by these differences.

Edges as first-class citizens require altered definiton of operations. In Ecore, models what we refer to as *"edges"* are essentially references between objects, and cannot be retrieved on their own. They do not store properties and each reference's identity is tied to that of the source object, the target object and the reference's name (as shown in Figure 2.3). We have seen that in RGA and Cypher[9] edges are handled as first-class citizens, i.e., identifiers can be assigned to them and they might have properties like vertices. In fact, every property graph implementation in our knowledge uses edges with their own identity (usually represented with an underlying object), and are retrievable on their own from the storage layer. This requires us to alter the original operations in a way that they can reference variables assigned not only to vertices, but edges as well.

Schemaless operations require complete overhaul of the estimator functions. As described by the original paper, extend operations carry information about the source object's class and the navigated reference's name, which can be inferred from the query at all times. This is used by the weight estimator to retrieve statistics from the underlying platform.

In contrast to Ecore models, property graphs are schemaless, and – as we have already mentioned earlier – edges are represented independently from vertices earlier; which make the original estimation method inapplicable. Also, it is not necessary to specify either labels or types in Cypher patterns, which makes finding a good estimation method even more complicated. As the statistical analysis of dynamic graphs are out of scope, we implemented only rudimentary method on in this thesis. We admit that this is a major limitation of the current work.

Richer indexers. Stemming from the dynamic (schemaless) nature of property graphs, creating an efficient indexer is more difficult than for Ecore (Chapter 5).

[9] and also in modern property graph languages, see Section 6.1.1

Chapter 4

Overview of the approach

In this chapter we give a detailed presentation of the theoretical background behind our search planner called SRE. The chapter starts out with describing the fundamental concepts, such as constraints, patterns and operations. Next, we give a mapping from the relational graph algebra tree used as the intermediate representation in ingraph to a pattern based representation specific to SRE. Continuing, we lay out the design of the search planner and the pattern compiler.

4.1 General Concepts

Constraints. We define a set of variables, a set of domains, and a set of constraints. Variables and domains are associated: the domain of a variable contains all values the variable can take. A constraint is composed of a sequence of variables, called its scope, and a set of their evaluations, which are the evaluations satisfying the constraint [47].

$$\mathbf{X} = \{X_1, X_2, \ldots, X_n\}$$
$$\mathbb{D} = \{D_1, D_2, \ldots, D_n\}$$
$$\mathbb{C} = \{C_1, C_2, \ldots, C_m\}$$

An evaluation of the variables is a function from a subset of variables to a particular set of values in the corresponding subset of domains. Constraints compose with conjunction, that is a (full) variable evaluation satisfies the set of constraints if every constraint is satisfied by the variable evaluation.

Constraint type. There is typing relation, mapping constraints to constraint types $t^C : \mathbb{C} \to \mathbb{T}^C$. Constraint types define a *name*, an ordered list of *symbolic variables* and an *implication* relation for the constraint.

Note: C denotes a constraint, $\mathbb{C}$ the constraint set, t^C constraint type, $\mathbb{T}^C$ the set of constraint types.

Implication relation. In SRE constraints form a preorder w.r.t. the function $i_{t^C} : \mathbb{C}_t^C \to 2^{\mathbb{C}}$, which we call implication. That is, $c <:_{i_{t^C}} d$ if $c \in \mathbb{C}_t^C$ and $d \in i_{t^C}(c)$, which conveys that every satisfiable evaluation of c is also a satisfiable evaluation of d with the variable mapping given by fi_{t^C}. Formally:

$$
\begin{aligned}
X_c &= \text{Vars}\{t_c^O\} \\
X_d &= \text{Vars}\{t_d^O\} \\
c <:_{I_{tC}} d \quad &\text{iff} \quad X_d \subseteq X_c \quad \wedge \forall x_1, \ldots, x_i, X_1, \ldots, X_i : \\
&\{X_1, \ldots, X_i\} = X_C \wedge c(X_1|x_1, .., X_i|x_i) = true \\
&\implies d(\pi_{V_d}(X_1|x_1, .., X_i|x_i)) = true
\end{aligned}
$$

Example 1. $c(1,2,3)$ *a constraint with its type* $\text{Name}(t_c^O) = \mathsf{Edge}$, $\text{Vars}(t_c^O) = (\mathsf{source}, \mathsf{link}, \mathsf{target})$, *and* $i_{t_c^O}(c_{\mathsf{Edge}}) = \{c_{\mathsf{Vertex}}(1), c_{\mathsf{Vertex}}(3)\}$

Implication closure. We call the set of constraints implied by c its *implication closure*, noted by c^+. We denote the implication closure of a set of constraints the same way (C^+) such as $C^+ = c_1^+ \cup \ldots \cup c_i^+$ iff $\mathbb{C} = \{c_1, \ldots, c_i\}$.

The special Known constraint. Furthermore we demand that every constraint imply the special unary constraint $\top(v)$ (also Known) for each of its variables.

Operations. An operation is defined as $o(v_1, v_2, v_3, \ldots, v_i)$ where vs are the variable bindings of the operation. Similarly to constraints, operation have types too.

Operation requirements and postconditions. Each operation has a type that specifies a name, an ordered list of symbolic variables and - here comes the difference between constraints and operations - maps the operation to *requirements* and *postconditions*.

$$
\begin{aligned}
\mathbf{R} &= \text{req}(o) \\
\mathbf{P} &= \text{post}(o) \\
\text{where } &R \subset \mathbb{C},\ \mathbf{R} = R^+, \\
&P \subseteq \mathbb{C},\ P \subset R,\ \mathbf{P} = P^+
\end{aligned}
$$

where $\mathbf{R}$ is the requirement set (or precondition set), $\mathbf{P}$ the postcondition set of the operation. On line 3 and 4 of the formula require these sets to be transitively complete w.r.t. implications. The set of postconditions should be strictly larger than the requirements, so that the operation satisfies at least one constraint. For each of the two sets, each operation type specifies constraint types and a function that binds the operation type's symbolic variables to that of each of the constraint types.

Example 2. $o(4)$ *is an operation with a type specifying* $\text{Name}_{t_o^O}(o) = \mathsf{GetVertices}$. $\text{Vars}(t_o^O) = (\mathsf{v})$, $\text{Req}(t_o^O) = \emptyset$ *and* $\text{Post}(t_o^O) = \{\{\mathsf{Vertex} : v\}\}$. *This means that* $\text{req}(o) = \emptyset$ *and* $\text{post}(o) = c_{\mathsf{Vertex}}(4)$.

Notation analogous to constraints: O, $\mathbb{O}$, t^O, $\mathbb{T}^O$.

Operation bias. Operation types also define a function $b_{tO} : \mathbb{O}_t^O \to \{\mathsf{R}, \mathsf{I}, \mathsf{D}\}$. The three values correspond to *regular*, *immediate* and *deferred* and govern how operations are chosen by the planner. We will see more on this later.

Configuration.

$$\mathcal{C}\langle \mathbb{T}^C, \mathbb{T}^O, e, t \rangle$$

is called the search *configuration*, where

$\mathbb{T}^C$ is a set of constraint types.

$\mathbb{T}^O$ is a set of operation types.

e is a total function $\mathbb{O} \to F_{e_{\mathbb{T}^O}}$ mapping o to its operation type's *estimator function*.

t is a total function $\mathbb{O} \to F_{t_{\mathbb{T}^O}}$ mapping o operation to its operation type's *task function*.

Scope.

$$\mathcal{S}\langle \mathcal{C}, \mathbb{C}^f, \mathbb{C}^b, \mathbf{V}, k, \mathsf{Ctx}^e \rangle$$

is called the search *scope*[1], where

$\mathcal{C}$ is the search configuration.

$\mathbb{C}^f$ is a set of *free* constraints. Note that it is invalid to have a constraint whose constraint type is not in $\mathbb{T}^C$.

$\mathbb{C}^b$ is a set of *bound* constraints. We require $\mathbb{C}^f$ and $\mathbb{C}^b$ to be disjoint.

$\mathbf{V}$ is the set of variables referenced by the constraints in $\mathbb{C}$.

k is the parameter already discussed in Section 3.1 used for tuning the search algorithm between optimality and performance.

Ctx^e Context provides the interface for accessing the underlying platform. During planning an *estimation context* is needed. We use the term *context-sensitive* to refer to the capability of inspecting the underlying graph elements.[2]

Operation applicability A set of operations can be split up into four categories with respect to $\mathbb{C}^f$ and $\mathbb{C}^b$.

If (a) $\mathbf{R}_O \subseteq \mathbb{C}^b$ and (b) $\mathbf{P}_O \subseteq \mathbb{C}^f$ we call the operation a *present* operation ($O \in \mathbb{O}^{Pr}_{\mathbb{C}}$). If item (a) holds and item (b) is violated, we call it a *past* operation ($O \in \mathbb{O}^{P}_{\mathbb{C}}$). If item (a) is violated and item (b) holds we call it a *future* operation ($O \in \mathbb{O}^{F}_{\mathbb{C}}$) ($\mathbb{O}^F$). We call these three sets *valid* operations w.r.t. to $\mathbb{C}$. If both is violated, the operation is *invalid*. Our search planning algorithm will never create invalid operations. If we denote $\mathbb{O}_{\mathbb{C}}$ the operations created for $\mathbb{C}$, $\mathbb{O}^{F}_{\mathbb{C}}$, $\mathbb{O}^{Pr}_{\mathbb{C}}$, $\mathbb{O}^{P}_{\mathbb{C}}$ is a full partitioning, thus e.g., the following equation holds: $\overline{\mathbb{O}^{P}_{\mathbb{C}}} = \mathbb{O}^{Pr}_{\mathbb{C}} \cup \mathbb{O}^{F}_{\mathbb{C}}$. Here we call $\overline{\mathbb{O}^{P}_{\mathbb{C}}}$ the set of non-past operations.

4.2 Search planning

Our search algorithm comprises two phases

(a) *planning*: assembling an efficient search operation sequence,

(b) *execution*: evaluating the search operations on the underlying graph;

following the approach used by Varró et al. [61]. In this section, we discuss the planning phase in detail.

Data structures

In the subsequent algorithms, we use various data structures, viz., lists, sets, sorted sets, maps (sometimes also called associative arrays, dictionaries or hashes), sorted maps and vectors.

[1] not to be confused with the scope of a constraint, mentioned earlier

[2] in lieu of *model-sensitive* used by MDE jargon.

Set We assume the reader is familiar with this basic data structure for storing unique elements efficiently. We use conventional mathematic notation for all operators, values, etc..

Sorted sets are sets with a particular total ordering relation between elements. They are denoted with $\emptyset_{\leq}$, $\{a, b, c, \ldots\}_{\leq}$, etc.. Additionally to set operations, one can get the least element $s = \bigvee S_{\leq} s,\ z \in S,\ \forall z : s \leq z$. For simplicity we call this `first` in the code.

Maps are partial functions with a finite set as domain called *keys*. Maps are denoted with $\emptyset^{M}$, $\{a \to 1, b \to 2, \ldots\}^{M}$, etc.. $\operatorname{Dom} m$ is used to get the keys and $m(x)$ to get the value at x. Furthermore we use $M \backslash S$ to remove a set of keys from a map, $M \cup N$ to merge two maps where conflicting keys are included from N.

Sorted maps are similar to maps, but their domain is a sorted set. They are denoted by $\emptyset^{M_{\leq}}$, $\{a \to 1, b \to 2, \ldots\}^{M_{\leq}}$ etc..

Vectors are

4.2.1 The `compile_search_plan` procedure

During planning, an operation sequence is created for the constraints using a dynamic search algorithm adapted from the one discussed in the previous chapter. Algorithm 4.2.1 shows the pseudocode of the body of the search planner algorithm called `compile_search_plan`, which gets as its single argument the search scope ($\mathcal{S}$).

```
1  n = |C_S^f|; # number of free constraints in S
2  non_past_ops = flat_map(lambda ot: bind_Post(C_S^f, ot), T_{C_S}^O)
3  [c, p] = [0, 1] # set the cost calculator
4  cell = ⟨non_past_ops, k, n, c, p, C_S^b, C_S^f⟩
5  plans = {n → {
6    cells: {cell}_{<c},
7    by_free: {C_S^f → cell}
8  } }^{M_>}
9  def run(plans):
10   keys = Dom plans
11   if keys is {0}: # finished searching
12     return first(plans(0).cells) # return best
13   j = first(keys)
14   column = plans(j)
15   return run(reduce(partial(step, k, C_S), plans\{j}, column.cells))
16 return run(plans)
```

Algorithm 4.2.1: The procedure `compile_search_plan`($\mathcal{S}$).

We determine the initial number of free constraints, and store the value in `n` (line 6). We partially construct all non-past operations for the constraints. This is done on the next line using the $\texttt{bind}_{\mathsf{Post}}$ function, which takes as input the free contraints in scope, an operation type, and matches symbolic variables in each of the operation type's post-conditions against the given free constraint set, to create a partial binding of operations in all ways that satisfy some constraints in the free constraint set.

On *line 4* we initialize the initial state of the cost calculator.

Next, we create a `cell` tuple that contains the optimality parameter (`n`), the number of free constraints (`n`), cost calculator attributes (`c` and `p`), the initial constraint set ($\mathbb{C}_{\mathcal{S}}^{b}$

and $\mathbb{C}^f_{\mathcal{S}}$), and the initial non-past operation bindings (`non_past_ops`). Cells serve the same purpose as the table cells in the original algorithm, i.e., they store information about a partial binding, each of them adding a state to the planning state space. The cell created here serves as the root state.

Next, we create a sorted map which stores *columns* (cf. the original algorithm), which are (instead of vectors that one expects) objects having two properties: `cells`, `by_free`. The first property is more important here, which is a sorted set that stores cells based on their costs by descending order. This data structure is used to leverage efficcient access and mutation on the most costly operation. We will return to the `by_free` property later. The resulting column is keyed by n (the number of free constraints in the cell) and sorted in descending order. The data structure was chosen for efficiency here as well. We define the `run` recursive procedure on line 9. In the body of this procedure, we check if the only remaining column is the one that has 0 free constraints (line 12). This serves as the recursion's base case and the last step of the `compile_search_plan` procedure as well. In the other case, we remove the column from the plan, and reduce the column's cells over the modified plan with the `step` function.

The `step` function is expected to advance the search, i.e., create new states having less free and more bound constraints. This justifies the removal of the current column as new candidates will have less free variables, thus they will be attempted to be inserted into columns specified by an index smaller than n.

4.2.2 The `step` procedure

We follow with the introduction of the `step` procedure, which is used to process a single cell, during which we may create new cells and insert them into the plan. We have 3 separate categories for operations according to bias as we already mentioned it earlier: these are regular, immediate and deferred operations.

Operation bias

Recalling the original algorithm, operations called `check` were chosen in a greedy manner. This means that all present operations are immediately appended to the search plan upon becoming available, which can reduce the amount of the search state spaces significantly. As the original implementation inserts cells based on variables and `check` operations do not change the variable adornment of a cell, appending these does not result in any state transitions.

We wanted to support such *immediate* operations because of the state size reduction, and the fact we seldom achieve better search plans with the optimization of certain operations. In the current algorithm however, we advance the search space everytime an operation is chosen, because (as stated in the previous section) each operation has to satisfy at least one constraint, and in so doing, the operation decrements the number of free constraints; resulting in a cell in a new column. This means that in each step, at most one immediate operation can be chosen. We still achieve search space shrinkage with this as can be easily verifed.

Futhermore we add greedy operations for the opposite purpose as well - *deferred* operations are only appended as a "last resort", i.e., when no regular or immediate operation can be appended in the current state.

This procedure is relatively simple. First, it groups `non_past_bindings` according to their bias. Then it tries to create present bindings for immediate operations and if there

exists at least one, an attempt is made to insert the first into the plans (line 5). Being greedy, the algorithm returns in this case without considering any other operation. However, if an applicable immediate operation does not exist, an attempt is made to construct applicable regular operations. Here, insertion of each regular operation is attempted on line 8 (exactly like in the original algorithm). In case no regular operation exists, we fall back to deferreds.

```
1     [non_past_bindings, …] = cell
2     [regulars, immediates, deferreds] = group_by(bias_{t_C^O}, non_past_bindings)
3     𝕆^(I,Pr) = flat_map(lambda i: bind_Pre(ℂ^b_S, i), immediates)
4     if 𝕆^(I,Pr) is not ∅: # we have an immediate operation
5       return insert_cell(cell, k, 𝒞_S, plans, first(𝕆^(I,Pr))) # greedy
6     𝕆^(R,Pr) = flat_map(lambda r: bind_Pre(ℂ^b_S, r), regulars)
7     if 𝕆^(R,Pr) is not ∅:
8       return reduce(partial(insert_cell, cell, k, 𝒞_S), plans, 𝕆^(R,Pr))
9     𝕆^(D,Pr) = flat_map(lambda d: bind_Pre(ℂ^b_S, d), deferreds)
10    if 𝕆^(D,Pr) is not ∅:
11      return insert_cell(cell, k, 𝒞_S, plans, first(𝕆^(D,Pr))) # greedy
```

Algorithm 4.2.2: The procedure `step`(k, $\mathcal{C}_S$, `plans`, `cell`)

4.2.3 The `insert_cell` procedure

The `insert_cell` procedure attempts to insert a cell into the search plan. It gets the previous state (`cell`) and the operation causing the state transition (`o`). It calculates the operation's weight and cost parameters with the help of the estimation context (which provides access to the underlying platform) on line 4. On the next line, the algorithm updates the cost with the generalized iterative cost function already mentioned in Chapter 3. We adjust the set of free and bound constraints, to reflect the changes after the operation is applied, i.e., move the satisfied constraints (post(`o`)) from free to bound. On line 8 we create the prospective cell waiting to be inserted.

The prospective cell has to meet one of the following criteria to get inserted.

(a) If an existing cell has the same set of free constraints ($\mathbb{C}^f_S$) as the prospective insert, they are subject to pruning (line 13).[3] In this case we retain the one with better cost, and discard the other, that is, the prospective cell is inserted only if it is better than the existing (line 14).

(b) If the prospective cell has no duplicate then we need to check if the column is full, i.e.,, that there are already k items in the column (line 24). If the column is *not* full, then we can insert the cell into the free slot any time (line 32).

(c) If the column is full however, we can only insert if it is better than the worst one stored (line 33).

(d) Otherwise, we discard the cell and do not modify the plan.

In this algorithm we have seen the use of `cells.by_free`. It serves as a fast lookup to check whether the candidate cell has a duplicate in the column.

[3] similarly to two cells having the same variable adornment in the original algorithm

```
⟨non_past_ops,k,n,c,p,ℂ_S^b,ℂ_S^f⟩ = cell
i = |ℂ_S^f \ P_O|
column = plans(i)
[σ, w] = e_{C_S}(o)
[next_c, next_p] = update_cost(σ, w)
next_ℂ_S^f = ℂ_S^f \ post(o)
next_ℂ_S^b = ℂ_S^b ∪ post(o)
next_cell = create_search_plan_cell(k, i, next_c, next_p, next_ℂ_S^f, next_ℂ_S^b)

dupe = column.by_free(next_ℂ_S^f) if next_ℂ_S^f ∈ Dom(column.by_free) else None
is_full = count(column.cells) >= k

if dupe:
  if (c < dupe.c): # this is better, evict dupe
    return compose(
      lambda p: update_in(p,
                          [i, 'cells'],
                          lambda cells: (cells\dupe) ∪ next_cell ),
      lambda p: update_in(p,
                          [i, 'by_free'],
                          lambda by_free: by_free ∪ {next_ℂ_S^f → next_cell}))(plans)
  else:
    return plan # don't change
elif (not is_full) or (c < first(column.cells).c):
  add_cell = compose(lambda p: update_in(p,
                                         [i, 'cells'],
                                         lambda cells: cells ∪ next_cell),
                     lambda p: update_in(next_plans,
                                         [i, 'by_free'],
                                         lambda by_free: by_free ∪ {next_ℂ_S^f → next_cell}))
  if (not is_full):
    return add_cell(plans)
  else:
    evicted = first(column.cells)
    return compose(
      add_cell,
      lambda p: update_in(next_plans,
                          [i, 'cells'],
                          lambda cells: cells\dupe),
      lambda p: update_in(next_plans,
                          [i, 'by_free'],
                          lambda by_free: by_free\next_ℂ_S^f))(plans)
else:
  return plans
```

Algorithm 4.2.3: The procedure `insert_cell`(`cell`, `k`, $\mathcal{C}_S$, `plans`, `o`)

4.3 Extending the search planner with patterns

The planner as described above only supports constraints in conjunctive form. However we have seen in Source 2.4.3b that Cypher queries can contain negated patterns as well.[4]

In order to support queries that contain negative patterns, the search planning algorithm has to be extended. We have seen in the previous sections that the planner works by finding an operation sequence for the input constraints. The operation sequence, when evaluated, yields all results that satisfy all the constraints. On the contrary, a negative

[4] They can also contain disjunctive patterns and have a special `OPTIONAL` modifier that permits `NULL` values. In this thesis we include support for negative patterns

pattern means that given some non-empty, already satisfied ($\mathbb{C}^b$) constraint set, and an also non-empty set of some free constraints ($\mathbb{C}^f$), there exists no evalution that satisfies ($\mathbb{C}^f$). Thus the negative pattern can be represented with a unique constraint that has to be satisfied during search by a *composite* operation that comprises an inner subsearch. If this operation finds a match, it returns a failure, else it returns the variables corresponding to the (partial) binding it started the search from.

Patterns A subsearch, like the one corresponding to a negative Cypher pattern, is introduced into sre under the same name - *pattern*. Similarly to constraints, patterns represent a constraint to be satisfied via search, thus they can be input the search planner. However, patterns get a *unique*, *fresh* type, which means that contrary to constraints, they can't be matched to platform operations, as simply there will be no operation that satisfies them. This is why patterns also specify a *unique*, *fresh* operation type, which is added to the search scope to satisfy that single pattern type. A pattern corresponds to what some others refer to as a "pattern call", i.e., it contains the bindings for variables, referring to an outer pattern (if exists). We refer to a pattern definition as a *pattern type*. We chose this naming to remain consistent with operations and constraints, where the term "call" does not apply, and the relationships between them and their corresponding types are the same as with patterns.

Pattern type. A pattern type is created whenever a new pattern is defined. Currently we only support anonymous, in-place pattern definitions. This suffices for our use-case as patterns cannot be reused in Cypher.

Pattern kinds. A family of pattern types. Only three pattern kinds are supported as of now:

Unit the kind of primitive (atomic) constraints. UNIT patterns do not modify the parent search configuration in any manner:

- They do not require compilation, because they do not create a search scope.
- They do not add a constraint type, as platform specific constraints are natively supported.
- They do not define operations as they have to be satisfied using platform specific operations.
- They add one constraint to the parent search scope.

A pattern type with the kind UNIT cannot be the root pattern.

Conj is the kind of conjuctive patterns defining a subsearch. Every pattern listed in the body of the CONJ pattern has to be satisfied.

- A subsearch is compiled for each pattern type of the CONJ kind.
- Conj types add a unique, fresh constraint type t_i^C to the parent search configuration.
- They also add a unique, fresh operation type t_i^O to the parent search configuration. Naturally they add an estimator function, and a task function for the operation type as well, and extend the domain of e and t in the configuration to map the operation type to the specified estimation and task function respectively.
- they add one constraint (with the type of t_i^O to the parent search scope).

Neg the family of negative patterns. Negative patterns have an one inner CONJ pattern.

- A subsearch is not compiled, NEG pattern types use the underlying CONJ search.

```
compiled_specs = group_compiled_specs(map(lambda s: compile(s, C_parent, opts), specs))
ℂ = compiled_specs.constraints
# the configuration modifications of the inner patterns are applied
# to the current configuration \sreconf_{\atom{this}}
𝕋^C = reduce(update_ct,C_parent, compiled_specs.constraint_types)
𝕋^O = reduce(update_ot,C_parent, compiled_specs.operation_types)
e = reduce(update_e, C_parent, compiled_specs.estimators)
t = reduce(update_t, C_parent, compiled_specs.tasks)
# create the configuration
C_this = ⟨𝕋^C, 𝕋^O, e, t⟩

# create free and bound constraints
[ℂ^f, P] = [(conj_pattern.reqs)^+, (conj_pattern.constraints\conj_pattern.reqs)^+]
V = vars(ℂ^f ∪ ℂ^b)
# create the scope
S_this = ⟨C_this, ℂ^f, ℂ^b, V, k, Ctx^e⟩
plan = compile_search_plan(S_this)
return [
  conj_pattern.constraint_type,
  conj_pattern.op_type,
  constantly([plan.c, plan.p]), # a zero arg function that returns the cost
  constantly(conj_step(map(t, plan.ops)))
]
```

Algorithm 4.3.1: The procedure `compile(conj_pattern, C_parent, k, Ctx^e)`

- Neg types add a unique, fresh constraint type t_i^C to the parent search configuration, same as ConjS.
- They also add a unique, fresh operation type t_i^O to the parent search configuration, same as ConjS.
- they add one constraint (with the type of t_i^O to the parent search scope).

Chapter 5

Adapting the search engine to ingraph

This chapter summarizes the steps involving the adaptation of our search engine to ingraph. We start by giving an architectural overview on ingraph, then we describe the changes made to the indexer to support estimation and evaluation and collect the resulting engine operations. We conclude with mapping relational graph algebra operations to constraints needed to translate queries into the input required by the engine.

5.1 The ingraph query engine

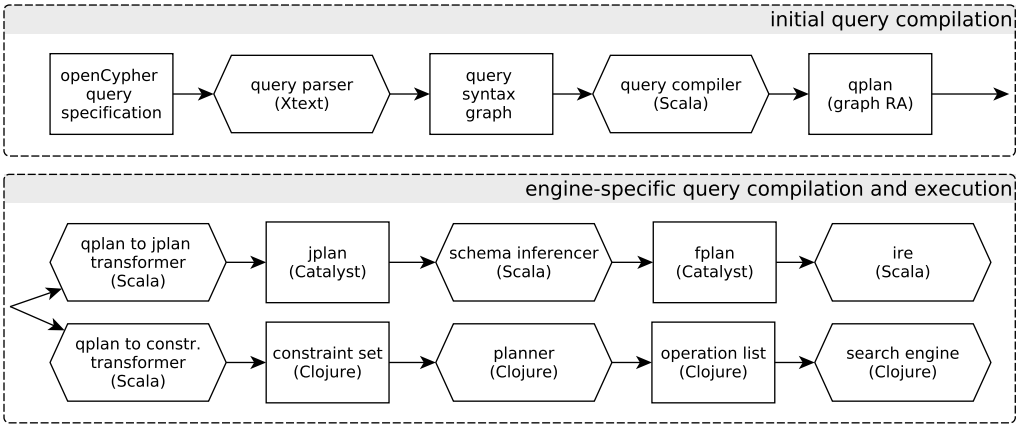


Figure 5.1: The workflow of query processing in the ingraph engine. Notation: *rectangles* represent artifacts (query specification, query plan, etc.); *hexagons* represent components (parser, compiler, etc.).

The *ingraph* project is a research prototype that aims to provide *in*cremental query evaluation of openCypher graph queries (Section 2.4.2) [16].

The workflow of the query engine and the proposed integration of the SRE components is shown in Figure 5.1. First, the query is compiled using the following steps and artifacts:

1. openCypher query specification: the query as a string.
2. query parser: an Xtext-based parser [15], using the openCypher grammar of the Slizaa project [43].

3. query syntax graph: the abstract syntax graph of the query.

4. compiler: transforms the syntax graph to a query plan.

5. qplan: a query plan formulated in graph relational algebra.

Second, the query is evaluated by either the IRE incremental engine, discussed in [54], or the SRE search-based engine, the central topic of this thesis work. The details for the compilation and evaluation steps for the incremental engine are out of scope for this work and are discussed in [51].

Efficient basic retrieval operations for graph elements is a prerequisite for search-based evaluation [64]. ingraph uses an indexer that allows efficient retrieval for vertices based on their labels, and for edges based on their type and start/end vertices.

5.2 Clo*j*ure

At the time the implementation of SRE started, the primary languages used in the ingraph project was Xtend and Java [27], with a lesser amount of Scala code. We already deprecated Xtend for any new functionality because the language has been stagnant for some time, with virtually no community and most original authors focused on developing Xtext. The author initially had preference for Scala, a statically typed language because of its rich, complex type system [10], support for functional programming and concise syntax compared to Java. However, according to early results, implementing a proof-of-concept, purely functional, type-safe, single-threaded evaluation machine proved far more challenging than reasonable, especially that the evaluation machine is one of simplest components of the engine. The author felt that Scala compile time were unsatisfactorily long for the early experimentation phase, and it hindered productivity. While Java is notably better w.r.t. compile time, the author has a strong preference against Java and he deemed a REPL-driven dynamic programming language with fast feedback loop far more suitable for prototyping a proof-of-concept implementation. Clojure [19, 18] is a LISP-dialect for the JVM, with emphasis on approachability and interactive development, so it was a natural choice for writing the core components of sre. For the interoperability layers, such the indexer or the relational tree to constraint compiler, we used Scala.

5.3 Mapping relational algebra operators to constraints

ingraph uses an RGA-based tree representation for the query, which we have to transform to constraints that we input the search planner. Table 5.1 shows the mapping between the RGA expressions and the corresponding constraints.

As many of the RGA operations have different versions depending on the included label or type predicates, we marked these with asterisks in the both the operation and the corresponding added constraint instead of creating separate rows, sparing some space. One can see that many of the relational algebra operators have overlapping constraints. This is partly because we created the constraints in such a way, that they don't provide information about e.g., how an edge should be navigated. A $\mathsf{DirectedEdge}(v, e, w)$ can correspond to either and $\uparrow^{(\mathtt{w})}_{(\mathtt{v})}[\mathsf{e}]$ or a $\downarrow^{(\mathtt{v})}_{(\mathtt{w})}[\mathsf{e}]$. Choosing an efficient navigation is up to the planner.

expression	constraint set	description
$\mathsf{O}_{(\mathtt{v})}$	$\{\ \mathsf{Vertex}(v)\ \}$	The GetVertices RGA operator corresponds to a single Vertex constraint.
$\mathsf{O}\!\rightarrow\!\mathsf{O}^{(\mathtt{w})}_{(\mathtt{v})}[\mathsf{e}]$	$\{\ \mathsf{DirectedEdge}(v,e,w)\ \}$	All navigation related operators are mapped to a DirectedEdge constraint.
$\uparrow^{(\mathtt{w})}_{(\mathtt{v})}[\mathsf{e}]\,(r)$		
$\downarrow^{(\mathtt{v})}_{(\mathtt{w})}[\mathsf{e}]\,(r)$		
$\not\equiv_P\,(r)$	$\{\mathsf{PropF}_{a\neq b}(a,b)\,\vert\,a,b\in P, a\neq b\}$	PropF is an atomic constraint wrapper for a propositional formula, which gets the variables referenced in the formula (which is in this case an inequality check) as parameters. It is atomic in the sense that formula is not further analyzed and broken up into constraints.
$R\bowtie S$	$\{\mathsf{PropF}(v_{R_n}=v_{S_n})\,\vert\,n\in sch(R)\cap sch(S)\}$	equality constraints are created for each variable pair used for joining. In the case of a natural join, variables with the same names are chosen.
$\sigma_F(r)$	$\{\ \mathsf{PropF}_{\mathsf{F}}(V_{i_F}\ldots)\ \}$	each selection operation corresponds to n-ary PropF constraint. The variables of the constraint are the variables of the corresponding formula.

Table 5.1: Mapping the relation graph algebra operators to constraints

```
(defconfig Ingraph)

(defconstraint Known [known])
(defconstraint Element [element] < Known [element])
(defconstraint Edge [edge] < Element [edge])
(defconstraint Vertex [vertex] < Element [vertex])
(defconstraint HasLabels [vertex labels] < Vertex [vertex] Known [labels])
(defconstraint HasType [edge type] < Edge [edge] Known [type])
(defconstraint Property [element key value]
  < Element [element] Known [key] Known [value])
(defconstraint DirectedEdge [source edge target] <
  Vertex [source]
  Edge [edge]
  Vertex [target])
(defconstraint GenUnaryAssertion [x cond] < Known [x] Known [cond])
(defconstraint GenBinaryAssertion [x y cond] < Known [x] Known [y] Known [cond])
(defconstraint Constant [x value] < Known [x])
```

Source 5.3.1: The set of constraints defined for ingraph

The constraints shown in tbl:constraint-mapping are implemented in the engine with our own `Config` DSL, written in Clojure. Config bears the same semantics as in the

previous chapter ($\mathcal{C}$). In this DSL, constraints after < are the implications of the one defined on the left side of the operator.

5.4 Platform operations

The search engine requires platform support in two contexts: estimation and evaluation. Estimation is required during the planning phase, as described by Chapter 4.

Estimation In the context of estimation, the current state of the underlying graph is observed to give an estimate branching factor for the search operations with the methods show in Source 5.4.1.

```
trait EstimationMethods {
  def getNumberOfVertices(): Int
  def getNumberOfEdges(): Int
  def getNumberOfLabels(): Int
  def getNumberOfTypes(): Int
  def getAverageNumberOfLabelsPerVertices(): Float
  def getNumberOfVerticesWithLabel(label: String): Int
  def getNumberOfEdgesWithType(`type`: String): Int
}
```

Source 5.4.1: Methods for branching estimation in the Indexer

Evaluation The evaluation context is not used by the planner, only the executor (second phase of the algorithm). The platform-specific methods used for navigating and accessing properties of graph elements are shown on Source 5.4.2.

```
trait Indexer {
  def edgesBySourceAndTarget(source: IngraphVertex, target: IngraphVertex)
  def edgesBySourceAndTargetAndType(source: IngraphVertex,
                                    target: IngraphVertex,
                                    `type`: String)
  def vertexById(id: Long): Option[IngraphVertex]
  def edgeById(id: Long): Option[IngraphEdge]
  def vertices(): Iterator[IngraphVertex]
  def verticesByLabel(label: String): Iterator[IngraphVertex]
  def edges(): Iterator[IngraphEdge]
  def edgesJava(): JIterator[IngraphEdge]
  def edgesByType(label: String): Iterator[IngraphEdge]
}

trait IngraphVertex extends IngraphElement {
  def edgesOutByTypeJavaIterator(key: String): JIterator[IngraphEdge]
  def edgesOutJavaIterator: JIterator[IngraphEdge]
  def edgesInByTypeJavaIterator(key: String): JIterator[IngraphEdge]
  def edgesInJavaIterator: JIterator[IngraphEdge]
}

trait IngraphEdge extends IngraphElement {
  def inverse(): IngraphEdge
  def source():IngraphVertex
  def target(): IngrapVertex
}
```

Source 5.4.2: Methods for getting elements and traversing the property graph

We closely model the platform support by our search engine operations, from which a couple are shown in Source 5.4.3.

```
(defop GetVertices [vertex] -> Vertex [vertex])
(defop
  GetVerticesByLabels [vertex labels]
  Known [labels] -> Vertex [vertex] HasLabels [vertex labels])
(defop
  GetEdges [source edge target] -> DirectedEdge [source edge target])
(defop
  GetEdgesByType [source edge target type]
  Known [type] -> DirectedEdge [source edge target]
                  HasType [edge type])
(defop
  AccessPropertyByKey [element key val]
  Element [element] Known [key] -> Property [element key val]
  :opts {:immediate true})
(defop
  ExtendOut [source edge target]
  Vertex [source] -> DirectedEdge [source edge target])
(defop
  ExtendIn [target edge source]
  Vertex [target] -> DirectedEdge [source edge target])
(defop
  ExtendOutByType [source edge target type]
  Vertex [source] Known [type] -> DirectedEdge [source edge target]
                                  HasType [edge type])
(defop
  ExtendInByType [target edge source type]
  Vertex [target] Known [type] -> DirectedEdge [source edge target]
                                  HasType [edge type])
```

Source 5.4.3: The set of operations defined for ingraph

5.5 Evaluation

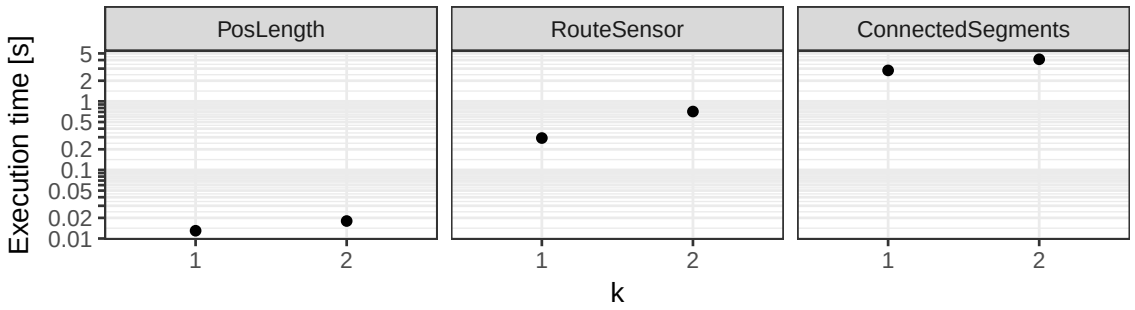


Figure 5.2: Planning times for the Train Benchmark queries.

We ran the engine on workloads specified in the Train Benchmark. We measured both the planner's performance, and the efficiency of the resulting search plans. The benchmark was performed on a MacBook Pro with 8 GB RAM and an i5 CPU. We used JDK 8 and Clojure 1.8. The source code is available the ingraph repository.[1]

Figure 5.2 shows planning times for the three queries and k values 1 and 2. Figure 5.3 shows execution times for graph of increasing size $(1, 2, \ldots, 64)$. On both figures, the x

[1] `https://github.com/FTSRG/ingraph`

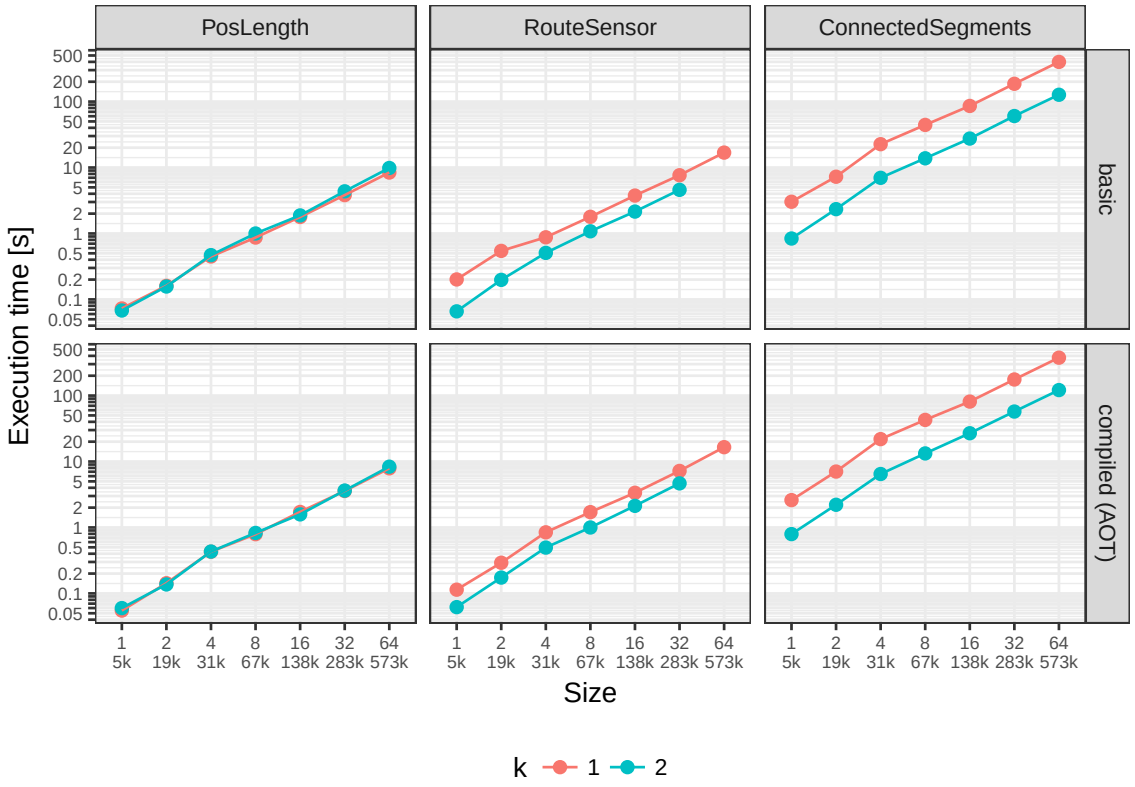


Figure 5.3: Search execution times for the Train Benchmark queries.

axis uses a $\log_2$ scale, while the y axis uses a $\log_{10}$ scale. Figure 5.2 shows that the number of variables in a pattern is a key factor in the planing time. For the most complicated pattern, ConnectedSegments, planning takex approx. 5 seconds. Figure 5.3 shows that query evaluation is the characteristics of the evaluation follows a low-order polynomial. However, evaluation time is rather slow for large graphs. We tried to mitigate the problem by using *ahead of time compilation* (AOT) compilation strategies[2], but this did not result in a significant speedup.

5.6 Conclusion

In this chapter, we presented and approach to adapt the search engine to the ingraph query engine. Our work highlighted that mapping query representations from graph relational algebra expressions to constraints is cumbersome in many cases. Based on this experience, an intermediate representation different from relational algebra could yield more a straightforward mapping. A prime candidate for this role is the *dataflow dependency graph* approach [41] used in the Cypher for Apache Spark project [38, 29], and discussed in Section 6.1.1 in more detail.

[2] `https://clojure.org/reference/compilation`

Chapter 6

Related work

In this chapter, we cover related work from two key aspects: (1) defining and evaluating queries on propery graphs, and (2) search-based graph query evaluation. We also briefly present the 2015 Transformation Tool Contest case that featured the Train Benchmark.

6.1 Queries on property graphs

A recent survey by Angles et al. [6] gives an excellent overview on property graph queries, with a particular emphasis on the difference between *graph patterns* (Sec. 3 of said paper) and *navigational queries* (Sec. 4, ibid.). Based on this classification, this thesis work touches on both of these graph query processing aspects: it mainly focuses on navigation, but also aims to incorporate pattern matching-style features, such as projection, filtering and negative conditions.

6.1.1 Query languages

While this thesis discusses openCypher queries, its approach could be adapted to other declarative property graph languages. Here, we examine the state-of-the-art property graph query languages today with a focus on their features related to search-based query processing. Many languages offer (at least partial) support for regular path queries (RPQs) [63, 6] that allow users to express complex traversals. While local search-based algorithms are a good fit for evaluating RPQs, very little research was conducted for applying about the intersection of local search algorithms and RPQs [22].

openCypher and Cypher 10

Cypher 10 [46, 5] is the next release of the Cypher language, scheduled for release in 2018. Key novelties in Cypher 10 will include support for support for multiple graphs, subqueries and relational path queries. Subqueries can be tackled efficiently using a local-search based approach, as already demonstrated by existing systems such as VIATRA [59].

Existing implementations use different approaches to represent Cypher queries. The Cypher engine in the Neo4j database uses logical plans that consist of mostly relational algebraic operators with a few graph-specific extensions, most notably `Expand` and `VarExpand` [55]. For optimization, the planner follows an approach similar to the dynamic programming algorithm of System R [49].

As part of the openCypher project, QUIL (**qu**ery **i**ntermediate **l**anguage) was proposed as a model to represent Cypher queries in a compact and portable way [41]. The idea of QUIL is to create a *dataflow dependency graph* of building blocks, each of which describes a query part using a declarative syntax similar to the constraints used by the search planner (see Section 5.3 for the mapping from graph relational algebra to constraints[1]). The dataflow dependency graph requires as few ordering between building blocks as possible, which makes this representation more "planner-friendly", i.e., it is easier to optimize than relational algebra-based representations.

The Cypher for Apache Spark project (abbrev. CAPS) [38] uses a building block-based approach inspired by QUIL. It then transforms the intermediate representation to Spark's DataFrames API and uses Spark's query planner, Catalyst for optimization [29].

PGQL

PGQL [57] is "an SQL-like query language for property graph"[2] developed by Oracle Labs. It mixes graph query features such as pattern matching for vertices, edges and paths with SQL-like capabilities such as aggregation and ordering. It also supports RPQs, multiple graphs and temporal data types.

G-CORE

G-CORE [5] is a language developed by the Linked Data Benchmark Council's [7] Query Language task force. G-CORE has many similarities to cutting-edges industrial graph query languages (including ASCII-art syntax and support for regular paths queries), but has a few distinguishing features:

1. It handles *paths as first class citizens*. According to the typical definition, a graph consists of vertices and edges, i.e., $G = (V, E)$, plus their labels and properties. In contrast, G-CORE uses the concepts of *path property graph*, defined as a set of vertices, edges and paths, i.e., $G = (V, E, P)$, with paths also having their own labels and properties.
2. It is designed to support *composability*, therefore the output of a graph query is a graph (and not a table as in Cypher, PGQL, etc.).

Up to our best knowledge, the challenges introduced by these features have not yet been investigated in the context of graph search algorithms. Also, as of 2017, G-CORE does not yet have a reference implementation of any kind. Hence, adapting search-based strategies to support (a meaningful subset of) the language with a prototype implementation demonstrating its usability and performance would yield a promising novel line of research.

Gremlin

Gremlin [44] is a functional query language supporting both imperative and declarative style queries, also allowing users to use a mix of the two manners. This way, users can

[1] It can be observed that the `given %any-node(v)` construct of QUIL corresponds to the $\mathsf{Vertex}(v)$ constraint of our approach and `where connected(v, e, w)` – with given `v`, `e` and `w` variables – to $\text{O}\!\!\rightarrow\!\!\text{O}^{(w)}_{(v)}\,[\mathsf{e}]$.

[2] `http://pgql-lang.org/`

formulate queries with the approach that best fits their needs: imperative for low-level optimizations and declarative with a runtime query planner.

GraphScript

GraphScript [39] is SAP's high-level imperative language for defining analytical workloads and graph traversals. Due to its imperative nature, the search plan is explicitly specified in the implementation, which leaves little chance for applying sophisticated optimization methods.

6.1.2 Query engines

Neo4j [31] is arguably the most popular graph database [11]. It uses a mix of search-based and relational approaches: its *"expand"* operator is supported by and efficient indexing and storage layer, while many of its operators (e.g., aggregation, filtering) are purely relational.

The SAP HANA database provides a Graph Extension [45] that is able to evaluate queries in openCypher and GraphScript.

Graphflow [23] is an *active graph database*, developed at the University of Waterloo.[3] It uses a relational engine based that calculates non-incremental queries using the Generic Join algorithm [36] and also supports incremental (active) queries with the Delta Generic Join algorithm [4].

GRAPE (Graph Rewriting and Persistence Engine) is a domain-specific language embedded to Clojure, that generates Cypher code from the query specification.

6.2 Search-based graph queries

6.2.1 Database technologies

Although the term *local search* is seldom used, similar techniques are present in the database research community.

Trinity is a prototype engine, developed by Microsoft Research [64], operating on the RDF data model [62]. It evaluates queries in a distributed in-memory execution environment. It uses a technique called *graph exploration*, which is similar to local search.

Zhao et al. [65] proposed a sophisticated query optimization method for graph queries. The optimization uses neighborhood and path analysis for speeding up the queries. Krause et al. [25] defined a SQL-based query language for graph pattern matching. For evaluating graph queries, their approach uses the SAP HANA database, including its optimization engine.

6.2.2 Model-driven technologies

FunnyQT [21] is a local search-based model query and transformation engine. Similarly to the implementation of this thesis work, FunnyQT is also built on Clojure. It does not

[3] `http://graphflow.io/`

use a planner as the search plan is determined by the order of vertex and edges symbols in the pattern specification.

The rest of this subsection discusses earlier local search-based approaches and is based on the related work section of paper [54] by Gábor Szárnyas et al.

The Fujaba [37] graph transformation tool performs local search starting from the vertex selected by the system designer and extends the matching step-by-step by neighboring vertices and edges. Fujaba fixes a single, breadth-first traversal strategy at compile-time, using simple heuristics, e.g., that navigation along an edge with an at most one multiplicity constraint precedes navigations along edges with arbitrary multiplicity.

PROGRES [48] uses a sophisticated cost model for basic operations and generates the search plan at compile-time by a greedy algorithm. GrGen.NET [17] provides a dynamic, runtime optimization engine, which uses a mix of heuristical and cost-based techniques [17].

6.3 Train Benchmark in the Transformation Tool Contest

The Train Benchmark [53] (Section 2.2) was presented as a case for contestants at the 2015 Transformation Tool Contest by Gábor Szárnyas et al. [52]. The case authors provided a Viatra (then called EMF-IncQuery) solution with an incremental and a search-based implementation. Contestants provided solutions with various model transformation tools in the model-driven engineering community (ATL, FunnyQT, NMF, and SIGMA), many of which used search-based algorithms and tackled the problem efficiently.

Chapter 7

Summary and future work

In this thesis, we investigated the applicability of search-based pattern matching on openCypher property graph queries. In this chapter, we first summarize our theoretical contributions and practical accomplishments, then outline future research directions.

Theoretical contributions

- We studied the dynamic-programming based model-sensitive planning algorithm by G. Varró et al. [61]. We identified adaptation challenges that prevent straightforward applicability of this algorithm for property graph queries (Section 3.1–3.1).
- We suggested improvements to the original algorithm and presented an example which showcases the superiority of our approach (Section 3.3).
- We extended the set of *constraints and operations* used to capture query semantics and generate high-quality search plans. We incorporated these changes to the planner algorithm to support property graph queries (Chapter 4).
- We defined a mapping from *relational graph algebra* to the extended set of constraints (Section 5.4–5.3).

Practical accomplishments

- Implemented the planner algorithm in Clojure (Section 5.2).
- Conducted performance experiments to assess the performance of the planning and evaluation on three queries of the Train Benchmark framework [53] (Section 5.5).

We believe that this work opens up interesting future research directions and potential collaboration between the graph database and model-driven engineering research communities. In particular, we propose the following research questions for future work:

- How to mix relational operators, such as *left outer join aggregation*, *union*, *sorting* and *limiting*, with search-based pattern matching?[1]

[1]Some of these operators, such as aggregation, require a data model that supports *multiset semantics*, while others require *list semantics*. Graph search algorithms typically operate on sets, so combining them with *relational operators that use multiset or list semantics* is not yet investigated in depth.

- How to support nested data structures of the data model, such as *lists* and *maps*, along with nesting and unnesting constructs of the query language[2] using search-based query engines?
- How to create an efficient query plan for regular path queries (RPQs) for search-based evaluation?

[2]For example, Cypher's `collect()` function and `UNWIND` clause.

Acknowledgements

I would like to thank my supervisors Gábor Szárnyas and Márton Búr for their overwhelming support. Without Gábor's continuous help on core topics such as ingraph, relational algebra, the Train Benchmark, property graphs, graph databases; his point-on advices on related material and present developments in the graph community; and the tremendous effort that he put into reviewing and correcting my writing, the resulting work would be uncomparable to this one. I would like to thank Márton for his patience when giving first-hand lessons on the dynamic search algorithm, and that his help was always available when I was stuck. I would like to express my gratitude towards their enduring, faithful guidance in the last half year.

I would like to thank all other colleagues in the Fault Tolerant Systems Research Group who provided help or thoughtful remarks. Namely, I thank Gábor Bergmann and Dániel Varró for their help on constraint satisfaction problems, VIATRA and related fields, Bálint Hegyi for his LaTeX template, Oszkár Semeráth, Kristóf Marussy for their valuable observations and suggestions.

Furthermore I would like to thank my family; their encouragement and support to pursue university; their consolation and reassurance when I needed it; and for the faith they had in me all these years.

Bibliography

[1] About AgensGraph. http://bitnine.net/solutions/agensgraph/.

[2] Eclipse Modeling Framework – Interview with Ed Merks. https://jaxenter.com/eclipse-modeling-framework-interview-with-ed-merks-100007.html, 2010.

[3] Graph Processing with SAP HANA 2. https://blogs.sap.com/2016/12/01/graph-processing-with-sap-hana-2/, 2016.

[4] Khaled Ammar, Frank McSherry, and Semih Salihoglu. Delta generic join. Technical report, University of Waterloo, 2017. https://cs.uwaterloo.ca/~ssalihog/papers/deltagj.pdf.

[5] R. Angles, M. Arenas, P. Barceló, P. Boncz, G. H. L. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. Sequeda, O. van Rest, and H. Voigt. G-CORE: A Core for Future Graph Query Languages. *ArXiv e-prints*, December 2017. arXiv:1712.01550.

[6] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5):68:1–68:40, September 2017. ISSN 0360-0300. DOI: 10.1145/3104031.

[7] Peter A. Boncz. LDBC: benchmarks for graph and RDF data management. In *IDEAS*, pages 1–2. ACM, 2013. DOI: 10.1145/2513591.2527070.

[8] Márton Búr. A general purpose local search-based pattern matching framework. Master's thesis, Budapest University of Technology and Economics, 2015.

[9] Márton Búr, Zoltán Ujhelyi, Ákos Horváth, and Dániel Varró. Local search-based pattern matching features in EMF-IncQuery. In *ICGT*, volume 9151 of *LNCS*, pages 275–282. Springer, 2015. DOI: 10.1007/978-3-319-21145-9_18.

[10] Paul Chiusano and Rnar Bjarnason. *Functional Programming in Scala*. Manning Publications, Greenwich, CT, USA, 1st edition, 2014. ISBN 9781617290657.

[11] DB-Engines. Ranking of graph DBMS. https://db-engines.com/en/ranking/graph+dbms, 2017.

[12] François-Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *IJCAI, Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, 1995.

[13] Lisa Ehrlinger and Wolfram Wöß. Towards a definition of knowledge graphs. In *SEMANTiCS (Posters, Demos, SuCCESS)*, 2016.

[14] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley-Longman, 3rd edition, 2000. ISBN 978-0-8053-1755-8.

[15] Moritz Eysholdt and Heiko Behrens. Xtext: Implement your language faster than the quick and dirty way. In *SIGPLAN, SPLASH/OOPSLA*, pages 307–309, 2010. DOI: 10.1145/1869542.1869625.

[16] Fault Tolerant Systems Research Group. ingraph. docs.inf.mit.bme.hu/ingraph/, 2017.

[17] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam Szalkowski. GrGen: A fast SPO-based graph rewriting tool. In *ICGT*, pages 383–397, 2006. DOI: 10.1007/11841883_27.

[18] Stuart Halloway. *Programming Clojure*. Pragmatic Bookshelf, 1st edition, 2009. ISBN 1934356336, 9781934356333.

[19] Rich Hickey. The Clojure programming language. In *DLS*. ACM, 2008. ISBN 978-1-60558-270-2. DOI: 10.1145/1408681.1408682.

[20] Jürgen Hölsch and Michael Grossniklaus. An algebra and equivalences to transform graph patterns in Neo4j. In *GraphQ at EDBT/ICDT*, 2016. URL http://ceur-ws.org/Vol-1558/paper24.pdf.

[21] Tassilo Horn. Graph pattern matching as an embedded Clojure DSL. In *ICGT*, volume 9151 of *Lecture Notes in Computer Science*, pages 189–204. Springer, 2015. DOI: 10.1007/978-3-319-21145-9_12.

[22] Māris Jukšs, Clark Verbrugge, Maged Elaasar, and Hans Vangheluwe. Scope in model transformations. *Software & Systems Modeling*, Aug 2016. ISSN 1619-1374. URL https://doi.org/10.1007/s10270-016-0555-8.

[23] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An active graph database. In *SIGMOD*, pages 1695–1698. ACM, 2017. DOI: 10.1145/3035918.3056445.

[24] Donald Kossmann and Konrad Stocker. Iterative dynamic programming: A new class of query optimization algorithms. *ACM Trans. on Database Systems*, 25:2000, 1998.

[25] Christian Krause, Daniel Johannsen, Radwan Deeb, Kai-Uwe Sattler, David Knacker, and Anton Niadzelka. An SQL-based query language and engine for graph pattern matching. In *ICGT*, pages 153–169, 2016. DOI: 10.1007/978-3-319-40530-8_10.

[26] M. Lin, Q. Ye, and Y. Ye. Graph theory based mobile network insight analysis framework. In *IEEE 7th Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON)*, pages 1–7, Oct 2016. DOI: 10.1109/UEMCON.2016.7777888.

[27] József Marton, Gábor Szárnyas, and Márton Búr. Model-driven engineering of an opencypher engine: Using graph queries to compile graph queries. In *SDL*, pages 80–98, 2017. DOI: 10.1007/978-3-319-68015-6_6.

[28] József Marton, Gábor Szárnyas, and Dániel Varró. Formalising openCypher graph queries in relational algebra. In *ADBIS*, pages 182–196, 2017. DOI: 10.1007/978-3-319-66917-5_13.

[29] Mats. CAPS: Cypher for Apache Spark. https://github.com/opencypher/cypher-for-apache-spark, 2017. Second openCypher Implementers Meeting (OCIM2), London.

[30] Murray Newlands. 6 Disruptive Trends In Technology. https://www.forbes.com/sites/mnewlands/2016/08/31/6-disruptive-trends-in-technology-for-2017/#6cf292b528ec, 2016.

[31] Neo4j. Neo4j DBMS. https://www.neo4j.org/, 2017.

[32] Neo4j. Developer Manual: Cypher. https://neo4j.com/docs/developer-manual/current/cypher/, 2017.

[33] Neo4j. Cypher Technology Compatibility Kit. https://github.com/opencypher/openCypher, 2017.

[34] Neo4j. openCypher project. http://www.opencypher.org/, 2017.

[35] Neo4j contributors. APOC User Guide. https://neo4j-contrib.github.io/neo4j-apoc-procedures/#_overview_of_apoc_procedures_functions, 2017.

[36] Hung Q. Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Record*, 42(4):5–16, 2013. DOI: 10.1145/2590989.2590991.

[37] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In *ICSE*, pages 742–745. ACM, 2000. ISBN 1-58113-206-9. DOI: 10.1145/337180.337620.

[38] openCypher. CAPS: Cypher for Apache Spark. https://github.com/opencypher/cypher-for-apache-spark, 2017.

[39] Marcus Paradies, Cornelia Kinder, Jan Bross, Thomas Fischer, Romans Kasperovics, and Hinnerk Gildhoff. GraphScript: implementing complex graph algorithms in SAP HANA. In *DBPL*, pages 13:1–13:4. ACM, 2017. DOI: 10.1145/3122831.3122841.

[40] Heiko Paulheim. Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic Web*, 8(3):489–508, 2017. DOI: 10.3233/SW-160218.

[41] Stefan Plantikow. QUIL: Intermediary representation for Cypher. https://s3.amazonaws.com/artifacts.opencypher.org/website/ocim1/slides/12-00+-+QUIL.pdf, 2017. First openCypher Implementers Meeting (OCIM1), Walldorf.

[42] Serguei Popov. The Tangle, 2017. https://iota.org/IOTA_Whitepaper.pdf.

[43] Slizza project. Xtext based parser/editor for the Cypher query language. https://github.com/slizaa/slizaa-opencypher-xtext/, 2017.

[44] Marko A. Rodriguez. The Gremlin graph traversal machine and language (invited talk). In *DBPL*, pages 1–10. ACM, 2015. ISBN 978-1-4503-3902-5. DOI: 10.1145/2815072.2815073.

[45] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. The graph story of the SAP HANA database. In *BTW*, volume 214, pages 403–420. GI, 2013. URL http://www.btw-2013.de/proceedings/The%20Graph%20Story%20of%20the%20SAP%20HANA%20Database.pdf.

[46] Mats Rydberg. openCypher versioning. https://s3.amazonaws.com/artifacts.opencypher.org/website/ocig5/openCypher+Versioning.pdf, 2017. Fifth openCypher Implementers Group Meeting (OCIG5).

[47] Ashish Sabharwal and Bart Selman. S. Russell, P. Norvig, Artificial intelligence: A modern approach, Third edition. *Artif. Intell.*, 175(5-6):935–937, 2011. DOI: 10.1016/j.artint.2011.01.005. Book review.

[48] A. Schürr, A. J. Winter, and A. Zündorf. Handbook of graph grammars and computing by graph transformation. pages 487–550. World Scientific Publishing Co., Inc., 1999. ISBN 981-02-4020-1. URL http://dl.acm.org/citation.cfm?id=328523.328617.

[49] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34. ACM, 1979. DOI: 10.1145/582095.582099.

[50] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009. ISBN 0321331885.

[51] Gábor Szárnyas. Incremental View Maintenance for Property Graph Queries. *ArXiv e-prints*, December 2017. arXiv:1712.04108.

[52] Gábor Szárnyas, Oszkár Semeráth, István Ráth, and Dániel Varró. The TTC 2015 Train Benchmark case for incremental model validation. In *TTC*, pages 129–141, 2015. URL http://ceur-ws.org/Vol-1524/paper2.pdf.

[53] Gábor Szárnyas, Benedek Izsó, István Ráth, and Dániel Varró. The Train Benchmark: cross-technology performance evaluation of continuous model queries. *Software & Systems Modeling*, 2017. ISSN 1619-1374. DOI: 10.1007/s10270-016-0571-8.

[54] Gábor Szárnyas, János Maginecz, and Dániel Varró. Evaluation of optimization strategies for incremental graph queries. *Periodica Polytechnica Electrical Engineering and Computer Science*, 61(2):175–192, 2017. ISSN 2064-5279. DOI: 10.3311/PPee.9769.

[55] Andrés Taylor. Neo4j Cypher implementation. https://s3.amazonaws.com/artifacts.opencypher.org/website/ocim1/slides/11-20+-+Neo4j+Cypher+implementation.pdf, 2017. First openCypher Implementers Meeting (OCIM1), Walldorf.

[56] Jeffrey D. Ullman, Hector Garcia-Molina, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001. ISBN 0130319953.

[57] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. PGQL: a property graph query language. In Peter A. Boncz and Josep-Lluís Larriba-Pey, editors, *GRADES at SIGMOD*. ACM, 2016. ISBN 978-1-4503-4780-8. DOI: 10.1145/2960414.2960421.

[58] Moshe Y. Vardi. Constraint Satisfaction and Database Theory: A Tutorial. In *PODS*, pages 76–85. ACM, 2000. ISBN 1-58113-214-X. DOI: 10.1145/335168.335209.

[59] Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Software & Systems Modeling*, 15(3): 609–629, 2016. DOI: 10.1007/s10270-016-0530-4.

[60] Gergely Varró, Anthony Anjorin, and Andy Schürr. Unification of compiled and interpreter-based pattern matching techniques. In *ECMFA*, pages 368–383. Springer, 2012. ISBN 978-3-642-31490-2. DOI: 10.1007/978-3-642-31491-9_28.

[61] Gergely Varró, Frederik Deckwerth, Martin Wieber, and Andy Schürr. An algorithm for generating model-sensitive search plans for pattern matching on EMF models. *Software & Systems Modeling*, 14(2):597–621, May 2015. ISSN 1619-1374. DOI: 10.1007/s10270-013-0372-2.

[62] W3C. Resource Description Framework. https://www.w3.org/RDF/, 2014.

[63] Peter T. Wood. Query languages for graph databases. *SIGMOD Record*, 41(1):50–60, 2012. DOI: 10.1145/2206869.2206879.

[64] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A distributed graph engine for web scale RDF data. *PVLDB*, 6(4):265–276, 2013. URL http://www.vldb.org/pvldb/vol6/p265-zeng.pdf.

[65] Peixiang Zhao and Jiawei Han. On graph query optimization in large networks. *PVLDB*, 3(1):340–351, 2010.

Appendix

A.1 Generalized cost function

The monotonic cost function has been originally defined as

$$c_n = \sum_{i=1}^{n} \prod_{j=1}^{i} w_j$$

We've seen that this function can be computed iteratively by

$$c_0 = 0, \quad p_0 = 1, \quad c_i = c_{i-1} + p_i, \quad p_i = p_{n-1} w_i$$

where w is the operation's weight. One can also see that it can be sliced up into multiple iteratively computable parts:

$$\begin{aligned} \sum_{i=1}^{n} \prod_{j=1}^{i} w_j &= \sum_{i=1}^{k} \prod_{j=1}^{i} w_j + \sum_{i=k+1}^{n} \prod_{j=1}^{i} w_j \\ &= \sum_{i=1}^{k} \prod_{j=1}^{i} w_j + \prod_{j=1}^{k} w_j \sum_{i=k+1}^{n} \prod_{j=k+1}^{i} w_j \end{aligned}$$

which gives the recursive definition:

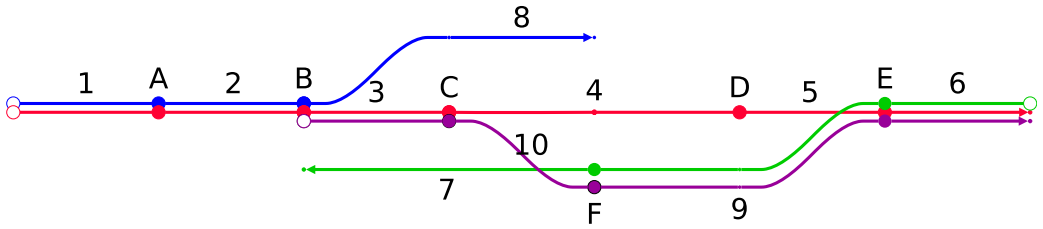
$$\begin{gathered} i, k, n \in \mathbb{N}, \quad i \leq k < n \\ c_i^i = 0, \quad p_i^i = 1, \quad p_{i-1}^i = c_{i-1}^i = w_i \\ c_i^n = c_i^k + p_i^k c_k^n, \quad p_i^n = p_i^k p_k^n \end{gathered}$$

Now, we can introduce arbitrarily partitioned composite operations, which will prove useful for subqueries:

$$c_i = c_{i-1} + p_i \sigma_i, \quad p_i = p_{i-1} w_i$$

We call w and σ the operation's weight and cost respectively.

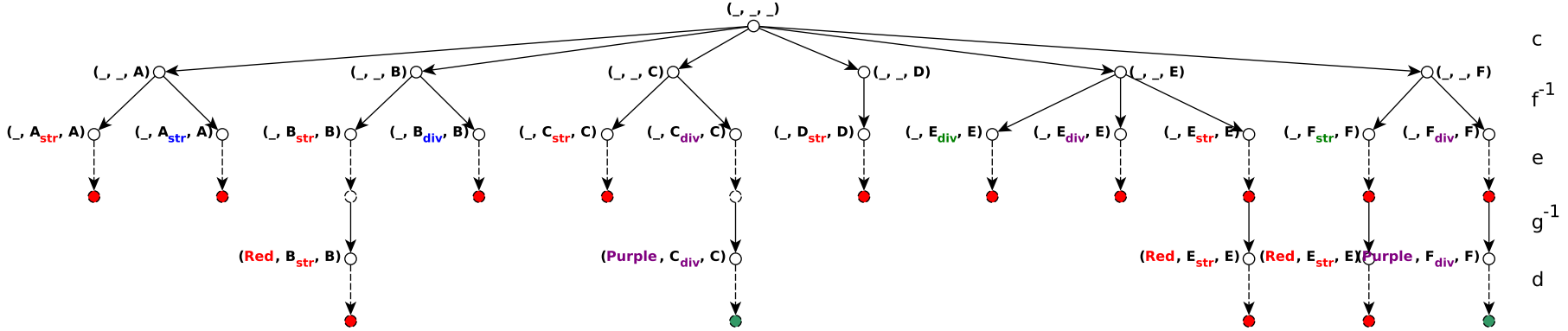
A.2 ActiveRoute evaluation with costs



(a) Routes in the railway example (repeat of Figure 2.1d to make the pattern matching example easier to follow).

[a] $s_{\mathsf{Route}}(r)$, [b] $s_{\mathsf{SwitchPosition}}(swP)$, [c] $s_{\mathsf{Switch}}(sw)$, [d] $r.\mathsf{active} = \mathsf{true}$, [e] $swP.\mathsf{position} \neq sw.\mathsf{currentPosition}$, [f] $e_{\mathsf{target}}(swP, sw)$, [g] $e_{\mathsf{follows}}(r, swP)$

(b) Operation costs (based on Figure 3.1b).



(c) The complete pattern matching state space of ActiveRoute: the $cf^{-}1eg^{-}1d$ task sequence.

Figure A.2.1: Pattern matching sequence of pattern ActiveRoute.

A.3 Source code of the traversal

The Clojure source code of the execution engine is included hereby.

```
(ns sre.execution.executor
  (:require [clojure
             [set :refer :all]
             [zip :as z]]
             [clojure.algo.generic.functor :refer :all]
             [sre.core :refer :all]
             [sre.plan.task :refer [ISearch search]]
             [clojure.pprint :as pprint])
  (:refer-clojure :exclude [name])
  (:import [clojure.lang LazySeq PersistentList]))

(defrecord ZipperNode [variables ctx ^PersistentList left])
(def search-tree-zipper
  (partial z/zipper
           (fn [^ZipperNode node] (some? (:left node)))
           (fn [^ZipperNode node]
             (let [[first & rest] (:left node)
                   heads (search (:type first) (:bindings first)
                                 (:variables node) (:ctx node))
                   children (map #(->ZipperNode %1 (:ctx node) rest) heads)]
               (if-not (empty? children) children)))        ; convert empty list to nil or
               ↪  else strange things happen :(
           (fn [^ZipperNode _ ^LazySeq c] c)))
(def dft (partial iterate #(p :next (z/next %))))
(def take-until-end (take-while (complement z/end?)))
(def filter-leaf (filter #(or (false? (z/branch? %))
                              (empty? (z/children %)))))
(def filter-complete (filter #(-> % z/node :left nil?)))
(def map-variables (map #(-> % z/node :variables)))
(defn remap-keys [m key-map] (reduce-kv (fn [a k v] (assoc a (key-map k) v)) {} m))
(defn rebind-variables [variables bindings key-map]
  (-> variables (select-keys bindings) (remap-keys key-map)))
(defn conj-step-search [this bindings variables ctx]
  (let [{subtasks :subtasks outer-vars :outer-vars inner-vars :inner-vars} this
        key-map (zipmap bindings outer-vars)
        variables (rebind-variables variables bindings key-map)
        find-matches (fn [x] (eduction take-until-end
                                       filter-leaf
                                       filter-complete
                                       map-variables
                                       (map #(p :rebind-vars (rebind-variables % outer-vars
                                       ↪  (map-invert key-map))))
                                       (dft x)))]
(find-matches (search-tree-zipper (->ZipperNode variables ctx subtasks)))))
```