ADAPTING GRAPH SIMULATION ALGORITHMS FOR GRAPH DATABASE QUERY PROCESSING

by

ARAVIND KALIMURTHY

(Under the Direction of John A Miller)

ABSTRACT

With data exponentially increasing in almost all fields in today's world, there comes the necessity of handling and querying the data efficiently. Recently, many graph databases have emerged to handle big data. This is traditionally done using regular query handling processes and subgraph isomorphism. In this research, we introduce edge labels into graph simulation algorithms, so that we can quickly query and filter graphs not only using the vertex labels but also using edge labels. We have also accommodated cardinality restrictions for edge labeled graphs that improves the quality of the search results. Query processing involves taking a query graph and trying to find its pattern in a larger data graph. We have added the capability to query the graph database using wildcards, regular expressions, and variables. This is done by replacing, in a query graph, one or more strings in edge labels with wildcards, regular expressions or variables. Experiments are done on very large graphs with up to 30 million edges.

INDEX WORDS: Graph Database, Graph Simulation Algorithms, Edge Label, Cardinality, Query Processing, Wildcard, Regular Expression
Adapting Graph Simulation Algorithms for Graph Database Query Processing

by

Aravind Kalimurthy

Major Professor: John A Miller

Committee: Lakshmish Ramaswamy
Liming cai

Electronic Version Approved:

Suzanne Barbour
Dean of the Graduate School
The University of Georgia
December 2017
DEDICATION

To my family and professor for their love, support, motivation and belief.
I would like to thank my major professor Dr. John A Miller for guiding me throughout my masters. He is patient, supportive, and his guidance has always helped me. Thank you for educating me.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgements</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>viii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>x</td>
</tr>
<tr>
<td>Chapter</td>
<td></td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Background</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Graphs</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Subgraph Pattern Matching</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Types of Subgraph Pattern Matching</td>
<td>5</td>
</tr>
<tr>
<td>2.4 Edge Label</td>
<td>10</td>
</tr>
<tr>
<td>2.5 Graph Representation</td>
<td>12</td>
</tr>
<tr>
<td>2.6 Cardinality Restriction</td>
<td>12</td>
</tr>
<tr>
<td>2.7 Database Systems</td>
<td>13</td>
</tr>
<tr>
<td>2.8 Query Processing</td>
<td>14</td>
</tr>
<tr>
<td>2.9 Query Languages</td>
<td>15</td>
</tr>
<tr>
<td>3 Graph Analytics With Cardinality and Edge Labels</td>
<td>17</td>
</tr>
<tr>
<td>3.1 CAR-Graph Simulation with Edge Labels</td>
<td>17</td>
</tr>
<tr>
<td>3.2 CAR-Dual Simulation with Edge Labels</td>
<td>19</td>
</tr>
<tr>
<td>3.3 CAR-Strict Simulation with Edge Labels</td>
<td>21</td>
</tr>
<tr>
<td>3.4 CAR-Tight Simulation with Edge Labels</td>
<td>22</td>
</tr>
</tbody>
</table>
4 Query Pattern Processing .............................................. 25
   4.1 WildCards ......................................................... 25
   4.2 Regular Expressions .......................................... 27
5 Experiment Results ...................................................... 30
   5.1 Precision of Simulation Algorithms ....................... 31
   5.2 Mutable vs. Immutable Data-structures ..................... 34
   5.3 Comparison of Graphs with Edge Labeled Graphs ......... 35
   5.4 Comparison of Graphs with Cardinality Graphs ........... 37
   5.5 Comparison of Cardinality Graphs with Edge Labeled Cardinality Graphs ................................................ 38
   5.6 Simulation Performance Comparisons ....................... 40
   5.7 Wildcard vs. RegEx vs. None .................................. 40
6 Conclusions .............................................................. 43
7 Future Work .............................................................. 44
Bibliography .............................................................. 45
Appendix: dk.brics.automaton.regexp  ................................ 48
## List of Figures

2.1 A Sample Graph .................................................. 4
2.2 Subgraph Isomorphism ............................................. 6
2.3 Graph Simulation ................................................... 7
2.4 Dual Simulation .................................................... 8
2.5 Ball Creation ....................................................... 9
2.6 Query Graph with String Labels ................................. 11
2.7 Cardinality restriction ............................................. 13
3.1 Data for Graph Sim and Dual Sim .............................. 18
3.2 Data for Strict Sim and Tight Sim .............................. 22
4.1 Multiple Wildcard Blocks ........................................ 26
4.2 Multiple Wildcard matching flow ............................... 26
4.3 Regex matching ................................................... 28
5.1 Precision Random Graph .......................................... 32
5.2 Precision for Power Graph ...................................... 32
5.3 Precision for Real-world Graph ................................. 33
5.4 Graph Mutable vs. Immutable ................................. 34
5.5 Dual Mutable vs. Immutable .................................... 35
5.6 GraphSim vs. MGraphSim ....................................... 36
5.7 DualSim vs. MDualSim ........................................... 36
5.8 Graph vs. GraphCAR ............................................. 37
5.9 Dual vs. DualCAR ................................................ 38
5.10 GraphCAR vs. MGraphCAR .................................. 39
5.11 DualCAR vs. MDualCAR ........................................ 39
5.12 Comparison of Simulation Algorithms ................................. 40
5.13 MGraphSim Wc vs. Rx vs. None ........................................ 41
5.14 MDualSim Wc vs. Rx vs. None ........................................ 42
LIST OF TABLES

3.1 Graph Simulation Matches ............................ 19
3.2 Dual Simulation Matches ............................. 21
3.3 Strict Simulation Matches ............................ 23
3.4 Tight Simulation Matches ............................. 24
CHAPTER 1

INTRODUCTION

We know that the data is exponentially increasing in almost all fields in today’s world. For example, Facebook has around 2061 million users, YouTube has 1500 million users, Instagram has 700 million users and Twitter has 328 million users\(^1\). As the data is increasing exponentially there comes the necessity to handle this large data easily and efficiently. This can be done by graphs due to their capability to express multiple classes of problems. Graph Databases have gained popularity due to their fundamental graph data structure and their model of data with their relationships which seems to be better than relational databases for storing and manipulating the given data. It uses vertices to store data and the connected edges define a relation between the vertices. Each vertex in the database has a direct pointer towards the adjacent vertex and it can access any information stored in the adjacent vertex. Graph databases are been used to study various problems in all different fields. They have been used in many real-world application domains. Some of the most common application graphs have are Facebook, Amazon, Twitter, YouTube, LinkedIn, Networking, Transportation control, Real-time Recommendation engine, Fraud detection, Plagiarism detection, Identity and access management, Web search engines, and the study of molecular biology like protein to protein molecular bonds.

In this research, we focus on query processing where the database consists of a single large graph and a small query graph. The aim would be to find the all the subgraphs that are similar to the small graph in the large data graph. However, this problem is related to subgraph isomorphism and is NP-hard [2]. Researchers have come up with heuristics for this problem that run in polynomial

time, which we will study in detail in upcoming chapters.

The outline of this thesis is as follows: In chapter 2, we discuss the required background for this research, including the description of graphs and the classification of pattern matching problem along with several types of subgraph pattern matching algorithms. In chapter 3, we discuss our versions of simulation algorithms with edge labels and cardinality restrictions. In chapter 4, we discuss our implementation of query processing for simulation algorithms using wildcards and regular expression. In chapter 5, we see the experimental results and impact of our pruning techniques in synthesized and real-world graphs. In chapter 6, we conclude by summarizing the outcomes from this research. The last chapter talks about possible future work in this research area of graph simulation algorithms.
CHAPTER 2

BACKGROUND

In this background section, we discuss the basics of graphs and the related graph structure we use in this research before describing the graph simulation algorithms which tackle subgraph pattern matching.

2.1 Graphs

Graphs are the data structure used to model various information inform of vertices and represent the relationship between the vertices in terms of edges connecting between them. We have taken into account only directed graph, with vertex labels and edge labels, which can be represented as $G(V, E, L, l_v, l_e)$ extension from $G(V, E, l)$ [14] where

- $V =$ set of Vertices
- $E \subseteq \{(u, v) \mid u \in V, v \in V \text{ and } u \neq v\}$ set of directed edges
- $L =$ set of labels
- $l_v = V \rightarrow L$ function mapping vertices to labels
- $l_e = E \rightarrow L$ function mapping edges to labels

Vertices are usually the entities in the graph database. They contain the properties or sometimes the actual meta data. Every vertex is connected with relationships or edges and is directed from one vertex to another. Edges also contain properties or data. Every vertex or edge has a label used to represent particular information. Vertices and edges can have multiple information or multiple labels but we limit ourselves to a single label for this research.
A sample graph is shown in Figure 2.1, where each vertex or edge has a label.

Figure 2.1: A Sample Graph
2.2 Subgraph Pattern Matching

The main purpose of graph pattern matching is to find the accurate matches among the vertices and edges of a smaller query graph in an existing large data graph. The problem of subgraph pattern matching can be defined as finding a query pattern \( Q(V_q, E_q, L_q, l_{vq}, l_{eq}) \) in a data graph \( G(V, E, L, l_v, l_e) \) where \( V_q \) is the set of vertices in the query graph, \( E_q \) is the set of edges in the query graph, \( L_q \) is set of labels on the query graph, \( l_{vq} \) is the labeling function which maps vertices to labels \( V_q \), \( l_{eq} \) is the labeling function which maps edges to labels on \( E_q \). The goal is to find all the subgraphs from \( G \) that matches the query pattern \( Q \). Therefore, \( G'(V', E', L', l'_v, l'_e) \) is a subgraph of \( G \) iff

1. \( V' \subseteq V \)
2. \( E' \subseteq E \)
3. \( L' \subseteq L \)
4. \( \forall u \in V' : l'_v(u) = l_v(u) \)
5. \( \forall u \in E' : l'_e(u) = l_e(u) \)

We here assume that the query graph is connected because the result of pattern matching for a disconnected graph is a union of all its results.

2.3 Types of SubGraph Pattern Matching

In this section, we will review different types of pattern matching algorithms and how they match query patterns. Subgraph isomorphism is the basic model for the pattern matching problem. However, this approach is NP-hard. Subgraph isomorphism has many real-world applications and much research has been done to find all subgraph patterns efficiently. To reduce the strictness of subgraph isomorphism several other models have been developed that match the query in polynomial time. Some of these algorithms are Graph Simulation [10] [12], Dual Simulation [12], Strong Simulation [12, 13], Strict Simulation [6] and the most recent being Tight Simulation [6]. In this section,
we will look at Subgraph Isomorphism, Graph Simulation, Dual Simulation, Strict Simulation and Tight Simulation.

2.3.1 Subgraph Isomorphism

The subgraph isomorphism problem is to find the precise matches between two given graphs, a large data graph, and a small query graph. It extracts all the characteristics of the query graph from data graph to form the resulting subgraph. This problem is said to be an exact matching problem and the problem is NP-hard. Ullmann’s algorithm [18], VF2 algorithm [3], TurboIso [9], DualIso [17] are some of the most popular subgraph isomorphism algorithms.

Figure 2.2: Subgraph Isomorphism

Figure 2.2 shows an example for subgraph isomorphism for the given query graph. The result match set is represented in colored vertices. We get two subgraph isomorphism match results since the vertex id 0 from the query graph matches with vertex id 0 and 5 in the data graph.
2.3.2 **Graph Simulation**

A new model [12] was developed to reduce the complexity and was based on relaxing the constraints on the match set. Let us consider vertices of the query graph \( u \in V_Q \) and vertices of the data graph \( v \in V_G \), Graph simulation requires that the following constraints be satisfied, regarding the mapping of vertices in \( V_Q \) to vertices in \( V' \subseteq V_G \):

1. Each \( u \in V_Q \) must match at least one vertex \( v \in V' \)
2. Labels of matching vertices \( u \) and \( v \) should match, i.e. \( l(u) = l(v) \)
3. Child labels of vertex \( u \in V_Q \) must be a subset of child labels of matching vertex \( v \in V' \)

Figure 2.3: Graph Simulation

Figure 2.3 shows an example for Graph simulation where the vertices of result match set are colored. Since vertex id 7 does not have a child with label P as in the given query graph, it does not match by Graph simulation.

2.3.3 **Dual Simulation**

Dual simulation [12] is an extended version of Graph simulation adding an additional pruning condition. It not only checks for the data graph vertices for children match but also checks the vertices for their parent match. This additional condition could make the algorithm faster, since
vertices may be pruned away more quickly. Dual simulation requires that the following constraints be satisfied, regarding the mapping of vertices in $V_Q$ to vertices in $V' \subseteq V_G$:

1. Each $u \in V_Q$ must match at least one vertex in $v \in V'$
2. Labels of matching vertices $u$ and $v$ should match, i.e. $l(u) = l(v)$
3. Child labels of vertex $u \in V_Q$ must be a subset of child labels of matching vertex $v \in V'$
4. Parent labels of vertex $u \in V_Q$ must be a subset of parent labels of matching vertex $v \in V'$

Figure 2.4: Dual Simulation

Figure 2.4 shows an example for Dual simulation where the vertices of result match set are colored. Since vertex id’s 6 and 8 do not have a parent with label B as in the given query graph, it does not match by Dual simulation.

2.3.4 **Strict Simulation**

Strict simulation was introduced by Fard et al. [7] and is an extension of Strong simulation introduced by Ma et al. [13]. Here they add a locality property to dual simulation. They introduce the
concept of a ball to define locality.

A ball $b$ in $G(V, E)$, denoted by $G'(v, r)$ is a subgraph of $G$ such that it contains all the vertices that are not more distant than a given radius $r$ from a center $v \in V$ and the radius is acquired from the query graph which is the diameter of the query graph. A size of a ball is the number of vertices it has. In strict simulation, balls are created from the dual match set rather than the original data graph which reduces the solution.

Figure 2.5 shows the creation of balls of radius 1 for tight simulation. Consider vertex id 1 with label P from query graph, its match set from the data graph is1, 6, 8. A ball graph for all the match set is said to have all the vertices of radius 1. From figure 2.5 we can see that the vertex id 1 has three initial balls i.e.

$$b(1) = (0, 1, 2, 3, 4, 5), (2, 3, 6, 7), (3, 4, 7, 8).$$
2.3.5 Tight Simulation

Tight simulation [6] is an advanced modified version of Strict simulation. A further condition is added to reduce the number of balls. First, before applying dual simulation on $G$ to find the dual simulation match set, they find the vertex candidate in $Q$ which decreases the size of the balls and also reduces the number of balls. In addition to this, the radius of the ball is equal to the radius of the query graph, not the diameter, of the query graph as in strict simulation.

2.4 Edge Label

In the previous graph simulation algorithms, edge labels were not considered and were ignored. All data and query graphs were of the type directed graph but did not have edge labels. We need edge labels to know more about the relationship between two vertices. An example application that uses edge labels is Wordnet3, which uses 9 different kinds of edge labels. So we introduce edge labels to all the simulation algorithms and hence further prune the match set by removing the corresponding vertices and edges whose edge labels do not match. Here similar to vertex label, edge label can be Integers, Doubles or Strings based on the requirement. Figure 2.6 shows query graph with vertex and edge labels of type strings.

$$l_e : V \rightarrow L_e \text{ (Edge labeling function)}$$
Figure 2.6: Query Graph with String Labels
2.5 **Graph Representation**

It is important to know how the graphs are represented in the memory. In our implementation, vertices of the graphs are represented in memory as the indices of an array and each vertex is mapped to its labels using a separate array called `label`. The children of a vertex `v` are represented by an adjacent vertex set `ch(v)`. Edge labels are represented using a map called `elabel` between a pair of vertices to their label, where the pair of vertices are outgoing and incoming vertices, respectively. Here all the vertices are labeled singularly and there will be edges connected between two vertices which are always directed from one vertex to another. There are no multiple labels for a vertex and no multiple edges between two vertices. The `inverse` parameter specifies whether to store inverse adjacency sets for parents. The `name` parameter just specifies the name of the graph.

```scala
class MGraph [TLabel: ClassTag] (ch: Array[SET[Int]],
   label: Array[TLabel],
   elabel: Map[Pair, TLabel],
   inverse: Boolean,
   name: String)
```

2.6 **Cardinality Restriction**

Fard [8] introduced modified versions of graph simulation algorithms called cardinality restricted. For example, he defines “cardinality restricted or CAR-dual simulation, in which the number of match children or parents with the same label in the data graph should not be less than their correspondents in the query.” A graph `G` is said to be the cardinality of graph `Q` if the number of matches of children and parents with the same label in `G` is not less than that of the number of children and parents matches with same labels in `Q`. Cardinality restriction improves the accuracy and the quality of the result set. By this condition, experiment results show that many vertices have
been pruned to match the query graph and this is a step towards subgraph isomorphism.

![Figure 2.7: Cardinality restriction](image)

Figure 2.7 shows examples for the condition where the data and query graphs match by graph simulation algorithms but do not match by cardinality restriction. In the data graph, vertex id 2 has only one child with label M, whereas it is supposed to have two children with label M.

## 2.7 Database Systems

A Graph Databases use a graph structure for storing data in the form of vertices and their relationship in form of edges. This relationship allows each of the vertex data to be linked together. The graph data model is very simple and thus benefits easy retrieval of data as compared to traditional databases.
2.7.1 **NEO4J**

Neo4j\(^1\) the most popular open source graph database implemented in Java. It uses Cypher query language and is a transactional database. It implements the property graph model efficiently as opposed to graph processing or in-memory libraries. It saves data as vertices and edges and uses Lucene for indexing.

2.7.2 **ORIENT DB**

It is an Multi-model Database Management System which supports Graph, Document, key/value store object model\(^2\). It is implemented in Java. It supports schema-less, schema-full and schema-mixed modes and includes SQL among its query languages. It uses Gremlin query language to query the database and uses B-Tree algorithm for indexing along with many other indexing techniques.

2.8 **QUERY PROCESSING**

Query processing is the process of finding the matches of a given query in the data graph. The corresponding vertex or edge label criteria can be relaxed using Wildcards, Regular expressions, Variables, and Predicates.

Query processing for graph databases can be processed in many different approaches. Query processing usually contains two or three main steps, Query formulating, processing the query efficiently using different approaches and fetching the appropriate results. In this section we will see how query processing has been approached in recent years.

\(^1\)[http://neo4j.com/developer/graph-database/#_what_is_neo4j]
\(^2\)[http://orientdb.com/orientdb/]
The query processing for graphs are divided into two steps [21]. First step is filtering and indexing. The filtering process is done by indexing which eliminates all the unmatched results. Then query processing step which verifies whether the query is a match for each subgraphs. Since the eliminated graph is much smaller than the original data graph, query processing using the indices is more efficient.

In [1] Query processing is done using FG-index (Frequent subGraphs indexing feature). It is shown that the Frequent graph queries are answered without verification and infrequent queries only a small number of verifications. However this approach is suitable only for query patterns that are frequent. When all the queries are totally infrequent this takes lot of processing time and increases the cost of indexing.

It is very interesting to see how gStore and [20] handle the graph based SPARQL query engine. They develop an index on the data graph signature with pruning rules and search algorithms. They have an adjacency list table with the vertex id and vertex label corresponding to its outgoing edges and its edge labels. And they have hash function where they map two vertex ids to its corresponding edge label.

There are many approaches involving indexing as to load the index into the memory before graph query processing. These approaches are fast but will not work if the graph are very large and the index becomes too large for the memory.

2.9 Query Languages

Graph databases often have their own query language to insert, delete, update and query the data from its database. Unlike SQL they does not have a single universal query language. The main goal of query processing or patter matching is to find the exact match of the small query graph in
the existing huge data graph more quickly and efficiently. Some of the popular query languages for graph database are as follows.

2.9.1 **SPARQL**

SPARQL\(^3\) is an RDF (Resource Description Framework) Query language and is not based on XML. RDF model is based on graphs which are in form of Subject-Predicate-Object expressions and are known as triplets. They store the data internally as triples. Subject and Object denotes resource or nodes, Predicate denotes edges or relationship between the subject and the object.

2.9.2 **Cypher**

Cypher\(^4\) is a graph query language used for the Neo4j database and graphs through this can be represented easily and efficiently. Querying and updating (insertion, deletion, and updating) the database is easy as comparable to the other query language and hence is called simple query language but yet powerful. It is sometimes called as the SQL for the graph database. Cypher uses certain keywords like CREATE, MATCH, RETURN which are inspired by SQL and SPARQL.

2.9.3 **Gremlin**

Gremlin\(^5\) is graph query language developed for a multi-relational graph (key-value pair) which is also called as a property graph. Gremlin works very well when there is an extreme level of graph traversals. It makes use of various graph databases, packages, and framework. Gremlin is Single-Relational graph database (all the edge relationship has the same meaning) Gremlin has been implemented in Java and is open source, developed by Apache Tinkerpop.

---

\(^3\)[https://www.w3.org/TR/rdf-sparql-query/]
\(^4\)[https://neo4j.com/developer/cypher/]
\(^5\)[http://gremlindocs.spmallette.documentup.com/]

16
CHAPTER 3

GRAPH ANALYTICS WITH CARDINALITY AND EDGE LABELS

3.1 CAR-GRAPH SIMULATION WITH EDGE LABELS

CAR-graph simulation is an extension of Graph Simulation [12], which adds the cardinality restriction to the match of the graph. Here the number of children matches with the same label in $G$ should not be less than the number of children matches with same labels in $Q$. Along with checking for vertex label for child matches we also check for edge labels for child matches.
Algorithm 1: CAR-Graph Sim for Edge Labeled graph

**Input:** Data Graph $G$, Query Graph $Q$

**Output:** CAR-Graph Edge labeled Match Set

1. Calculate Array of match set for query graph in data graph
2. Calculate $cMatch =$ Child match for $Q$ in $G$
3. Filter on Edge labels in $Q$ for each child $(u, u_c)$ in $(v, v_c)$
4. Check,
   (I) If $v_c$ does not have child in $u_c$, then remove $v$.
   (II) If there is no $cMatch$ for $Q$ in $G$, then remove $v$.
5. Repeat Steps 3 and 4, till qRange
6. We will have the result set for CAR Graph Sim with edge labels.

Algorithm 1 states how we apply cardinality and edge label restriction to the Graph simulation algorithm. Existing Graph simulation calculates the match set and removes the vertices that do not have the same children match set. We here calculate the children match with Multiset data structure as it allows duplicates. Along with filtering on edge labels in step 3, we also check for the child match from that bag of Multiset and remove the set of vertices which were not found in step 4(ii).

Figure 3.1: Data for Graph Sim and Dual Sim

The results for Graph simulation without edge label, cardinality restriction without edge labels, Graph simulation with edge labels, and cardinality restriction with edge labels are shown in table
Table 3.1: Graph Simulation Matches

<table>
<thead>
<tr>
<th>Simulation Algorithm</th>
<th>Matched Set</th>
<th>(#distVertices, #distEdges)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph Simulation</td>
<td>0 --&gt; (0, 5, 7) 1 --&gt; (1, 6, 8) 2 --&gt; (3) 3 --&gt; (9, 2, 10, 4, 11) 4 --&gt; (9, 2, 10, 4, 11)</td>
<td>(12,18)</td>
</tr>
<tr>
<td>Graph Sim CAR</td>
<td>0 --&gt; (0, 5, 7) 1 --&gt; (1, 8) 2 --&gt; (3) 3 --&gt; (9, 2, 10, 4, 11) 4 --&gt; (9, 2, 10, 4, 11)</td>
<td>(11,14)</td>
</tr>
<tr>
<td>MGraph Simulation</td>
<td>0 --&gt; (0, 7) 1 --&gt; (1, 6) 2 --&gt; (3) 3 --&gt; (9, 2, 10, 4, 11) 4 --&gt; (9, 2, 10, 4, 11)</td>
<td>(10,10)</td>
</tr>
<tr>
<td>MGraph Sim CAR</td>
<td>0 --&gt; (0) 1 --&gt; (1) 2 --&gt; (3) 3 --&gt; (9, 2, 10, 4, 11) 4 --&gt; (9, 2, 10, 4, 11)</td>
<td>(8,5)</td>
</tr>
</tbody>
</table>

3.1 for the data graph shown in Figure 3.1. The vertex set (5, 8) are removed for MGraph simulation result set from Graph simulation result set since the edge labels for the corresponding vertices do not match with the query graph edge labels. Since the vertex set (6, 7) from MGraph simulation do not have another child with vertex label M, they are removed for the cardinality match results with edge labels.

3.2 CAR-DUAL SIMULATION WITH EDGE LABELS

CAR-Dual simulation is an extension of Dual Simulation [12], which adds the cardinality restriction to the match of the graph. Here the number of child matches with the same label in G should not be less than the number of child matches with same labels in Q and the number of parent
matches with the same label in $G$ should not be less than the number of parent matches with same labels in $Q$. Along with checking for cardinality restriction with vertex labels for parent and child matches, we check for edge labels for parent and child matches.

**Algorithm 2: CAR-Dual Sim for Edge Labeled graph**

**Input:** Data Graph $G$, Query Graph $Q$

**Output:** CAR-Dual Edge Labeled Match Set

1. Calculate Array of match set for query graph in data graph
2. Calculate cMatch = Child match for $Q$ in $G$
3. Calculate pMatch = Parent match for $Q$ in $G$
4. Filter on Edge labels in $Q$ for each child $(u, u_c) in (v, v_c)$
5. Check,
   (I) If $v_c$ does not have child in $u_c$, then remove $v$.
   (II) If there is no cMatch for $Q$ in $G$, then remove $v$.
6. Repeat Steps 4 and 5, till qRange
7. Filter on Edge labels in $Q$ for each parent $(u_p, u) in (v_p, v)$
8. Check,
   (I) If $v_p$ does not have Parent in $u_p$, then remove $v$.
   (II) If there is no pMatch for $Q$ in $G$, then remove $v$.
9. Repeat Steps 7 and 8, till qRange
10. We will have the result set for CAR Dual Sim with edge labels.

Algorithm 2 is similar to algorithm 1, but extends to check cardinality and edge labels restriction for parents match set also. This is indicated in steps 7 and step 8. The results for Dual simulation without edge label, cardinality without edge labels, Dual simulation with edge labels, and cardinality restriction with edge labels are shown in table 3.2 for the data graph shown in Figure3.1. The vertex set (5, 8, 9) are removed for MDual simulation result set from Dual simulation result set since the edge labels for the corresponding vertices do not match with the query graph edge labels. Since the vertex set (6, 7) from MDual simulation do not have another child with vertex label M, they are removed for the cardinality match results.
Table 3.2: Dual Simulation Matches

<table>
<thead>
<tr>
<th>Simulation Algorithm</th>
<th>Matched Set</th>
<th>(#distVertices, #distEdges)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dual Simulation</td>
<td>0 -&gt; (0, 5, 7) 1 -&gt; (1, 6, 8) 2 -&gt; (3) 3 -&gt; (9, 2, 4) 4 -&gt; (9, 2, 4)</td>
<td>(10,16)</td>
</tr>
<tr>
<td>Dual Sim CAR</td>
<td>0 -&gt; (0, 5, 7) 1 -&gt; (1, 8) 2 -&gt; (3) 3 -&gt; (9, 2, 4) 4 -&gt; (9, 2, 4)</td>
<td>(9,12)</td>
</tr>
<tr>
<td>MDual Simulation</td>
<td>0 -&gt; (0, 7) 1 -&gt; (1, 6) 2 -&gt; (3) 3 -&gt; (2, 4) 4 -&gt; (2, 4)</td>
<td>(7,9)</td>
</tr>
<tr>
<td>MDual Sim CAR</td>
<td>0 -&gt; (0) 1 -&gt; (1) 2 -&gt; (3) 3 -&gt; (2, 4) 4 -&gt; (2, 4)</td>
<td>(5,5)</td>
</tr>
</tbody>
</table>

3.3 CAR-Strict Simulation with Edge Labels

CAR-Strict simulation is an extension of Strict Simulation [7], which adds the cardinality restriction to the match of the graph and along with that, we check for the label match for edge relationships. In CAR-Strict algorithms, instead of creating ball from Dual Simulation we create balls from the match set of a CAR-Dual algorithm. Everything else is same as the Strict Simulation algorithm.

The results for Strict simulation without edge label, with edge labels, and cardinality restriction with edge labels are shown in table 3.3 for the data graph shown in Figure3.2. The vertex set (5, 14) is removed for MStrict simulation result set from Strict simulation result set since the edge labels for the corresponding vertices does not match with the query graph edge labels. Since the
vertex set (12, 13, 15) from MStrict simulation do not match under cardinality restriction, the vertices within the set are removed for the cardinality result match set.

3.4 CAR-TIGHT SIMULATION WITH EDGE LABELS

CAR-Tight simulation is an extension of Tight Simulation [6], which adds the cardinality restriction to the match of the graph and along with that, we check for the label match for edge relationships. In CAR-Tight algorithms, instead of creating ball from Dual Simulation we create balls from the match set of a CAR-Dual algorithm. Everything else is same as Tight Simulation.

The results for Tight simulation without edge label, with edge labels, and cardinality restriction with edge labels are shown in Table 3.4 for the data graph shown in Figure3.2. The vertex set (5, 8, 9) is removed for MTight simulation result set from Tight simulation result set since the edge
Table 3.3: Strict Simulation Matches

<table>
<thead>
<tr>
<th>Simulation Algorithm</th>
<th>Matched Set</th>
<th>(#distVertices, #distEdges)</th>
<th>Ball Centers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strict Simulation</td>
<td>0 -&gt; (0, 9, 13, 5, 7) 1 -&gt; (12, 1, 6, 14, 8) 2 -&gt; (15, 3) 3 -&gt; (2, 10, 4, 11) 4 -&gt; (2, 10, 4, 11)</td>
<td>(16,25)</td>
<td>(0, 15, 9, 1, 2, 3, 10, 4, 11, 12, 13, 5, 7, 14)</td>
</tr>
<tr>
<td>MStrict Sim</td>
<td>0 -&gt; (0, 9, 13, 7) 1 -&gt; (12, 1, 6, 8) 2 -&gt; (15, 3) 3 -&gt; (2, 10, 4, 11) 4 -&gt; (2, 10, 4, 11)</td>
<td>(14,19)</td>
<td>(0, 15, 12, 9, 1, 13, 2, 3, 10, 7, 4)</td>
</tr>
<tr>
<td>Strict Sim CAR</td>
<td>0 -&gt; (0, 9, 5, 7) 1 -&gt; (1, 6, 8) 2 -&gt; (3) 3 -&gt; (2, 10, 4, 11) 4 -&gt; (2, 10, 4, 11)</td>
<td>(12,17)</td>
<td>(0, 9, 1, 5, 2, 3, 7, 4)</td>
</tr>
<tr>
<td>MStrict Sim CAR</td>
<td>0 -&gt; (0, 9, 7) 1 -&gt; (1, 6, 8) 2 -&gt; (3) 3 -&gt; (2, 10, 4, 11) 4 -&gt; (2, 10, 4, 11)</td>
<td>(11,15)</td>
<td>(0, 9, 1, 2, 3, 7, 4)</td>
</tr>
</tbody>
</table>

labels for the corresponding vertices does not match with the query graph edge labels. Since the vertex set (5, 6, 7, 8, 9) from MTight simulation does not match under cardinality restriction, the vertices within the set are removed for the cardinality result match set.
Table 3.4: Tight Simulation Matches

<table>
<thead>
<tr>
<th>Simulation Algorithm</th>
<th>Matched Set</th>
<th>(#distVertices, #distEdges)</th>
<th>Ball Centers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tight Simulation</td>
<td>0 → (0, 13, 5) 1 → (12, 1, 14) 2 → (15, 3) 3 → (2, 10, 4, 11) 4 → (2, 10, 4, 11)</td>
<td>(12, 15)</td>
<td>(12, 1, 14)</td>
</tr>
<tr>
<td>MTight Sim</td>
<td>0 → (0, 13) 1 → (12, 1) 2 → (15, 3) 3 → (2, 10, 4) 4 → (2, 10, 4)</td>
<td>(9, 9)</td>
<td>(1)</td>
</tr>
<tr>
<td>Tight Sim CAR</td>
<td>0 → (0, 5) 1 → (1) 2 → (3) 3 → (2, 4) 4 → (2, 4)</td>
<td>(6, 7)</td>
<td>(1)</td>
</tr>
<tr>
<td>MTight Sim CAR</td>
<td>0 → (0) 1 → (1) 2 → (3) 3 → (2, 4) 4 → (2, 4)</td>
<td>(5, 5)</td>
<td>(1)</td>
</tr>
</tbody>
</table>
CHAPTER 4

QUERY PATTERN PROCESSING

4.1 WILDCARDS

Given an input string and a wildcard pattern, we have to implement an algorithm to find if the pattern is matched against the input string. The search pattern here is edge labels which can be extended to vertex labels. The wildcard pattern has two special characters.

1. ‘?’ or ‘_’ Match any single character in the string.
2. ‘*’ Match zero or more characters in the string.

This problem can be solved using a regular expression but is not efficient and affects the performance. For efficient string matching, we follow the solution proposed by[1] [16]

Matching any single character wildcard is quite simple, it checks the length of the pattern and the string and if both have the same length then it proceeds. In next step, it compares character by character of the string and the pattern and ignores a character if a single wildcard character is found in the pattern. For example "univ_rsi_y" matches "university". Here we consider ‘.’ as single wildcard character.

To match one or more character in the string, the natural approach is to find all the possible changes for every appearance of the wildcard character in the pattern but there would be a large number of matching possibilities. We only want to know if the string matches or not but not the

combination of all the matching possibilities, so the pattern string is divided into blocks as shown in Figure 4.1.

![Figure 4.1: Multiple Wildcard Blocks](image)

Each block except the first one will start with a wildcard character and end either when it encounters the next wildcard character or until the end of the pattern is reached. So in here it matches block by block and moves forward if successful, otherwise terminates when it fails. Matching of the pattern for each block is done and moves along the input string until the end of
the string is reached as shown in figure 4.2. The algorithm will terminate when any first solution is found since we only need to know if the pattern matches the string but not the combinations of all the matches.

4.2 Regular Expressions

A regular expression is a special text like pattern used to match one or more characters in a string. It specifies a part or an entire pattern string. The search pattern here is either vertex labels or edge labels. Regular expressions have a higher advantage than wildcards at the cost of performance. A regular expression\(^2\) can be a simple basic string such as "ban[a-z]∗" and would match strings containing the words "bank", "banana", "bandwidth". Regular expression also comprises of dashes, which are used to match a wide range of characters. For example, the regex "[a−z]" would match all the lowercase characters from a to z and the regex "[A−Z]" would match all the uppercase characters from A to Z and the regex "[0−9]" would match all the numbers from 0 to 9. A dot (.) in regex is used to match any single character and an asterisk (∗) is used to match zero or more occurrences of a character and a plus (+) matches one or more occurrences of a character and a question mark (?) is used to match zero or one occurrence of a character.

In this research, we have used two different libraries to check the performance of regular expression on graphs.
1. java.util.regex\(^3\)
2. dk.brics.automaton\(^4\)

The performance of regex is also based on the input string and its length.

\(^2\)www.techterms.com/definition/regular_expression
\(^3\)www.docs.oracle.com/javase/8/docs/api/java/util/regex/package-summary.html
\(^4\)www.brics.dk/automaton/index.html
Figure 4.3: Regex matching

Figure 4.3 shows the regular expression matching for graph databases. The query graph contains regex \[a-z A-Z]\*Smith which matches any string ending with "Smith" and Georgia[a-z A-Z]\* which matches with any string starting with "Georgia". It gets all the match result set with last name Smith who studied at university starting with Georgia.

4.2.1 JAVA.UTIL.REGEX

java.util.regex.Pattern uses BoyerMoore string search algorithm. This algorithm is based on backtracking and on the idea that the pattern can be shifted in the input string if it is matched from right to left. Initially places an anchor point where the starting string might start by moving from left to right. Then this algorithm starts scanning to right of the anchor point and starts scanning leftwards towards the anchor point.

Pattern and Matcher are the two classes for matching pattern specified by regular expression against
the string in java.util.regex package. A Pattern is said to be the compiled representation of a regular expression. It represents a regular expression that is defined in string. It returns the pattern which is to be interpreted by the matcher. A Matcher performs the match operation on the string by rendering a pattern. A matcher class creates a match from the pattern and it performs different matching operation against the string. A matcher function can also be used to search for same regular expression pattern in different strings.

Initially, a pattern instance is constructed from a given regular expression using pattern class. 

```scala
val compliedPat = Pattern.compile (regex)
```

Then from the created pattern instance, a matcher class is called to check if the given regular expression matcher the input string.

```scala
Matcher jMatcher = compliedPat.matcher (input)
```

The matches function of matcher class returns the result whether it matches, i.e, true or not.

```scala
def == (input: String): Boolean = { matcher (input) find () }
```

---

5. [www.docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html](www.docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html)

6. [www.docs.oracle.com/javase/8/docs/api/java/util/regex/Matcher.html](www.docs.oracle.com/javase/8/docs/api/java/util/regex/Matcher.html)
In this chapter, we discuss the experimental results to compare all the different graph simulation algorithms with Edge labels, Cardinality, Regular expression, and Wildcards.

We have mainly used synthesized datasets along with real-world graph datasets. The parameters for synthesized data graph are, the number of vertices for the data graph (1 Million to 5 Million), its average outdegree (5 to 20), number of distinct vertex labels (100 - 1000), and number of distinct edge labels (10-100). We generate data graphs in two ways, either graphs based on power law or generate uniformly distributed random graphs. Query graphs are generated by using BFS search process. It performs a Breadth-First Search until the required number of specified vertices are found or until the maximum number of retries have been exhausted. By this method, we can ensure that the query graph matches at least one instance in the data graph.

In terms of real-world datasets, we have used amazon-2008 which has 735,323 vertices and 5,158,388 edges; wordnet3 which has 82,670 vertices and 132,964 edges; web-google which has 916,428 vertices and 5,105,039 edges [15, 4]. All experiments were run on a machine with 128GB RAM, AMD Opteron, IB interconnect having 48 cores.
5.1 Precision of Simulation Algorithms

We here compare the precision of graph simulation algorithms. We get the total number of vertices in the match set for five iterations and calculate the mean precision of all simulation algorithms. We here consider Subgraph Isomorphism algorithm to be perfect and match the result set of all algorithm compared to the DualIsomorphism algorithm. Precision is defined to be the ratio of the number of vertices of Subgraph Isomorphism divided by the number of vertices of the simulation algorithm in the result set. Figure 5.1 shows the precision for Uniformly generated graphs for data graphs. We have used \texttt{genRandomConnectedGraph} for random generation of data graphs with 1000 and 2000 number of vertices and 10 number of vertex labels with average outdegree of 20. We have used \texttt{genBFSQuery} for query graph generation with size of 10 number of vertices and average outdegree of 2. We have used \texttt{genPowerLawGraph} to generate power law graphs with same number of parameters as random graph generation. Since the Graph simulation algorithm is less constrained it contains all its children matches and hence we can see its low precision. Dual Simulation is slightly more constrained where it checks its parents matches as well. Strict and Tight have slightly more precision as compared to Dual. The CAR versions of all the algorithm have high precision as it checks for cardinality constraints.

Figures 5.2 and 5.3 shows precision for Power graphs and Real world graphs. We can see apart from Graph simulation, that Dual Strict and Tight to have a prefect precision.
Figure 5.1: Precision Random Graph

Figure 5.2: Precision for Power Graph
Figure 5.3: Precision for Real-world Graph
5.2 **Mutable vs. Immutable Data-structures**

Mutable objects are the one which can be changed and Immutable objects are the ones that cannot be changed. In Scala, mutable collections are often faster than the immutable ones\(^1\) because we are working on the same memory addresses. Thus, data will not be copied over and over. Due to this literature survey\(^2\), we created graphs with the mutable data structures. Surprisingly our experiment results show the other way around for graphs simulation algorithms.

![Figure 5.4: Graph Mutable vs. Immutable](image)

**Figure 5.4: Graph Mutable vs. Immutable**

Figure 5.4 shows the performance evaluation for Mutable vs. Immutable Graph simulation and Figure 5.5 is for Dual simulation. We collected the results for uniform and power graphs. It shows that Immutable data-structure are faster than Mutable data-structure. This should be further investigated using profilers.

Also we can observe that the Power law graphs are faster than the uniformly distributed graphs (average outdegree at 5.4) due to the fact that the generation of edges are very less in power law

\(^{1}\)https://www.tobyhobson.co.uk/scala-collections-performance/
\(^{2}\)https://medium.com/@hussachai/scalas-immutable-collections-can-be-slow-as-a-snail-da6fc24bc688
\(^{3}\)https://capecoder.wordpress.com/2012/07/29/scalamapbenchmark/
graphs (average outdegree at 1.9) than uniformly generated random graphs and also the edges are unevenly distribution in power law graphs so as the search takes less time.

5.3 COMPARISON OF GRAPHS WITH EDGE LABELED GRAPHS

Since we added edge labels to graphs we wanted to check the performances of graph simulation algorithms with edge labels and compare it with graphs without edge labels. As expected we see an increase in time of graph simulation algorithms for graphs with edge labels.

Figures 5.6 and 5.7 shows the performance comparison for Graph and Dual with MGraph and MDual simulation algorithms. MDual simulation takes the most time among simulation algorithms as it checks for edge labels for parents match along with children match.
Figure 5.6: GraphSim vs. MGraphSim

Figure 5.7: DualSim vs. MDualSim
5.4 **Comparison of Graphs with Cardinality Graphs**

In this section, we compare the performance evaluation for graphs with cardinality. We have added the cardinality restriction criteria to the existing graphs and we see an increase in time for cardinality restriction. Figures 5.8 and 5.9 shows the performance comparison for Graph and Dual with and without cardinality restrictions. We have used multiset to check the children matches and parent matches for query graph in the data graph. Dual simulation checks edge labels for both child and parent result match set and take more time as compared to Graph simulation which checks only for child edge label match set. The CAR versions of both graph simulation algorithm take more time due to additional constraints for cardinality restriction.

![Graph vs. GraphCAR](image-url)

*Figure 5.8: Graph vs. GraphCAR*
In this section, we evaluate the performance of cardinality graphs with and without edge labels. Since we added edge label and cardinality restrictions to graph simulation algorithms we wanted to check their performance. Figures 5.10 and 5.11 shows the performance of Graph and Dual CAR with MGraph and MDual CAR, respectively. We can see from the graph that the random MGraph CAR and random MDual CAR algorithm take the most time among all of them.

5.5 Comparison of Cardinality Graphs with Edge Labeled Cardinality Graphs
Figure 5.10: GraphCAR vs. MGraphCAR

Figure 5.11: DualCAR vs. MDualCAR
5.6 Simulation Performance Comparisons

In this section, we compare all the graph simulation algorithm for their performance and it is shown in Figure 5.12 in logarithmic scale. we can see that Immutable DualSim2 takes the least amount of time followed by GraphSim2, because it changes the order of loop and batch subtraction of vertices which did not match. Strict and Tight simulation algorithms in the mutable package take more time and this needs to be fixed.

![Comparison of Simulation Algorithms](image)

Figure 5.12: Comparison of Simulation Algorithms

5.7 Wildcard vs. Regex vs. None

In this section, we evaluate the performances of graph query processing using wildcards and regular expression in three different cases.

(1) Query graph can have either all of its edge labels as wildcards or regex.

(2) We specify query graphs to have half of its edge labels to be wildcards or regex and another half to be regular strings.

(3) All the edge labels are of type string.
Figures 5.13 and 5.14 describe the performance comparisons of Graph and Dual simulation with Wildcard and Regex. We can see that the Regex takes more time than wildcard or string comparison. It runs even faster when Scala’s equals is used.

![MGraph Wc vs. Rx](image)

Figure 5.13: MGraphSim Wc vs. Rx vs. None
Figure 5.14: MDualSim Wc vs. Rx vs. None
In this research, we have implemented a mutable graph structure from immutable expecting it to be faster. We have extended edge labels for graph, strict and tight simulation algorithms which was not considered till now. So far the vertex labels were represented by integers or double, but here we have changed it from fixed type definition to a type parameter. We have done experiments for edge labels of type integer, double and strings.

We have implemented the concept of cardinality restriction [5] for the different simulation algorithms and implemented it to the edge labels as well. The results are more convincing in comparison to non-CAR matching. Also, we have used Multiset for the CAR implementation.

In this research, we have implemented query processing for graph simulation algorithms using Wildcards and extended the Regular expression matching to other graph simulation algorithms.

All the algorithms have been run on real-world graphs along with synthesized large data graphs with millions of vertices and edges and all the performance results were measured and compared.
In this research, we concentrate on query processing for vertex and edge labels using Wildcards, Regular expression, and a single variable. This can be extended to multiple variables like in SPARQL. Also, query processing can be done for predicates like not equal, lesser than, greater than, lesser than or equal, greater than or equal and many more operations.

An extension can also be added to all the graph simulation algorithms to check not only the parents and children of the matched vertices but also their grandparents and grandchildren of the matched vertices, which might reduce the result match set, making it closer to subgraph isomorphism.

Also, we can extend the algorithms to use multiple vertex/edge labels and also to have multiple edges between the same two vertices.
BIBLIOGRAPHY


APPENDIX

DK.BRICS.AUTOMATON.REGEXP

dk.brics.automaton.regex\(^1\) is written in Java and it applies deterministic finite-state automata (DFA) unlike most of the other regular expression packages which are all based on nondeterministic automata (NFA). Independent of the complexity of the regular expression, the matching is done in linear time to check if the regex matches the string or not and this is because it converts the regular expression into an automaton and is matched against the string.

At first, it constructs a new RunAutomaton\(^2\) from a deterministic automaton, i.e. it compiles the regular expression to recognize the regex.

\[
\text{val compliedPat = new RunAutomaton (new RegExp (regexString).toAutomaton)}
\]

Then it will create a pattern matcher that will match the given input against the regular expression pattern.

\[
\text{var bMatcher AutomatonMatcher = compliedPat.newMatcher (inputString)}
\]

The equals equals match operator determines whether the input string matches the given regex string and returns the boolean value true if matches, else return false by calling the find function which checks the next occurrence of regex.

\[
\text{def == (_input: String): Boolean = { matcher (_input) find () }}
\]

Brics results rarely do not match with Java regex results depending upon the regular expression. This report\(^3\) shows the benchmark of different libraries available and it shows that the Brics

---

\(^1\)www.brics.dk/automaton/index.html  
\(^3\)www.tusker.org/regex/regex_benchmark.html
regex is nearly hundred of times faster in executions of operations per second but Brics only agrees with java.util.regex library in 2 of 5 test cases.