BB-GRAPH: A NEW SUBGRAPH ISOMORPHISM ALGORITHM FOR QUERYING BIG GRAPH DATABASES

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

MERVE ASİLER

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

SEPTEMBER 2016

Approval of the thesis:

BB-GRAPH: A NEW SUBGRAPH ISOMORPHISM ALGORITHM FOR QUERYING BIG GRAPH DATABASES

submitted by **MERVE ASİLER** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

ABSTRACT

BB-GRAPH: A NEW SUBGRAPH ISOMORPHISM ALGORITHM FOR QUERYING BIG GRAPH DATABASES

Asiler, Merve
M.S., Department of Computer Engineering
Supervisor : Prof. Dr. Adnan Yazıcı

September 2016, 61 pages

With the emergence of the big data concept, the big graph database model has become very popular since it provides very flexible and quick querying for the cases that require costly join operations in RDBMs. However, it is a big challenge to find all exact matches of a query graph in a big database graph, which is known as the subgraph isomorphism problem. Although many related studies exist in literature, there is not a perfect algorithm that works for all types of queries efficiently since it is an NP-hard problem. The current subgraph isomorphism approaches built on Ullmann's idea focus on the strategy of pruning out the irrelevant candidates. Nevertheless, for some databases and queries, their pruning techniques are inadequate. Therefore, they result in poor performance. Moreover, some of those algorithms need large indices that cause extra memory consumption. Motivated by these, we introduce a new subgraph isomorphism algorithm, namely BB-Graph, for querying big database graphs in an efficient manner without requiring large data structures. We test and compare our algorithm with the existing ones, GraphQL and Cypher of Neo4j, on some very big graph database applications and show that our algorithm performs better for most of the query types.

Keywords: graph isomorphism algorithm, pruning, graph database, Neo4j

ÖZ

BB-GRAPH: BÜYÜK ÇİZGE VERİTABANLARINI SORGULAMAK İÇİN YENİ BİR ALTÇİZGE EŞYAPILILIK ALGORİTMASI

Asiler, Merve
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Prof. Dr. Adnan Yazıcı

Eylül 2016 , 61 sayfa

Büyük veri kavramının doğmasıyla, ilişkisel veritabanı yönetim sisteminde masraflı birleştirme operasyonları gerektiren durumlarda daha esnek ve hızlı sorgulama sağladığı için büyük çizge veritabanı modeli çok popüler olmuştur. Ancak, altçizge eşyapılılık problemi olarak bilinen, bir çizgesel sorgunun bir veritabanı çizgesindeki tüm tam eşleşmelerini bulmak oldukça zordur. Literatürde ilgili birçok çalışma olmasına rağmen, bu NP-hard bir problem olduğundan, tüm sorgu tipleri için verimli çalışan kusursuz bir algoritma bulunmamaktadır. Ullmann'ın fikri üzerine kurulan şu anki altçizge eşyapılılık yaklaşımları, ilgisiz adayları budama stratejisine odaklanmaktadır. Yine de, bazı veritabanları ve sorgular için, bu algoritmaların kullandıkları budama teknikleri yetersiz kalmaktadır. O yüzden, zayıf performansa yol açarlar. Ayrıca, bu algoritmalardan bazıları ek hafıza tüketimine sebep olan dizinlere gereksinim duyar. Bunlardan motivasyon alarak, büyük veritabanı çizgelerini ek veri yapılarına ihtiyaç olmadan verimlice sorgulamak için yeni bir altçizge eşyapılılık algoritması BB-Graph'ı geliştirdik. Algoritmamızı çok büyük bazı çizge veritabanı uygulamalarında var olan algoritmalarla, GraphQL ve Neo4j'nin Cypher'ı ile, karşılaştırarak test ettik ve BB-Graph'ın çoğu sorgu tipi için daha iyi performans sergilediğini gösterdik.

Anahtar Kelimeler: altçizge eşyapılılık algoritması, budamak, çizge veritabanı, Neo4j

To my beloved ones...

ACKNOWLEDGMENTS

I want to acknowledge that I have written this thesis by granting so much things from my own besides making hard efforts and spending long working hours. However, I have always taken strength from the wonderful people around me, and I am not sure that if I can find eneough words to express all my gratitude to them.

Initially, I want to express my gratitude towards my supervisor Prof. Dr. Adnan YAZICI. Throughout the whole thesis work, I have always been more eager to study with his motivation, high enery and favour. I am so much thankful for his patience and understanding that he showed at every point that I feel myself in a bottleneck. He always encouraged me for the better, and hence, I believe in that I have done a good job under his guidance.

I would also like to thank the committee members of my M.Sc. thesis, Prof. Dr. Ahmet COŞAR, Prof. Dr. Halit OĞUZTÜZÜN, Assoc. Prof. Dr. İbrahim KÖRPEOĞLU and Assoc. Prof. Dr. Murat KOYUNCU for their precious discussions and suggestions about my thesis.

My very special thanks to KALE YAZILIM for sharing their resources, especially the real-world databases, and giving the opportunity to work among them in their company. I would like to thank Ali İNCE and Volkan YIKILMAZ for their suggestions. Also, I am so thankful to İrfan Nuri KARACA since he never hesitated to spare time for helping me and searching for a solution.

I would also like to thank the people who are very speacial for me, my best friends Adnan KILIÇ, Alperen EROĞLU, Hilal ARSLAN, Serdar ÇİFTÇİ and Okan Tarhan TURSUN. Since the suffering years of double major education, Adnan has always been a great friend. With his colorful personality, all difficulties has turned into an amusement. Whenever I get upset, he cheered me up with his humors and he has been the first person who I consult in any technical subject. Alperen was my best teacher during the undergraduate years and my best workmate along my assistanship. At each time I feel desperate, he showed me some way out. He has always listened to me tirelessly. He is indeed one of the most helpful and bighearted people that I have ever known. Hilal, my dear roommate, has been the one with whom I have established a close friendship since the first day that we met. Her positive point of view has always inspired me. With her warm personality, I could feel that I am not alone. Serdar has also been the person who does not leave me alone in the department. I have learned many things from him. When I need to decide about something, I have trusted in his experiences. His advices have really been very beneficial for me. Last, but not least, Okan, my mentor, has been the one who has given the hope that beautiful things can happen in life. From the beginning of my graduate study, he has been the person that I have taken a leaf out of his book. He has always tried to give me a lead about how I can improve my skills. Especially, he helped me very much for the final touches on my thesis. With his own smile over the misfortunes of life, he could make me smile too. In short, it has been a big fun to spend time with all of them and I am really lucky to have friends like them. They are the best friends one can have.

There are more friends that I want to thank. Even though we can not have the opportunity of meeting frequently, my dear friends Gülay Özdemir, Eda Ceren GÜNGÖR and Aslı PAKKAN have never refrained to support me. Gülay, my classmate in undergraduate, has never stopped keeping in touch after graduating and continued her friendship through the messages most of the time and almost every day in the last term of my thesis. I really appreciate her being such a talented person in showing empathy towards people, especially me. I could feel much better when she understands me. Eda was my first friend that I made in the university and since that time she has stayed as one of my most sincere friends. Her unusual way of thinking has passed to me also, and this has brought me the skill of evaluating the things in life from very different aspects. It has always been entertaining and worthful to spend time and analyzing the life with her. Moreover, I want to thank Aslı for her precious friendship and suggestions on life plans. Her nice thoughts about me have really motivated me expecially in the last term of my thesis.

Lastly, I want to thank three wonderful people; my father Hasan ASİLER, my mother Nazife ASİLER, and my brother Murat Çağrı ASİLER. When one gets old, s/he understands worth of the family is something immeasurable. I even can not imagine that how I could overcome the difficulties in life without their understanding, attention and love. With the support of my father, I could always believe in the power inside me. His intimate love and hugs could improve my mood immediately even when I felt I was in the deepest. I have admired him since my childhood, and I have tried to be hardworking, virtuous and friendly like him. In many times, my mother's sweet words inspired me to go on. I have learned from her that the things which seem unfeasible can be achieved by believing and struggling in fact. Moreover, her affection has also made me behave polite to people and be understanding towards them. Next, I want to thank my brother Çağrı for bringing joy to our lives with his funny and humorous way of thinking. Listening to his daily life adventures have always been the time of the day that I can not wait. His tact has also made my heart warm in my difficult days. He is, in fact, the life and soul of our family.

I want to acknowledge that I have been supported by TUBITAK (The Scientific and Technological Research Council of Turkey) within BIDEB 2210 graduate study scholarship program along my studies for master degree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF ABBREVIATIONS

DB	Database
w.l.o.g.	Without Loss Of Generality
s.t.	Such That
w.r.t.	With Respect To
i.e.	In Other Words
LIFO	Last In First Out
query node	Node In Query Graph
database node	Node In Database Graph
query relationship	Relationship In Query Graph
database relationship	Relationship In Database Graph
DFS	Depth First Search
BFS	Breadth First Search

LIST OF SYMBOLS

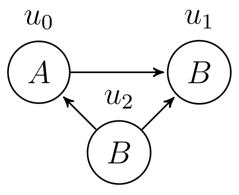
q, Q	Query graph
g, G	Database graph
M	Set of exact matches of q in g
M_{node}	Set of current node matches
M_{rel}	Set of current relationship matches
$M_{<u,v>}$	The match of query node u and database node v
$< u, u' >$	The relationship in $OUTGOING$ direction from the node u to node u'
S	Stack
$r.type$	The type of relationship r
$r.dir^u$	The direction of relationship r w.r.t. node u
$u.deg$	The degree (i.e. the number of relationships) of node u

CHAPTER 1

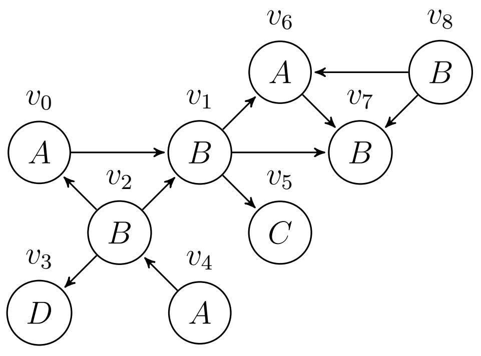
INTRODUCTION

In the last decade, big graph database models have widespreaded in a large variety of industry areas such as communications (Cisco.com, Cisco HMP, Deutsche Telecom, Telenor), logistics (Accenture), online job search (GlassDoor), Web/ISV (Adobe), Network Management (SFR, Hewlett Packard), social networks (Viadeo), mobile communication applications (Maaii), data center management (Junisphere), education (teachscape), bioinformatics (SevenBridges Genomics), etc. Graph database modelling has recently been preferred to the others for database modelling because of its flexibly better fitting into structure of data and providing higher performance than RDBMs in many cases [9], [3], [17], [1].

The subgraph isomorphism problem has been one of the most frequently encountered challenges in big graph database applications. It can be defined as follows: Given a query graph q and a database graph g, find all matching instances of q in g. To illustrate, in Figure 1.1 there exist two instances of q in g, one is the subgraph consisting of the nodes v_0, v_1, v_2 and the relationships $< v_0, v_1 >$, $< v_2, v_0 >$, $< v_2, v_1 >$ and the other one is the subgraph consisting of the nodes v_6, v_7, v_8 and the relationships $< v_6, v_7 >$, $< v_8, v_6 >$, $< v_8, v_7 >$. The subgraph isomorphism problem is known as NP-hard [2]. In almost all big database applications using graphs, there frequently occur queries directly or indirectly related to subgraph isomorphism problem, therefore it is definitely important to find an efficient solution to this problem.



(a) query graph: q



(b) database graph: g

Figure 1.1: An example query and a database graph with unlabeled relationships

1.1 Motivation

Most of the graph isomorphism algorithms in the literature are based on feature indexing or based on the idea of checking candidates node-by-node developed from Ullmann's Algorithm [16]. The algorithms of the first type aims to decrase the number of candidate data sets by using the *filtering-and-verification* technique together with an index. They first create an index of small graphs (features) and then eliminate irrelevant candidate data graphs with respect to indexed features. GraphGrep [5], GIndex [21], Labeled Walk Index (LWI) [15], Closure-Tree [7], Graph Decomposition Indexing [18], TreePi [22], TreeDelta [25] are examples to this type of algorithms. The algorithms of the second type applies the *branch-and-bound* technique on nodes and edges by following the *backtracking* strategy such that they first find all candidates for each node of the query graph and then they check for the existence of candidate edges between the matched database nodes for each relationship in the query graph. At each step, they expand partially-matched graphs edge by edge. If there doesn't exist any candidate edge for some relationship in the query, then the algorithm jumps one step back , i.e. backtracks, and replaces the previous match with an other candidate edge or node. VF2 [4], QuickSI [14], GADDI [23], GraphQL [8], SPath [24] belong to this type of algorithms.

Although both approaches are inspiring and well-considered, the existing studies may perform poorly depending on the database size (number of nodes, relationships and disconnected graph pieces), density, variety of labels, and query type. To the best of our knowledge, the current indexing methods are not applicable to one-piece graphs since their purpose is to prune out the irrelevant members from the database consisting of many pieces where each piece is independent from the others. Also, they are insufficient to deduce all of the exact matches, that is; they are designed to focus on decreasing the size of solution set by eliminating pieces which are guarenteed not having any instance of query, hence they don't go further when a piece is proved to include at least one exact match or not to contain any matches. On the other hand, the algorithms which are derived from Ullmann's idea have a very high computational complexity due to a large number of recursive calls. Actually, the fact that there are lots of candidates for each query node cause this circumstance. However, a great majority of these candidate checks are redundant, since they are done for the nodes which are not in close neighbourhood of each other. The existing algorithms try to reduce the number of candidate nodes by applying their own pruning rules. Although they become successful to throw out the ones which can never be the sought member node of an exact match, there may still remains lots of candidates. The reason is that; these algorithms extract the candidates for each query node from all accross the database graph in beginning, and keeping members from every isomorphic instance of a query in one set complicates the matching phase because the vertices which are candidate for the same node but, in fact, belong to different isomophic patterns can not be discriminated from each other. Hence, searching for a relationship between two nodes which locate in distinct matches of query graph causes inefficiency. An other disadvantage of these methods is that they generally need additional large data structures or use their own indexes to keep some piece of cooked data on hand, but bulding up those structures requires an extra amount of labor and memory. Also, operations using such data may cause a time loss much more than the predicted while the main purpose is to save time by pruning out irrelevant candidates. As a result of all these, those algorithms do not have satisfying performance for querying big graph databases.

Motivated by these, in this thesis, a new branch-and-bound algorithm blended with backtracking strategy for graph isomorphism problem, called BB-Graph, is introduced. BB-Graph searches for candidates of the first query node to be matched (the start node) throughout the whole database, but unlike the other algorithms, for each exact match it selects the candidates of other query nodes from the local area of the one matched with start node. Hence, it follows a more efficient way of finding and matching candidates through the substantial decrease in a number of redundant checks by annihilating most of the impossible combinations. Moreover, BB-Graph doesn't use large data structures except the built-in information supplied by Neo4j. Also, contrary to the experiments done with the other algorithms existing in literature, BB-Graph is tested on a very big and real graph database with directed edges consisting of nearly 70 millions of nodes by connecting to remote database instead of reading data from a file (as in iGraph framework [6]). The experiments conducted with this real database show that BB-Graph performs better than GraphQL and Neo4j's graph query language, Cypher, in many of the cases. Lastly, in this thesis the experimental results obtained for different matching orders are presented.

1.2 Contribution

The main contributions of this thesis can be summarized as in the following:

1. A new subgraph isomorphism algorithm, BB-Graph, is introduced. Similar to the other algorithms, BB-Graph also uses branch-and-bound tecnique while expanding a partial match and pruning out the anomalious nodes and relationships in order to reach an exact match of the query, and backtracks to handle the other possible relationship and node combinations which was skipped in previous checks. Nevertheless, BB-Graph has many advantages over existing methods in the literature:

   (a) Contrary to the others, BB-Graph follows a more efficient search strategy while matching graph elements (nodes and relationships): It selects start node candidates from all across the database graph and then, for each candidate isomorphism region potentially created by the start node match, it selects the other nodes' candidates locally by traversing every neighbour node and relationship.

   (b) BB-Graph uses only the built-in data structures supplied by Neo4j. It doesn't consume extra memory for large indexing or data storing.

(c) Unlike some other algorithms' experiments conducted on undirected graphs by reading the input from a BLOB file, BB-Graph has been tested and evaluated on a very big one-piece directed graph with almost 70 millions of nodes through remote database connection.

(d) A number of experiments have been done for comparing our algorithm (BB-Graph) with Cypher and GraphQL have showed that BB-Graph has a better performance than those for many query types.

2. The effect of change in node matching order is given by analysing the experimental results obtained by different matching orders.

1.3 Outline

Organization of the rest of this thesis is as in the following: Section 2 gives the background and related work. Section 3 presents our algorithm, *BB-Graph*, and we provide the pseudocodes of proposed solutions. Section 4 shows the experimental results and comparison of BB-Graph with Cypher and GraphQL. Lastly, the conclusion and feature work is given in Section 5.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Problem Definition

In this thesis, the notation $g : (V, E)$ means that there is a graph g with the vertex set V and edge set E. For a vertex v; L_v and P_v denote the label set and property set of v, respectively. Similarly, for an edge $e =< u, v >$; L_e, P_e and dir_e^u denote the label set, property set and direction of e with respect to node u, respectively.

For two graphs $g_1 : (V_1, E_1)$ and $g_2 : (V_2, E_2)$, g_2 is said to be the **exact match** of g_1, if there is a one-to-one and onto function $f : V_1 \longrightarrow V_2$ such that $L_v \subseteq L_{f(v)}$, $P_v \subseteq P_{f(v)}$ $\forall v \in V_1$ and $s =< f(u), f(v) > \in E_2$ $\forall r =< u, v > \in E_1$ where $L_r \subseteq L_s$, $P_r \subseteq P_s$, $dir_r^u = dir_s^{f(u)}$.

Definition: Given a query graph q and a data graph g, the problem of finding all distinct exact matches of q in g is defined as **Graph Isomorphism Problem**.

From the definition, it is clearly understood that if a node u is matched with v then $L_u \subseteq L_v$ and $u.deg \leq v.deg$. Thus, the tests to search for the existence of these two conditions are called as *label comparison test* and *degree comparison test*, respectively.

2.2 BFS Coding

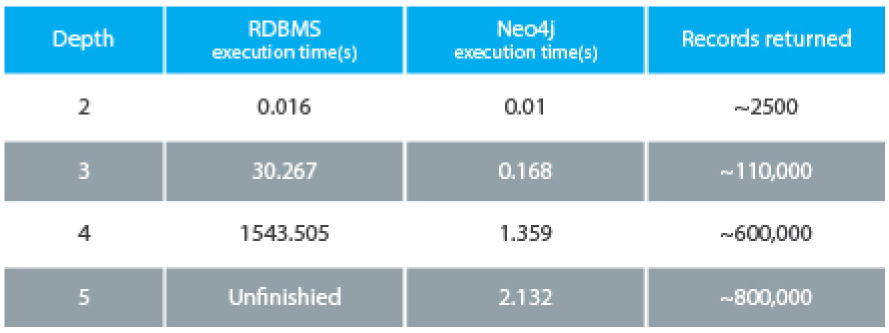
To give the query graph as input to our code, there is a need for some representation technique in lexicographic order. To achieve this, BFS Coding was preferred; that is a transformation from DFS Coding introduced in [20]. BFS Coding is done as in the following:

1. Enumerate the nodes starting from 0 like $v_0, v_1, v_2, ..., v_n$ in the order of BFS traversal.

2. Starting from v_0 write all edges traversing the graph in BFS order with the following rule: $(i, j, l_i, p_i, l_r, p_r, l_j, p_j)$ where i, l_i and p_i are the id, label and property set of v_i, respectively, l_r and p_r are the label and property set of the relation r with start node v_i and end node v_j, respectively, and j, l_j and p_j are the id, label and property set of v_j, respectively and $i < j$.

2.3 Comparisons Between Graph Databases and Relational Databases

Recently, graph databases have been more popular than relational databases because they can efficiently handle some database operations which can be a challenge in RDBMs. Many existing studies comparing the graph databases and relational databases show that graph databases perform much better than RDBMs with respect to many aspects [9], [3], [17], [1], [10], [19], [12], [13]. According to those studies, generally, the join operations on relational tables cause high costs in querying RDBMs since that kind of operations require pre-knowledge of how the tables are structured and related to each other. On the other hand, according to their experimental results previously done in the literature, such a problem disappears in graph databases, mostly by directly traversing the nodes and relationships. In Figure 2.1, the results of an experiment done with recursive queries that require many join operations are given [1]. Experimental results of the comparison between relational databases and graph databases show that graph databases have quite higher performance than RDBMs for such type of queries. Moreover, the previous studies also show that for insert-delete-update operations graph databases provide more flexibilty than relational databases do.



Depth	RDBMS execution time(s)	Neo4j execution time(s)	Records returned
2	0.016	0.01	~2500
3	30.267	0.168	~110,000
4	1543.505	1.359	~600,000
5	Unfinishied	2.132	~800,000

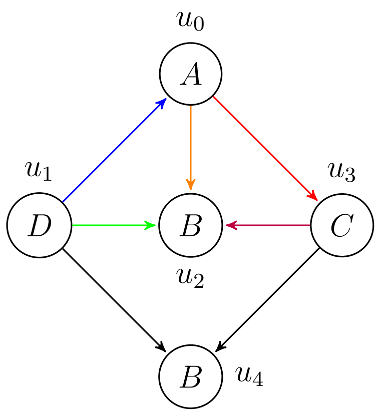
Figure 2.1: Comparison results for running time performaces of searching a recursive query with difference depths in RDBMs and Neo4j [1]

2.4 Ullmann Algorithm And The Methods Derived From It

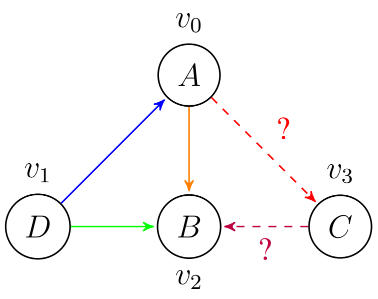
Ullmann Algorithm [16] is the first search method that is developed to find isomorphic patterns of query graphs in large graphs. It basically consists of 4 main steps:

- Filtering candidates for each query graph node,
- Selecting a candidate for each query graph node, trying to match the node with its candidate together by matching the corresponding relationships between the currently processed node and previously matched nodes,
- Replacing the selected candidate with an another one if the current match doesn't work, and
- Backtracking in order to try other candidates and to find more isomorphisms.

To filter the candidates for each query node, Ullmann Algorithm applies the label and vertex degree comparison tests (described in Section 2.1). After the filtering step, it starts with matching the nodes in a recursive manner with the order of the nodes given in the input. To match two nodes u and v, it checks for each relationship between u and some previously matched query graph node u', if there is a corresponding relationship between v and the database graph node matched with u'. In other words, to be intermateable, u and v must satisfy that for each relationship $r =< u, u' >$ (or $r =< u', u >$) where $M_{<u',v'>} \in M_{node}$ for some $v' \in V_G$, there is a corresponding relationship $s =< v, v' >$ (or $s =< v', v >$, respectively) where $r.type = s.type$. In the study of Lee et al.[11], the procedure to check the existence of this condition is called as ISJOINABLE. In Figure 2.2, ISJOINABLE operation for the match of u_3 and v_3 in graphs A and B, respectively, is illustrated. Each u_i has been matched with v_i where $0 \leq i \leq 2$ and the corresponding relationships matched with each other are showed in the same color. When the matching turn comes to u_3 and v_3, ISJOINABLE procedure searchs candidates for the relationships between u_3 and previously matched nodes u_0 and u_2 (the relationship between u_3 and u_4 isn't considered since u_4 isn't matched yet), so it checks if there is any relationship between v_0 and v_3 and between v_2 and v_3 which has the same characteristics (direction, type) with the one between u_0 and u_3 and the one between u_2 and u_3, respectively.



(a) graph A



(b) graph B

Figure 2.2: ISJOINABLE procedure of u_3 and v_3 while matching the graphs A and B

In the matching phase of query node u and database node v, if there isn't any problem in the end of ISJOINABLE procedure, $M_{<u,v>}$ is added into the list of matched nodes. Then, isomorphism search continues by picking the next not-yet-matched query node in order to match that with one of its candidates. In the case that ISJOINABLE procedure returns false for a query node and its matching node, the current match is cancelled and the next candidate is selected this time. In either case when all the matched nodes result in an exact match of the query graph or one of the node matches results in an failure, the algorithm backtracks to try other candidates.

Ullmann Algorithm is good in terms of finding all isomorphic patterns of the query graph in the large database graph and working in backtracking manner so that it can handle the evaluation of possible combinations of node matches in an efficient way because of testing the common parts of different combinations at a time instead of evaluating each combination one-by-one and wasting time with checks of the common matches repeatedly. Although it has a robust structure, Ullmann Algorithm is in a raw state such that its performance can be increased with well-thought matching order strategies and effective pruning rules and short cuts. Thereby, the following 5 methods are built on the skeleton of Ullmann Algorithm and enriched by their own techniques over the aspects mentioned above.

VF2 [4] is one of those algorithms derived from Ullmann's Algorithm. It matches the nodes in an increasing order of number of labels regarding a specified query graph by following their strategy. It selects the next query node from among the nodes connected to at least one of the previously matched query nodes with a relationship. In this way, it can eliminate more candidates during ISJOINABLE stage. Moreover, VF2 algorithm refines the candidates before passing to ISJOINABLE step by applying degree comparison test for already matched and not-yet-matched neighbour nodes. It divides the not-yet-matched adjacent nodes into two as the ones in first-degree-neighbourhood of the already matched nodes and the ones not in that area. Then it makes degree comparison separately for both of two sets adjacent to query node u with their correspondents that are adjacent to candidate of u.

Another algorithm derived from Ullmann's is **QuickSI** [14]. The key aspect of this method is that it defines a data structure named as QI-Sequence which provides efficient pruning; and thus, resulting in low-cost processing. QI-Sequence is a minimum spanning tree created based on edges weighted according to the number of each node label and the number of each <start node label - relationship type - end node label> triple in database graph. Starting to match from the relationships and nodes with low frequency, there occurs less possibilities to test and in this way QuickSI is able to decrease the number of recursive calls. Also, QuickSI uses QI-Sequence in indexing of the features in database graphs and takes advantage of its tree structure to prune candidate graphs. As its pruning strategy, QuickSI applies ISJOINABLE procedure of a query node u by beginning the checks from the relationship between u and its parent node in QI-Sequence (in case that u is not the root node).

The third algorithm is **GADDI** [23], introduced by Zhang et al.. Their solution is based on how to refine the candidate nodes by using a distance based indexing. Their main purpose is to eliminate the cadidate nodes by examining their k-neighbourhood in case there are not at least as many specific fragments as there are in the k-neighbourhood of query node. Selecting the fragments that are used in neighborhood comparison, the discriminative ones which occur with different frequencies in common k-neighbourhood of sample pairs of database node are picked. GADDI uses 3 different pruning rule to refine the candidates: For a query node u and a candidate node v, firstly for each node u' in k-neighborhood of u, it tries to find a candidate v' in k-neighbourhood of v by *label comparison test*. Secondly, for the common k-neighbourhood of each (u, u') pair, it counts the number of discriminative fragments in this area and prunes out u if there are less number of occurences of a specific fragment in the corresponding region obtained by (v, v'). Thirdly, for each u' it compares the length of shortest path, l, between u and u' with the one, l', between v and v' and u is eliminated in the case $l < l'$ for at least one u'. Furthermore, GADDI applies these pruning rules in reverse manner for each candidate v' in neighbourhood of v. Lastly, as matching order, GADDI selects the first node randomly, the rest are selected by DFS.

GraphQL [8], is another algorithm for subgraph isomorphism problem, focuses on neigbourhood relations to filter candidates for a query node. If a query node u can be matched with a database node v, then for each query node u' in k-neighbourhood of u, there must be a candidate node v' in k-neighbourhood of v. Thus, GraphQL uses this fact to prune out false candidates of a query node, by scanning their k-neighbourhood upto a refinement level r, incrementally for each k where $1 \leq k \leq r$. Also, GraphQL follows an optimized node matching strategy by selecting the query node which is estimated to decrease the cost at each intermediate step and adjacent to set of already matched nodes.

Lastly, as a candidate path matching version of Ullmann's method, **SPath** algorithm [24] handles candidate paths. It actually does nothing but matches more than one node on a linear sequence at a recursive call by applying ISJOINABLE procedure for each node on the path. SPath algorithm filters the candidate vertices by checking the number of each node label in their k-neighbourhood, where k is a parameter for the radius of neighbourhood. It applies a rule which is; for each node label L; total number of occurences of L in the neighbourhood upto k^{th} level of u must be less than or equal to the total number of occurences of L in the neighbourhood upto the k^{th} level of v where v is candidate node for a query vertex u. While matching the paths, the algorithm follows a decreasing order of path selectivity defined as a metric based on size of candidate node sets.

The algorithms summarized above remain incapable of showing good performance in some or many cases. Each algorithm has some drawbacks: The pruning techniques of VF2 are not powerful enough, and the matching order it follows is effective only when database graph has similar node label statistics with query graph. QuickSI has to go around the whole database to deduce the information about label and <label-relationship type-label> triple count used in edge weighting; moreover, it should keep up-to-date the data, therefore it additionally needs B+ -trees for this purpose. GADDI creates a large index that keeps the number of discriminative fragments in the intersected k-neighbourhood of each node pair in the whole database, which requires a very exhaustive pre-computation. Futhermore, the pruning rules of GADDI are not effective, and quite time-consuming. GraphQL compares neighbourhoods of query nodes with their candidates' neighbourhoods and tries to find a semi-perfect bipartite matching between the nodes in corresponding neighbouhoods but our experiments showed that this is already an exhaustive computation even when the refinement level is set 1 and not very effective for reducing the candidate set size of some query nodes. SPath needs a data structure including the number of each label with shortest distance i from v for each database node v and for each i from 1 to k; which requires a long pre-computation time and large storage. Also, the experiments in [11] show that the ordering based on path selectivity does not provide a good performance for searching the graph in database.

In their great study [11], Lee et al. compare these five algorithms VF2, QuickSI, GADDI, GraphQL, and SPath on various type of real-world data sets by using their own re-implementations on iGraph framework [6]. They work on small and large undirected graphs consisting of one or many pieces by testing with subgraph, clique and path queries. The experiments that they conduct show that there is not a perfect algorithm which works for all types of database queries efficiently. For instance, while

QuickSI shows a good performance in many cases, it fails to return the answer in a reasonable time for NASA dataset. According to the experimental results that they have obtained, GraphQL is the only algorithm succeeding to respond in a reasonable time for all tests. They state that these start-of-the-art algorithms may perform poorly because of their ineffective matching order and the imbalance between efficiency and overhead of their pruning methods.

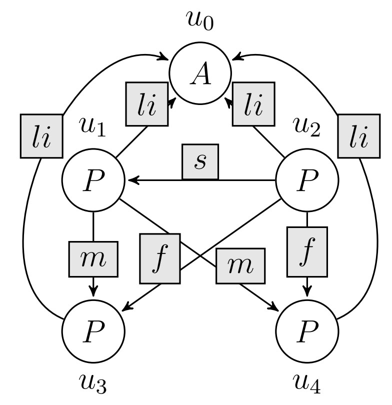
Although each algorithm has its own defect, there is a common point that causes all of these algorithms to perform poorly. All start to search by trying to find and filter candidate nodes for each query vertex. The pruning rules that they use are generally effective for eliminating the nodes that can never be a representative of the query node for which it is selected as a candidate in a real exact match. However, since they don't apply the prunings by regarding each isomorphism as an independent one, the nodes belonging to different isomorphisms cannot be discriminated until their relationships are checked in matching phase. Therefore; the database nodes which are candidate for different query vertices and members of distinct isomorphisms seem available for being members of the same isomorphism at first sight. Hence, the main time loss occurs at this point while searching for an actually non-existing reasonable connection between those irrelevant nodes. In order to remove this kind of possible cases, after the start node candidates are taken from all across the database, candidate nodes for the rest of the query vertices should be selected depending on the start node match. In other words, each start node candidate potentially creates a distinct isomorphism region, and so the representatives for the other query vertices should be locally chosen from the close neighbourhood of the start node for each distinct exact match.

CHAPTER 3

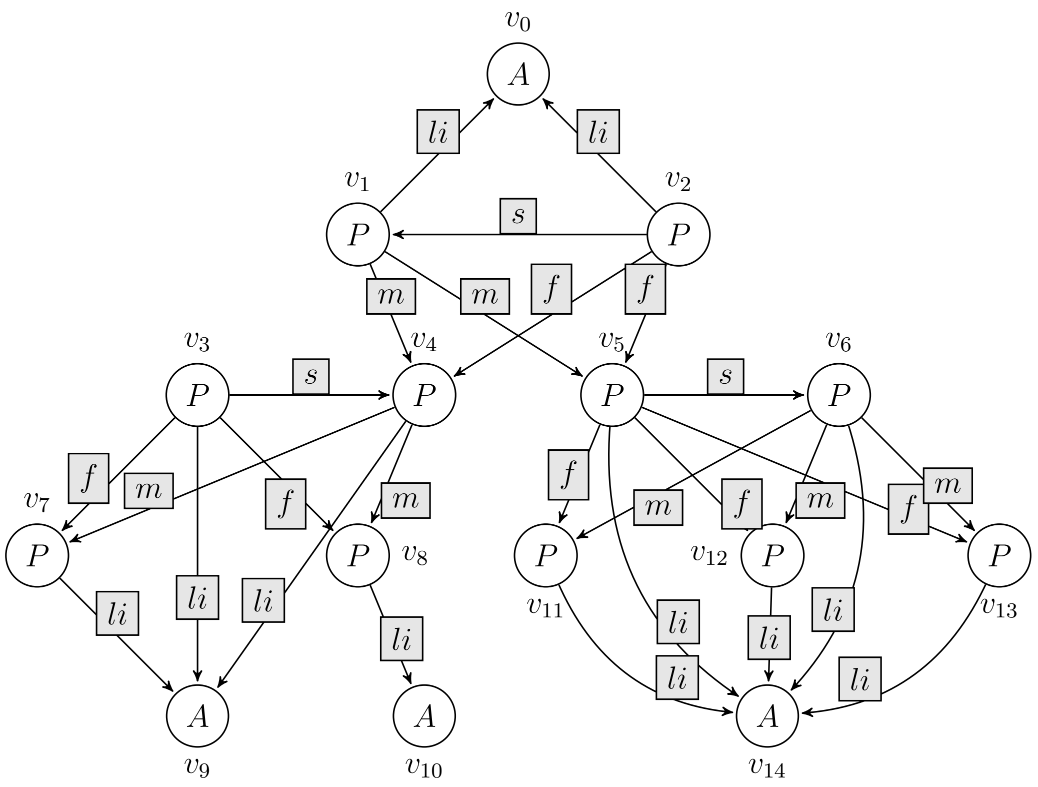
BB-GRAPH: BRANCH & BOUND ALGORITHM FOR GRAPH ISOMORPHISM IN BIG GRAPH DATA MODEL

BB-Graph firstly finds the candidate database nodes for the start node u_s of query graph. For each candidate node v_s found, whether isomorphism(s) of the query graph can or cannot be obtained by v_s is checked. Checking is done by branching as follows: Initially, BB-Graph examines the neighbourhood of v_s. It detects relationships of v_s which can be candidate for those of u_s by using some filtering rules and tries to match each relationship r of u_s with one of its candidates. During the match operations of relationships, if the mandatory node matches appear (e.g. if $r =< u_s, u'_s >$ and $s =< v_s, v'_s >$, then matching r and s brings the match of u'_s and v'_s), then those are pushed into a stack so as to match their relationships later. If all the relationships of u_s can be successfully matched, then an another node match, say $M_{<u,v>}$, is popped from the stack and the same branching procedure is applied for the non-matched relationships of u this time. Recursively, the branching operations are done for each node match until there occurs a contradiction with some match or each query graph item (nodes and relationships) are successfully matched and an exact match of the query is obtained. In both cases, a backtrace process is required to try the other possible relationship, and therefore node matches since there can be more than one choice for matching any relationship.

Section 3.1 gives all the details of algorithm steps which are filtering rules, branching and matching procedure, backtrack mechanism and usage of stack together with examples in figures and pseudocodes.



(a) query graph: q



(b) database graph: g

Figure 3.1: An Example query and database graph where P: Person, A: Address, m: motherOf, f: fatherOf, s: spouse, li: livesIn

3.1 Algorithm Steps

3.1.1 Filter For Start Node Of Query Graph And Filtering Rules

In the beginning, the start node of a query graph that is firstly matched during the isomorphism search is determined. The first query node given in the input is selected as the start node (Algorithm 1, line 3). After determining the start node, BB-Graph tries to find candidates for the start node by filtering the entire database. If there is not any node specified with some property, filtering is completed in 2 stages; otherwise, the propertied nodes passed from the first two stages go through a third stage.

Algorithm 1: BB-GRAPH SEARCH ALGORITHM

```
Input: $q$ : Query graph with n vertices
       $g$ : Database graph
Output: $M$ : Set of all exact matches of q in g

1  begin
2      $M \leftarrow \emptyset$
3      $u_s \leftarrow u_0$ where $u_0$ is the first node given in the input
4      $C_{u_s} \leftarrow$ FILTERBYLABEL($u_s$)
5      $C_{u_s} \leftarrow$ FILTERBYRELATIONSHIPS($u_s$, $C_{u_s}$)
6      if $\underline{u_s \text{ has property}}$ then
7          $C_{u_s} \leftarrow$ FILTERBYPROPERTY($u_s$, $C_{u_s}$)
8      end
9      forall $\underline{v_s \in C_{u_s}}$ do
           // Reset the temporary storage
10         $M_{node} \leftarrow \emptyset, M_{rel} \leftarrow \emptyset, S \leftarrow \emptyset$
11         $S$.push($M_{<u_s,v_s>}$)
12         $M_{node}$.add($M_{<u_s,v_s>}$)
13         TRANSITIONSTATE()
14     end
15     return $M$
16 end
```

The first stage is called as $FilterByLabel$, that is, database nodes are filtered depending on their labels (Algorithm 1, line 4 & Algorithm 2). Since, the database nodes which have a different label from query node u cannot be matched with u, BB-Graph directly eliminates those. This operation doesn't take any time since Neo4j has already the data structure holding the nodes grouped by their labels so it returns the result immediately (almost in 0 milisecond).

Algorithm 2: FILTERBYLABEL(u)

```
Input: $u$ : Query node whose candidates will be found
Output: $C_u$ : Set of candidate nodes for $u$
1 begin
2     Return the database nodes which includes all the labels of u
3 end
```

Algorithm 3: FILTERBYRELATIONSHIPS(u, C_u)

```
Input: $u$ : Query node whose candidates will be filtered
       $C_u$ : Candidate set for $u$ constructed by label
Output: $C_u^*$ : Set of candidate nodes for $u$
1 begin
2     $C_u^* \leftarrow \emptyset$
3     $L_G \leftarrow$ List of groups G of the adjacent relationships of $u$ based on
        $< type, direction >$
4     forall $v \in C_u$ do
5         for each $G_{<type,direction>} \in L_G$, if $v$ has at least $|G_{<type,direction>}|$ many
            number of adjacent relationships of type $type$ and direction $direction$
            w.r.t. $v$ then add $v$ into $C_u^*$
6     end
7     return $C_u^*$
8 end
```

The second stage is called as $FilterByRelationships$ such that the number of relationships of each candidate node are checked (Algorithm 1, line 5 & Algorithm 3). They are compared with the number of relationships of query node according to their type and direction. Let c_t^d be the number of relationships of query node u with type t and direction d with respect to u. Also, let $\{< t_i, d_i >\}$ be the set of different <type, direction> tuples representing the type and direction of relationships of u. Then, the database nodes which can be matched with u must have at least $c_{t_i}^{d_i}$ many relationships with type t_i and direction d_i with respect to that node for each i. If a node doesn't satisfy this rule, then it is eliminated. This stage takes linear time depending on the number of candidate nodes.

For the query nodes, which don't have any property, filtering process ends up here. However, for the nodes that have property, BB-Graph filters the nodes with respect to their property values, which is called as $FilterByProperty$ (Algorithm 1, line 6-8 &, Algorithm 4). The nodes that satisfy all the conditions specified in property values of query node successfully pass this stage as well. Since Neo4j provides a schema indexing feature for property-value indexing, this stage is dependent on the number of candidate nodes only and can be completed in linear time.

Algorithm 4: FILTERBYPROPERTY(u, C_u)

```
Input: $u$ : Query node whose candidates will be filtered
       $C_u$ : Candidate set for $u$ constructed by label and relationships
Output: $C_u^*$ : Set of candidate nodes for $u$
1 begin
2     $C_u^* \leftarrow \emptyset$
3     forall $\underline{v \in C_u}$ do
4         for each different property $p$ of $u$, if $v$ satisfies the same value
            conditions as $u$ for $p$ then add $v$ into $C_u^*$
5     end
6     return $C_u^*$
7 end
```

In Figure 3.1, a sample query and a database graph are showed. Graph g is the database of population and q is the query graph representing a family consisting of mother (u_1), father (u_2), two children (u_3 and u_4) all living in the same address (u_0). To illustrate how filtering stage works, let's assume that the start node is u_1. Initially, $FilterByLabel$ stage eliminates all address nodes v_0, v_9, v_{10} and v_{14} because they don't have the label of u_0 which is P. Next, the non-eliminated nodes are sent to $FilterByRelationships$ stage. In this phase, the nodes which have at least 2 outgoing m relationships, at least 1 incoming s and at least 1 outgoing li relationships are selected, and the rest is ruled out. Since only v_0, v_4, v_6 satisfy this condition, those are selected as candidate for u_0. Here, the filtering process ends up since u_0 doesn't have any property. However, to explain $FilterByProperty$ stage, assume that u_0 has $age \geq 40$ and v_0, v_4 and v_6 has $age = 65$, $age = 35$, and $age = 42$, respectively. Then, in this case BB-Graph would eliminate v_4 since the value of node property age is not ≥ 40, which is asked.

Three conditions used in the filtering stages mentioned above must actually be satisfied by candidates of every query node. Thus, except the first, the other two rules are used to eliminate some of candidates of query nodes matched in proceeding steps of BB-Graph also (The first condition is directly satisfied by "relationship type-node label uniqueness".).

Algorithm 5: TRANSITIONSTATE

```
/* global $S$, $M$, $M_{node}$, $M_{rel}$                       */
1  begin
2      if $\underline{S \neq \emptyset}$ then
3          $M_{<u,v>} \leftarrow S$.pop()
4          BRANCHANDMATCH($M_{<u,v>}$)
5          $S$.push($M_{<u,v>}$)            // Take back to backtrack
6      end
7      else
8          $M$.add( $(M_{node}, M_{rel})$ )      // An exact match found
9      end
10     return                               // Backtracking
11 end
```

Algorithm 6: BRANCHANDMATCH

```
/* global $S$, $M_{node}$, $M_{rel}$                                    */
Input: $M_{<u,v>}$: The node match to be branched
begin
    $R_u \leftarrow$ { relationship r of u | $\nexists r_x \in R_g$ s.t. $M_{<r,r_x>} \in M_{rel}$ }
    if $R_u \neq \emptyset$ then
        forall $r_i \in R_u$ do
            $C_{r_i} \leftarrow$ FINDCANDIDATERELATIONSHIPS($M_{<u,v>}, r_i$)
            $ind_i := 0,\ size_i := |C_{r_i}|$
        end
        $i := 0$
        while $i \geq 0$ do
            while $ind_i < size_i$ do
                if $i == |R_u|$ then
                    TRANSITIONSTATE()
                    $i$ - -                                  // Backtracking
                    Take back $M_{rel}$, $M_{node}$ and $S$ to the previous values
                    continue
                end
                $s_i \leftarrow C_{r_i}$.get( $ind_i$ )
                $ind_i$ ++
                if ISMATCHVALID($M_{<r_i,s_i>}, M_{<u,v>}$) then
                    $i$ ++
                end
            end
            $ind_i := 0,\ i$ - -                             // Backtracking
            Take back $M_{rel}$, $M_{node}$ and $S$ to the previous values
        end
    end
    return
end
```

3.1.2 Branching Process of Node Match

When a query node u is matched with one of its candidates v, the match $M_{<u,v>}$ is put into a stack, called "stack of not-branched nodes", in order to expand it later. When it is popped from the stack, previously branched node and relationship matches are assumed to be correct and the branching process starts for $M_{<u,v>}$. In fact, this is a long process which includes exploring and evaluating all possible ways of expanding node match around its first-level-neighborhood. However, BB-Graph interrupts the process after it finds out one valid expansion matching all the non-matched relationships of u and then it proceeds to branching process of other node matches through next recursive calls. The other options for expansion of $M_{<u,v>}$, if exits, are handled by backtracking mechanism later (see Section 3.1.3).

Algorithm 6 shows the steps of branching process. In the first step of branching process of $M_{<u,v>}$, initially, all non-matched relationships of u are detected (Algorithm 6, line 2). Then for each relationship r with type $r.type$ and direction $r.dir$, the candidate relationship set C_r is constituted from v's relationships with $r.type$ and $r.dir$ (Algorithm 6, lines 4-6 & Algorithm 7). Now, for each non-matched relationship r_0, r_1, ..., r_k of u, there exists candidate sets C_{r_0}, C_{r_1}, ..., C_{r_k}, respectively, and so there occur $|C_{r_0}| \times |C_{r_1}| \times ... \times |C_{r_k}|$ many combinations of matches. In Figure 3.2, an example of the candidate set constitution and the obtained combinations of relationships are given. In the second step, each combination $c \in C_{r_0} \times C_{r_1} \times ... \times C_{r_k}$ is checked to see whether or not the relationship matches obtained from c cause a conflict with the previous matches or between each other. In case of a conflict, that combination is ignored.

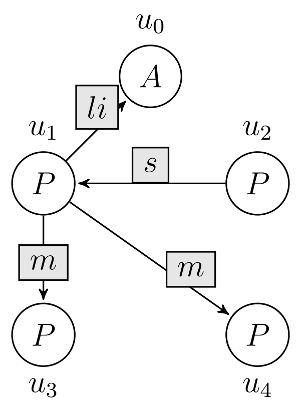
Algorithm 8 shows the conflict control: Let $c = (s_0, s_1, ..., s_k) \in C_{r_0} \times C_{r_1} \times ... \times C_{r_k}$ be a combination of relationships which are candidate for $r_0, r_1, ..., r_k$, where $r_i =< u, u_i >$ and $s_i =< v, v_i >$ (w.l.o.g. we assumed that they are outgoing relationships w.r.t. u and v, but it should be considered as $r_i =< u_i, u >$ and $s_i =< v_i, v >$ in case they are incoming relationships.). In c, there may have occurred 3 kinds of conflict due to any $M_{<r_i,s_i>}$ match: Firstly, providing that u_i has already been matched before, BB-Graph checks if u_i had been matched with v_i. If not, then this conflict invalidates the combination (Algorithm 8, line 4). Secondly, if u_i is not

Algorithm 7: FINDCANDIDATERELATIONSHIPS

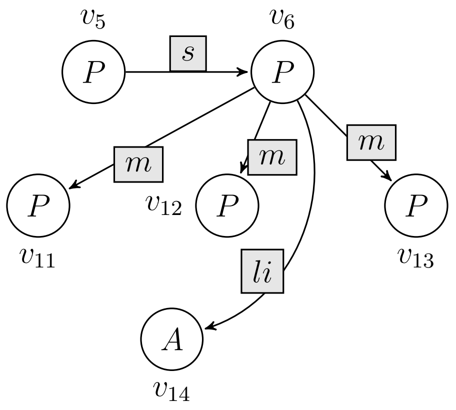
```
Input: $M_{<u,v>}$ : The node match that is branching
       $r_i$ : The query relationship adjacent to $u$ whose candidates are searched
Output: $C_{r_i}$ : Set of candidate relationships for $r_i$

1 begin
2     if $\underline{r_i.dir^u = OUTGOING}$ then
3         $C_{r_i} \leftarrow \{s_i =< v, v' > \mid s_i.type = r_i.type,\ v' \in V_G\}$
4     end
5     else
6         $C_{r_i} \leftarrow \{s_i =< v', v > \mid s_i.type = r_i.type,\ v' \in V_G\}$
7     end
8     return $C_{r_i}$
9 end
```

yet matched but v_i is already among the matched nodes, then it means v_i was previously matched with some node other than u_i, which makes the combination invalid again (Algorithm 8, line 7). Lastly, in case that $M_{<u_i,v_i>}$ is a new match, the convenience of two nodes is checked by $FilterByRelationships$ and $FilterByProperty$ tests explained in Section 3.1.1 (Algorithm 8, line 11). At this point, $FilterByLabel$ test is not necessary baceuse "relationship type-node label uniqueness" already satisfies the label condition; in other words, u_i and v_i cannot have different labels since they are end points of the same type of relationships r_i and s_i. When all the checks are completed, on the condition that there doesn't exist any conflict with $M_{<r_i,s_i>}$ $\forall i = 0, ..., k$; c is considered valid. Additionally, all new matches, $M_{<u_i,v_i>}$s, appeared with $M_{<r_i,s_i>}$ are pushed into "stack of not-branched matches", S, so as to branch in next recursive calls (Algorithm 8, line 13). Branching process of the match $M_{<u,v>}$ is terminated after each combination in $C_{r_0} \times C_{r_1} \times ... \times C_{r_k}$ is evaluated as valid or invalid (Algorithm 6, lines 8-24). Nevertheless, this is a discrete process such that each combination is evaluated at distinct times for the reason that the process is interrupted when a valid combination is explored, and extension of partial isomorphism is maintained from that point (Algorithm 6, line 11), later on the schedule is returned back by backtracking (Algorithm 6, lines 12-14).



(a) neighborhood of u_1 in Figure 3.1



(b) neighborhood of v_6 in Figure 3.1

$C_{<u_1,u_0>} := \{< v_6, v_{14} >\}$
$C_{<u_2,u_1>} := \{< v_5, v_6 >\}$
$C_{<u_1,u_3>} := \{< v_6, v_{11} >, < v_6, v_{12} >, < v_6, v_{13} >\}$
$C_{<u_1,u_4>} := \{< v_6, v_{11} >, < v_6, v_{12} >, < v_6, v_{13} >\}$

(c) candidate relationship sets for non-matched relationships of u_1

$(< v_6, v_{14} >, < v_5, v_6 >, < v_6, v_{11} >, < v_6, v_{11} >)$
$(< v_6, v_{14} >, < v_5, v_6 >, < v_6, v_{11} >, < v_6, v_{12} >)$
$(< v_6, v_{14} >, < v_5, v_6 >, < v_6, v_{11} >, < v_6, v_{13} >)$
$(< v_6, v_{14} >, < v_5, v_6 >, < v_6, v_{12} >, < v_6, v_{11} >)$
$(< v_6, v_{14} >, < v_5, v_6 >, < v_6, v_{12} >, < v_6, v_{12} >)$
$(< v_6, v_{14} >, < v_5, v_6 >, < v_6, v_{12} >, < v_6, v_{13} >)$
$(< v_6, v_{14} >, < v_5, v_6 >, < v_6, v_{13} >, < v_6, v_{11} >)$
$(< v_6, v_{14} >, < v_5, v_6 >, < v_6, v_{13} >, < v_6, v_{12} >)$
$(< v_6, v_{14} >, < v_5, v_6 >, < v_6, v_{13} >, < v_6, v_{13} >)$

(d) combinations of candidate relationships obtained from $C_{<u_1,u_0>} \times C_{<u_2,u_1>} \times C_{<u_1,u_3>} \times C_{<u_1,u_4>}$

Figure 3.2: An example to candidate set construction procedure for the query node u_1 and the database node v_6 given in Figure 3.1

3.1.3 Backtrack Mechanism

During the check of relationship combinations, some prunings are done in implementation phase. Each relationship r_i (where $i = 0, ..., k$) is matched with a candidate relationship $s_i \in C_{r_i}$ in such an incremental way that if there arises a problem (because of the reasons mentioned in Section 3.1.2) with $M_{<r_j,s_j>}$ at j^{th} step where $0 < j \leq k$ then rather than discarding the whole combination, only j^{th} relationship match, $M_{<r_j,s_j>}$, is discarded. Instead of discarded match, a new one is placed by matching r_i with the next candidate in C_{r_i} (Algorithm 6, lines 9, 16-21). If there can not be found an unproblematic match even though all candidates are tested, then the process is backtracked to $(j-1)^{th}$ step and $M_{<r_{j-1},s_{j-1}>}$ is tried to be replaced with a valid match, this time (Algorithm 6, lines 8, 22, 23). Similarly, if $M_{<r_{j-1},s_{j-1}>}$ causes a problem too, it is backtracked to $(j-2)^{th}$ step and so on. If there is a problem in the beginning step, in other words in case that each candidate for r_0 constitutes a match which conflicts with previous matches or leading to conflicting matches in the next steps, then it means that it was a mistake to match u with v, so BB-Graph backtracks to the former depth in chain of recursive calls. Furthermore, even if there doesn't occur any problem until the end of recursive calls and all the matches result in an isomorphism of the query graph, backtracking is done in order to test other cases if there exists any other isomorphism.

3.1.4 Transition State Operations And Stack Usage

For a node match $M_{<u,v>}$, when a valid combination of relationship matches is reached and each non-matched relationship of u is matched with an appropriate relationship of v, the branching process is interrupted and BB-Graph passes to a transition state to start a new branching process of another node match (Algorithm 6, line 11). Algorithm 5 gives the transition state operations. In this state, the node match that is branched in the next call is popped from the stack (Algorithm 5, line 4). After the pop operation, the algorithm recursively calls branching process for the popped match this time (Algorithm 5, line 5). However, if the stack is already empty and all of the query nodes and relationships are matched, then it means an exact match I of the query graph has been found. In this case, I is saved (Algorithm 5, line 8)

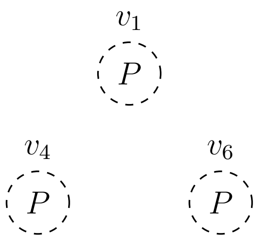
Algorithm 8: IsMatchValid

```
/* global $S$, $M_{node}$, $M_{rel}$ */
Input: $M_{<r_i,s_i>}$: The relationship match to be checked
       $M_{<u,v>}$: The node match currently being branched
begin
    $u_i$ := $r_i$.getOtherNode( $u$ )
    $v_i$ := $s_i$.getOtherNode( $v$ )
    if $\exists u_x \neq u_i$ s.t. $M_{<u_x,v_i>} \in M_{node}$ then
        return $false$
    end
    if $\exists v_x \neq v_i$ s.t. $M_{<u_i,v_x>} \in M_{node}$ then
        return $false$
    end
    if $M_{<u_i,v_i>} \notin M_{node}$ then
        if FilterByRelationships($u_i, \{v_i\}) \neq \emptyset$
         and FilterByProperty($u_i, \{v_i\}) \neq \emptyset$ then
            $M_{node}$.add( $M_{<u_i,v_i>}$ )
            $S_{node}$.add( $M_{<u_i,v_i>}$ )
        end
        else
            return $false$
        end
    end
    $M_{rel}$.add( $M_{<r_i,s_i>}$ )
    return $true$
end
```

and backtrack mechanism is activated with the aim of exploring other exact matches. When backtracking is done in order to evaluate other possibilities, stack and the other match records are turned back into the version before passing to the transition state (Algorithm 5 line 5, Algorithm 6 line 13).

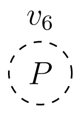
Here, stack is used, instead of queue, because of the strategy that BB-Graph uses in backtracking. Since changes in matches are done by beginning from the last match towards the back when backtracked, a LIFO-logic-data structure provides replacing the old matches with the new ones in an easier and more efficient way. Thus, the usage of stack, together with the backtrack mechanism, also affects the branching order of node matches which plays an important role in algorithm performance (see Section 4.4).

Finally, for the sample query and database graph shown in Figure 3.1, a sample piece of execution of BB-Graph for the case u_1 is the starting node and v_6 is the starting node candidate is described in Figure 3.3 .



Stack	$S = \emptyset$
Start Node	u_1
Candidate List	$C_{u_1} = \{v_6, v_1, v_4\}$

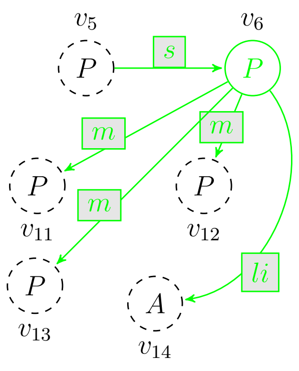
(a) Filtering for the start node



Stack	$S = \{M_{<u_1,v_6>}\}$
Matched Nodes	$M_{node} = \{M_{<u_1,v_6>}\}$
Candidate List	$C_{u_1} = \{v_1, v_4\}$ *(to be evaluated later by backtracking)*

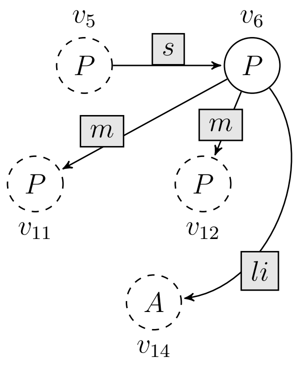
(b) Choosing a candidate for the start node

Figure 3.3: A sample piece of execution for the query and database graph given in Figure 3.1



Popped Match	$M_{<u_1,v_6>}$
Stack	$S = \emptyset$
Matched Nodes	$M_{node} = \{M_{<u_1,v_6>}\}$
Candidate Lists	$C_{<u_1,u_0>} = \{< v_6, v_{14} >\}$ $C_{<u_2,u_1>} = \{< v_5, v_6 >\}$ $C_{<u_1,u_3>} = \{< v_6, v_{11} >, < v_6, v_{12} >, < v_6, v_{13} >\}$ $C_{<u_1,u_4>} = \{< v_6, v_{11} >, < v_6, v_{12} >, < v_6, v_{13} >\}$

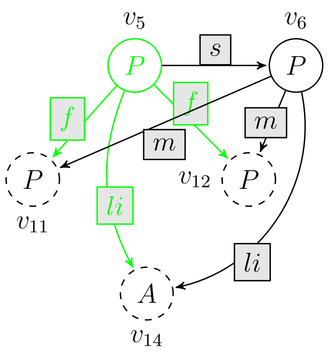
(c) Branching of $M_{<u_1,v_6>}$



Chosen Rel. Combination	$(< v_6, v_{14} >, < v_6, v_{12} >, < v_6, v_{11} >, < v_5, v_6 >)$ *(Other combinations are going to be evaluated later by backtracking)*
Matched Relationships	$M_{rel} = \{M_{<r_1,s_1>}, M_{<r_2,s_2>}, M_{<r_3,s_3>}, M_{<r_4,s_4>}\}$ where $r_1 :< u_1, u_0 >$, $s_1 :< v_6, v_{14} >$ $r_2 :< u_1, u_4 >$, $s_2 :< v_6, v_{12} >$ $r_3 :< u_1, u_3 >$, $s_3 :< v_6, v_{11} >$ $r_4 :< u_2, u_1 >$, $s_4 :< v_5, v_6 >$
Matched Nodes	$M_{node} = \{M_{<u_1,v_6>}, M_{<u_0,v_{14}>}, M_{<u_4,v_{12}>}, M_{<u_3,v_{11}>}, M_{<u_2,v_5>}\}$
Stack	$S = \{M_{<u_0,v_{14}>}, M_{<u_4,v_{12}>}, M_{<u_3,v_{11}>}, M_{<u_2,v_5>}\}$

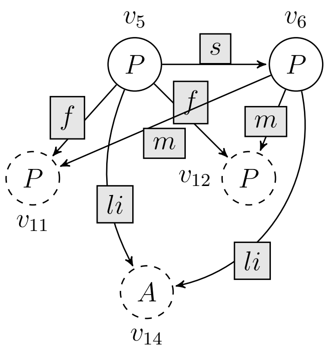
(d) Choosing a relationship combination to be matched

Figure 3.3: A sample piece of execution for the query and database graph given in Figure 3.1 (cont.)



Popped Match	$M_{<u_2,v_5>}$
Stack	$S = \{M_{<u_0,v_{14}>}, M_{<u_4,v_{12}>}, M_{<u_3,v_{11}>}\}$
Matched Nodes	$M_{node} = \{M_{<u_1,v_6>}, M_{<u_0,v_{14}>},$ $M_{<u_4,v_{12}>}, M_{<u_3,v_{11}>},$ $M_{<u_2,v_5>}\}$
Candidate Lists	$C_{<u_2,u_0>} = \{< v_5, v_{14} >\}$ $C_{<u_2,u_3>} = \{< v_5, v_{11} >,$ ~~$< v_5, v_{12} >$~~, ~~$< v_5, v_{13} >$~~$\}$ $C_{<u_2,u_4>} = \{$ ~~$< v_5, v_{11} >$~~, $< v_5, v_{12} >,$ ~~$< v_5, v_{13} >$~~$\}$ *(The crossed candidates are going to be eliminated by* IsMatchValid() *function later by backtracking)*

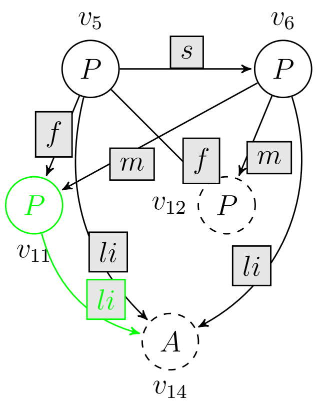
(e) Branching of $M_{<u_2,v_5>}$



Chosen Rel. Combination	$(< v_5, v_{14} >, < v_5, v_{12} >, < v_5, v_{11} >)$ *(Other combinations are going to be evaluated later by backtracking)*
Matched Relationships	$M_{rel} = \{M_{<r_1,s_1>}, M_{<r_2,s_2>}, M_{<r_3,s_3>},$ $M_{<r_4,s_4>}, M_{<r_5,s_5>}, M_{<r_6,s_6>},$ $M_{<r_7,s_7>}\}$ where r_i and s_i are as stated in prev. steps for $1 \leq i \leq 4$, $r_5 :< u_2, u_0 >, s_5 :< v_5, v_{14} >$ $r_6 :< u_2, u_4 >, s_6 :< v_5, v_{12} >$ $r_7 :< u_2, u_3 >, s_7 :< v_5, v_{11} >$
Matched Nodes	$M_{node} = \{M_{<u_1,v_6>}, M_{<u_0,v_{14}>},$ $M_{<u_4,v_{12}>}, M_{<u_3,v_{11}>},$ $M_{<u_2,v_5>}\}$
Stack	$S = \{M_{<u_0,v_{14}>}, M_{<u_4,v_{12}>}, M_{<u_3,v_{11}>}\}$

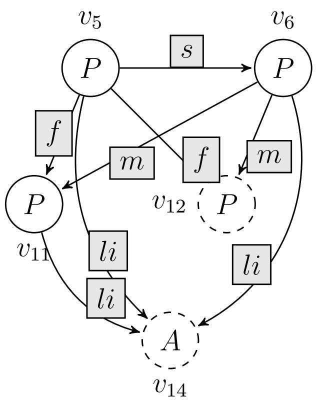
(f) Choosing a relationship combination to be matched

Figure 3.3: A sample piece of execution for the query and database graph given in Figure 3.1 (cont.)



Popped Match	$M_{<u_3,v_{11}>}$
Stack	$S = \{M_{<u_0,v_{14}>}, M_{<u_4,v_{12}>}\}$
Matched Nodes	$M_{node} = \{M_{<u_1,v_6>}, M_{<u_0,v_{14}>}, M_{<u_4,v_{12}>}, M_{<u_3,v_{11}>}, M_{<u_2,v_5>}\}$
Candidate Lists	$C_{<u_3,u_0>} = \{< v_{11}, v_{14} >\}$

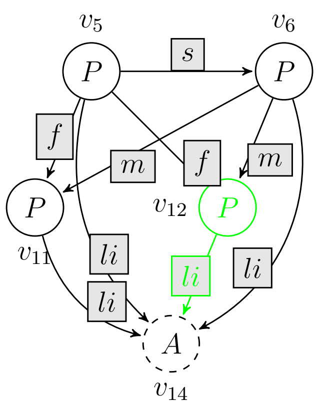
(g) Branching of $M_{<u_3,v_{11}>}$



Chosen Rel. Combination	$(< v_{11}, v_{14} >)$ *(Other combinations are going to be evaluated later by backtracking)*
Matched Relationships	$M_{rel} = \{M_{<r_1,s_1>}, M_{<r_2,s_2>}, M_{<r_3,s_3>}, M_{<r_4,s_4>}, M_{<r_5,s_5>}, M_{<r_6,s_6>}, M_{<r_7,s_7>}, M_{<r_8,s_8>}\}$ where r_i and s_i are as stated in prev. steps for $1 \leq i \leq 7$, $r_8 :< u_3, u_0 >$, $s_8 :< v_{11}, v_{14} >$
Matched Nodes	$M_{node} = \{M_{<u_1,v_6>}, M_{<u_0,v_{14}>}, M_{<u_4,v_{12}>}, M_{<u_3,v_{11}>}, M_{<u_2,v_5>}\}$
Stack	$S = \{M_{<u_0,v_{14}>}, M_{<u_4,v_{12}>}\}$

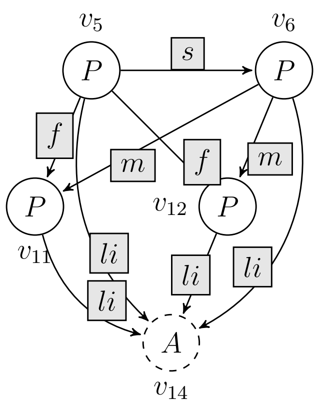
(h) Choosing a relationship combination to be matched

Figure 3.3: A sample piece of execution for the query and database graph given in Figure 3.1 (cont.)



Popped Match	$M_{<u_4,v_{12}>}$
Stack	$S = \{M_{<u_0,v_{14}>}\}$
Matched Nodes	$M_{node} = \{M_{<u_1,v_6>}, M_{<u_0,v_{14}>}, M_{<u_4,v_{12}>}, M_{<u_3,v_{11}>}, M_{<u_2,v_5>}\}$
Candidate Lists	$C_{<u_4,u_0>} = \{< v_{12}, v_{14} >\}$

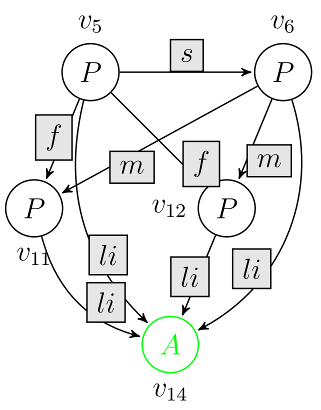
(i) Branching of $M_{<u_4,v_{12}>}$



Chosen Rel. Combination	$(< v_{12}, v_{14} >)$ *(Other combinations are going to be evaluated later by backtracking)*
Matched Relationships	$M_{rel} = \{M_{<r_1,s_1>}, M_{<r_2,s_2>}, M_{<r_3,s_3>}, M_{<r_4,s_4>}, M_{<r_5,s_5>}, M_{<r_6,s_6>}, M_{<r_7,s_7>}, M_{<r_8,s_8>}, M_{<r_9,s_9>}\}$ where r_i and s_i are as stated in prev. steps for $1 \leq i \leq 8$, $r_9 :< u_4, u_0 >$, $s_9 :< v_{12}, v_{14} >$
Matched Nodes	$M_{node} = \{M_{<u_1,v_6>}, M_{<u_0,v_{14}>}, M_{<u_4,v_{12}>}, M_{<u_3,v_{11}>}, M_{<u_2,v_5>}\}$
Stack	$S = \{M_{<u_0,v_{14}>}\}$

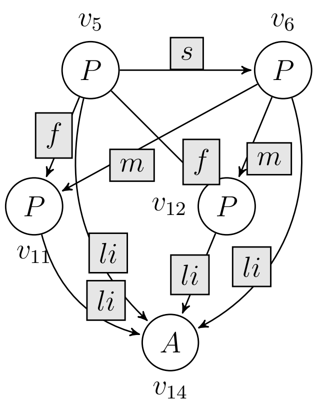
(j) Choosing a relationship combination to be matched

Figure 3.3: A sample piece of execution for the query and database graph given in Figure 3.1 (cont.)



Popped Match	$M_{<u_0,v_{14}>}$
Stack	$S = \emptyset$
Matched Nodes	$M_{node} = \{M_{<u_1,v_6>}, M_{<u_0,v_{14}>}, M_{<u_4,v_{12}>}, M_{<u_3,v_{11}>}, M_{<u_2,v_5>}\}$
Candidate Lists	There does not exist any non-matched relationship.

(k) Branching of $M_{<u_0,v_{14}>}$



Matched Relationships	$M_{rel} = E_q$
Matched Nodes	$M_{node} = V_q$
Exact Matches	$M = \{(M_{node}, M_{rel})\}$
Stack	$S = \emptyset$
Next Actions	Backtrack, Choose not-tried candidates, Apply the same procedure.

(l) Saving the found exact match

Figure 3.3: A sample piece of execution for the query and database graph given in Figure 3.1 (cont.)

3.2 Algorithm Complexity

In this section, running time complexity (computational complexity) analysis and space complexity analysis of BB-Graph are explained. All the parameters used in the analyses are given in Table 3.1.

3.2.1 Time Complexity

BB-Graph starts with the filtering part. It tries to find candidates for the start query node by filtering the database nodes with respect to their labels, adjacent relationships and property values (if exists). $FilterByLabel$ stage (Algorithm 2) takes $\mathcal{O}(1)$

Table 3.1: Explanation of the parameters used in the complexity analyses of BB-Graph

$\boldsymbol{\vert V_q \vert}$:	number of nodes (vertices) in query graph q
$\boldsymbol{\vert V_g \vert}$:	number of nodes (vertices) in database graph g
$\boldsymbol{\vert E_q \vert}$:	number of relationships (edges) in query graph q
$\boldsymbol{deg_q^{max}}$:	maximum of the node degrees in query graph q
$\boldsymbol{deg_g^{max}}$:	maximum of the node degrees in database graph g
$\boldsymbol{\rho_q^{max}}$:	maximum of the number of node properties in query graph q

since all nodes are kept with respect to their labels in Neo4j by dafault. In the worst case, all database nodes may have the labels of the start node, which means there exist $|V_g|$ number of candidate nodes that are going to be filtered with respect to their adjacent relationships. Therefore, $FilterByRelationships$ (Algorithm 3) stage takes $\mathcal{O}(|V_g| \times deg_q^{max})$. Similarly, in the worst case, all database nodes satisfy the adjacent relationship conditions and node property conditions. Therefore, $FilterByProperty$ stage (Algorithm 4) takes $\mathcal{O}(|V_g| \times \rho_q^{max})$. Totally, filtering part takes $\mathcal{O}(|V_g| \times (deg_q^{max} + \rho_q^{max})|)$. In the end of filtering part, there may be $|V_g|$ number of candidates for the start node in the worst case.

For each start node candidate the branching procedure is applied through Transition State. For that reason, the complexity of this part is going to be $|V_g| \times \mathcal{O}(branching)$. In the branching procedure (Algorithm 6), initially, all non-matched relationships of the current query node are detected and for each of them candidate relationship sets are constructed. If we assume that there are x_i many number of non-matched relationships for the current node in the depth i of the recursive process, since FINDCANDIDATERELATIONSHIPS (Algorithm 7) takes $\mathcal{O}(1)$, complexity of candidate relationship set construction becomes $\mathcal{O}(x_i)$. For each non-matched relationship, there can be at most deg_g^{max} many number of candidate relationships (If there are more than one different type of non-matched relationships of the current node, then it is certain that the candidate set size is less than deg_g^{max}. Nevertheless, we take deg_g^{max} as an upper bound for the candidate set size of each non-matched query relationship). This means, there occurs $(deg_g^{max})^{(x_i)}$ many number of different combinations of relationship matching for a query node. Since a query node can have at most deg_q^{max} non-matched relationships, the upper bound for number of combinations of relationship matching becomes $(deg_g^{max})^{(deg_q^{max})}$. At each candidate relationship selection, it is checked whether there occurs any conflict with the match through ISMATCHVALID function (Algorithm 8). In ISMATCHVALID, end nodes of the selected candidate relationship are compared with the already matched query and database nodes. After that, the filtering procedure is applied to the end nodes. For that reason, complexity of this function becomes $\mathcal{O}(|V_q| + deg_q^{max} + \rho_q^{max})$. As a result, at each depth of branching procedure, the cost becomes $\mathcal{O}((deg_g^{max})^{(deg_q^{max})} \times (|V_q| + deg_q^{max} + \rho_q^{max} + x_i))$. Since the maximum depth of the recursive branching can equal to the number of query nodes, $|V_q|$, and the summation of x_is is the number of query relationships, $|E_q|$, the overall complexity of branching procedure becomes $\mathcal{O}((deg_g^{max})^{(deg_q^{max})^{|V_q|}} \times (|V_q| + deg_q^{max} + \rho_q^{max}))$ (Actually it is $\mathcal{O}((deg_g^{max})^{(deg_q^{max})^{|V_q|}} \times (|V_q| + deg_q^{max} + \rho_q^{max}) + (deg_g^{max})^{(deg_q^{max})} \times |E_q|)$. However, the part $\mathcal{O}((deg_g^{max})^{(deg_q^{max})} \times |E_q|)$ is negligible.).

To conclude, the computational cost of BB-Graph equals to the summation of cost of filtering part and cost of branching part, which is, $\mathcal{O}(|V_g| \times (deg_q^{max} + \rho_q^{max})|) + \mathcal{O}(|V_g| \times (deg_g^{max})^{(deg_q^{max})^{|V_q|}} \times (|V_q| + deg_q^{max} + \rho_q^{max}))$. Since the cost of the filtering part is negligible when it is compared to the cost of the branching part, the runnin time complexity of BB-Graph becomes $\boldsymbol{\mathcal{O}(|V_g| \times (deg_g^{max})^{(deg_q^{max})^{|V_q|}} \times (|V_q| + deg_q^{max} + \rho_q^{max}))}$.

3.2.2 Space Complexity

For the filtering part, BB-Graph uses a list to hold the candidate database nodes for the start node. Since there can be maximum $\boldsymbol{|V_g|}$ number of candidates for the start node (in other words all the database nodes), there is a need for a list of size $\boldsymbol{|V_g|}$ in the worst case. Also, for each query relationship, there is constructed a candidate relationship set. Since there can be maximum $\boldsymbol{deg_g^{max}}$ number of candidates for a query relationship, size of a constructed candidate set becomes $\boldsymbol{deg_g^{max}}$. The case in which the recursive computation reaches to the deepest value is the worst case requiring the largest storage for the sets of candidate relationships. Since in the highest depth of the recursion, all of the query relationships are going to have been put in the process, the candidate relationship sets consume $|E_q| \times deg_g^{max}$ units of space at most. When the algorithm backtracks, the space containing candidate relationship sets constructed at that depth is released. Therefore $|E_q| \times deg_g^{max}$ is the maximum value of storage needed by them. Additionally, there exists the global storages consuming some memory. There are used two global hashing maps to hold the already matched query graph and database graph items; one map for already-matched-nodes and one map for already-matched-relationships. These two maps include the information of the fact that which query node or relationship was matched with which database node or relationship, respectively, that is, they can consume at most $2 \times |V_q|$ and $2 \times |E_q|$ units of storage, respectiely. Lastly, the stack which is used to hold not-yet-branched node matches needs $2 \times |V_q|$ units of storage, that is the space required in the case that start node is neighbour to all the remaining query nodes, and as a result, all of the query nodes are matched at one time and pushed into the stack. Throughout the whole execution of BB-Graph, all the other variables comsume quite a little space and can be neglected. Consequently, the space complexity of BB-Graph equals to the summation of all the mentioned values in the worst case, which is $\mathcal{O}(|V_g| + |V_q| + |E_q| \times deg_g^{max})$.

CHAPTER 4

EXPERIMENTAL WORK

In this chapter, the performances of BB-Graph, Cypher and GraphQL are compared on a big Neo4j directed graph database including 70 millions of nodes, and anothor two databases including 100 thousands and 45 thousands of nodes. BB-Graph and GraphQL were implemented in Java and Cypher queries were executed inside a Java code.

4.1 Database

In this study, the Population Database supplied by Kale Yazılım was used. Population Database consists of real logical data based on people and family relationships among them (like mother, father, spouse, ... etc.). It also includes the personal information of people like their address, birth date, state register, ... etc. If this kind of data is kept on relational tables, while it is not a hard work to fetch the family relationship information just for one person, it becomes a big problem to find connections among many people or to deduce relations of so many nodes because of requiring costly join operations on many tables or an extensive search needing a large amount of labour and time. Also, for the detection of erroneous data, all the related registrations should be checked and this may be impossible due to incapabilities of relational databases. Therefore, to meet the needs of such a big data, a graph database system is more compatible than RDBMS. Additionally, for the experimental purposes, Bank Database supplied by Kale Yazılım and WorldCup Database, which is public , were used so that measuring the performances of all three algorithms on a smaller sized database. Bank Database consists of the branches of a bank and the relationships between accounts or credits of customers and the branches. WorldCup database includes the player, match, squad relationships of countries that joined in the World Cup Tournament in the past.

All of three databases are structured on Neo4j Database, consist of directed relationships and each node/relationship has exactly one label. In Table 4.1, features of Population Database, Bank Database and WorldCup Database are given in a detailed way. Population Database is actually a much bigger data set than the ones used in [11].

Table 4.1: General Features of Population, Bank and WorldCup Databases

	WorldCup	Bank	Population
Size	20 MiB	510.23 MiB	10.32 GiB
# of graphs	1	1	1
# of nodes	45348	105085	70422787
# of relationships	86577	107898	77163109
# of distinct node labels	12	15	14
# of distinct relationship types	17	18	18
Avg. # of labels per node	1	1	1

4.2 Queries

For the experiments, there were used 10 real-world queries for Population Database and 5 real-world queries for each of Bank and WorldCup Databases where each query have different number of nodes and relationships and also have different types of node labels and relationships. For BB-Graph and GraphQL, each query was given in BFS code format explained in Section 2.2. For Cypher tests, each query were written as a string and executed through Java. For each query, BB-Graph and Cypher experiments were repeated 10 times, and the averages of the elapsed times were taken. For GraphQL experiments, each query was executed once since it takes a long time. Also, in GraphQL experiments, refinement-level was adjusted to 1 and refining process was repeated as many times as the number of nodes in a query graph.

4.3 Setup

All the experiments were conducted on the same PC with Intel Octa Core 2.27GHz, 8 GB of main memory, and 100 GB hard disk, running Debian GNU, Linux 7.8 (wheezy). BB-Graph and GraphQL were implemented in Java on Eclipse. The Neo4j version used is 2.3.1. In the experiments, all the exact matches were found in one time without any break. To compare the algorithm performances, total elapsed time were calculated.

4.4 Experimental Evaluation

In this section, the experimental results obtained by executing different types of queries on WorldCup Database, Bank Database and Population Database are given.

4.4.1 WorldCup Database

Table 4.2: Relationship abbreviations for WorldCup Database

N_S	NAMED_SQUAD	P_A_T	PLAYED_AT_TIME
I_S	IN_SQUAD	P_I	PLAYED_IN
ST	STARTED	C_M	CONTAINS_MATCH
SU	SUBSTITUTE	H_T	HOME_TEAM
I_M	IN_MATCH	A_T	AWAY_TEAM
S_G	SCORED_GOAL		

- **Query-1** : Players who join squad of different countries

  Figure 4.1 shows the graph of query-1 executed in WorldCup Database. It is a path query consisting of 5 nodes and 4 relationships. The results obtained with

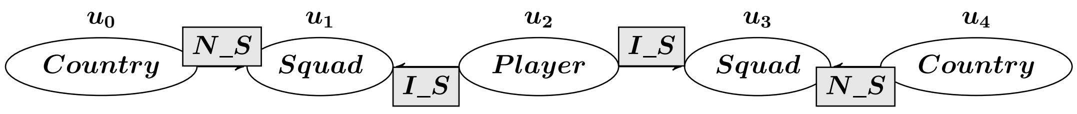


Figure 4.1: Graph of Query-1 in WorldCup Database

Cypher, GraphQL and BB-Graph are shown in Table 4.3. It is clearly seen that BB-Graph has the best performance among all while GraphQL is the slowest.

Table 4.3: Results for Query-1 in WorldCup Database

Cypher	GraphQL	BB-Graph	BB-Graph Matching Order
1629 ms	28538 ms	451 ms	u_0, u_1, u_2, u_3, u_4

- **Query-2** : Players who take role as both substitute and active (STARTED) in the same match

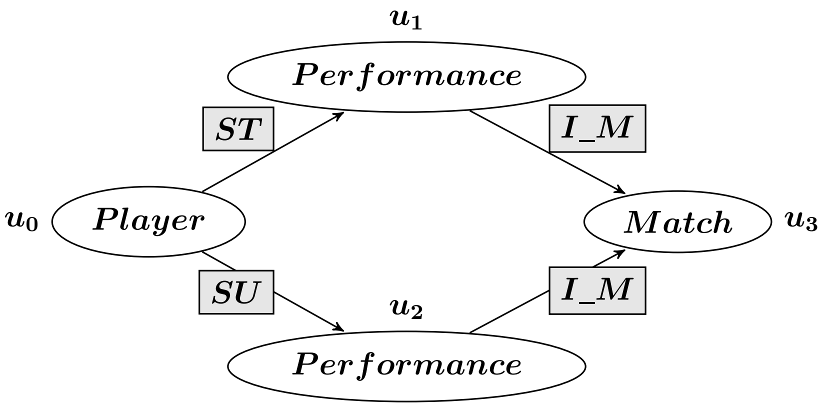


Figure 4.2: Graph of Query-2 in WorldCup Database

Figure 4.2 shows the graph of query-2 executed in WorldCup Database. It is a cyclic query consisting of 4 nodes and 4 relationships. In this query, the computationally exhaustive part for BB-Graph occurs during the match of the last node, which is u_1. Since it is the connective node making u_0 - u_2 - u_3 path a cycle, the candidate node v_1 for u_1 must be adjacent to v_0 where $M_{<u_0,v_0>} \in M_{node}$ in addition to its adjacency with v_3 where $M_{<u_3,v_3>}$. The algorithm performances are given in Table 4.4. BB-Graph has performed the best whereas GraphQL is the one performing the worst.

Table 4.4: Results for Query-2 in WorldCup Database

Cypher	GraphQL	BB-Graph	BB-Graph Matching Order
10123 ms	1112510 ms	1949 ms	u_0, u_2, u_3, u_1

- **Query-3** : Matches between the same countries occured in different world cups

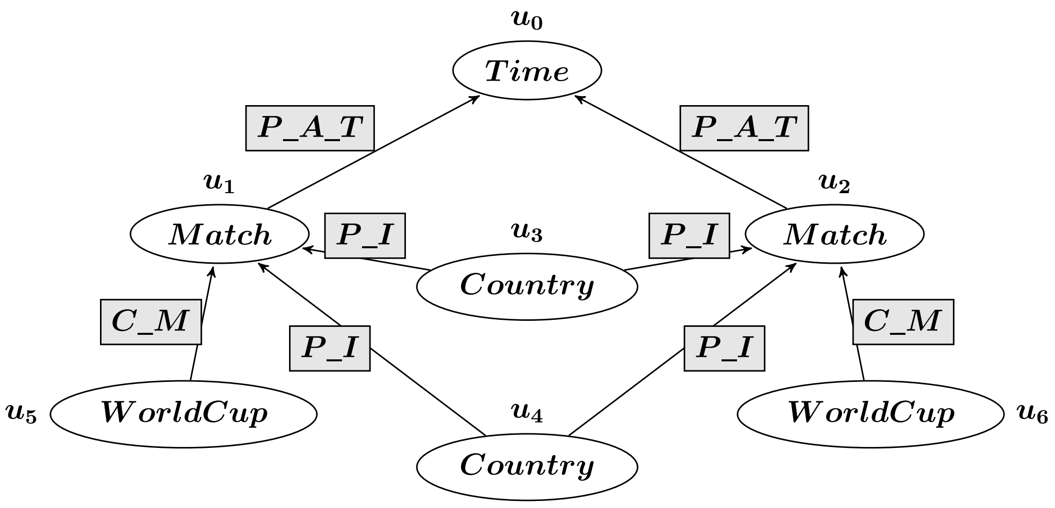


Figure 4.3: Graph of Query-3 in WorldCup Database

Figure 4.3 shows the graph of query-3 executed in WorldCup Database. It consists of 7 nodes and 8 relationships. This query consists of 2 important cycles which are (u_0 - u_1 - u_3 - u_2 - u_0) and (u_1 - u_3 - u_2 - u_4 - u_1). For this type of query, BB-Graph has again the best performance among the others 4.5; however it performs a bit worse than it did for query-2.

Table 4.5: Results for Query-3 in WorldCup Database

Cypher	GraphQL	BB-Graph	BB-Graph Matching Order
7851 ms	17995 ms	3951 ms	$u_0, u_2, u_6, u_4, u_3, u_1, u_5$

- **Query-4** : Cases at which two countries played at least 2 matches (as away team in one and home team in the other) in the same world cup and the same player scored at least 1 goal in both matches

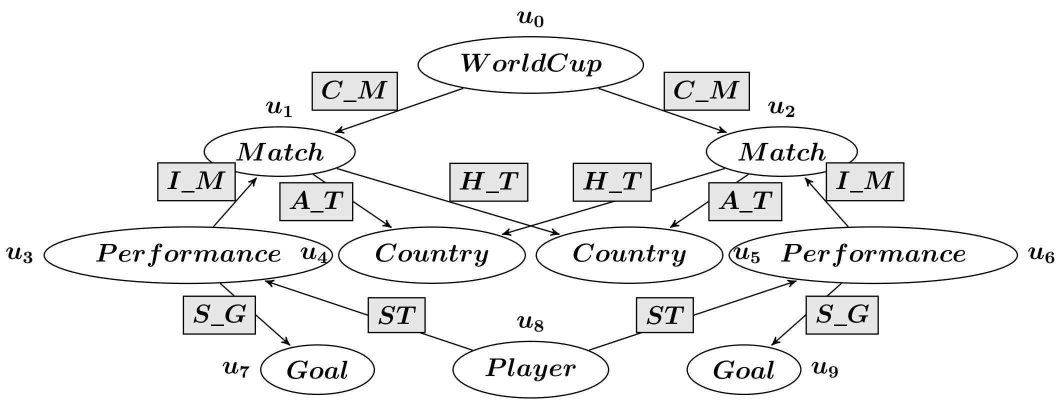


Figure 4.4: Graph of Query-4 in WorldCup Database

Figure 4.4 shows the graph of query-4 executed in WorldCup Database. It has 10 nodes and 12 relationships. This query includes many important cycles like (u_0 - u_1 - u_4 - u_2 - u_0), (u_0 - u_1 - u_5 - u_2 - u_0), (u_1 - u_3 - u_8 - u_6 - u_2 - u_4 - u_1) and (u_1 - u_3 - u_8 - u_6 - u_2 - u_5 - u_1). The performance results are shown in Table 4.6. This time, GraphQL shows very close performance to BB-Graph and Cypher's performance falls behind them.

Table 4.6: Results for Query-4 in WorldCup Database

Cypher	GraphQL	BB-Graph	BB-Graph Matching Order
26711 ms	8194 ms	7954 ms	$u_0, u_2, u_6, u_8, u_3, u_7, u_9, u_5, u_4, u_1$

- **Query-5** : Players who take role in any match at least 3 world cups

  Figure 4.5 shows the graph of query-5 executed in WorldCup Database. It is tree-type of query such that the root mode u_0 has 3 branches where each of them is a path. This query consists of 10 nodes and 9 relationships. For this query, GraphQL could not give all the matching results in a reasonable time. In Table 4.7, it is seen that Cypher performs better than BB-Graph.

Table 4.7: Results for Query-5 in WorldCup Database

Cypher	GraphQL	BB-Graph	BB-Graph Matching Order
26911 ms	> 30 mins	35796 ms	$u_0, u_3, u_6, u_9, u_2, u_5, u_8, u_1, u_4, u_7$

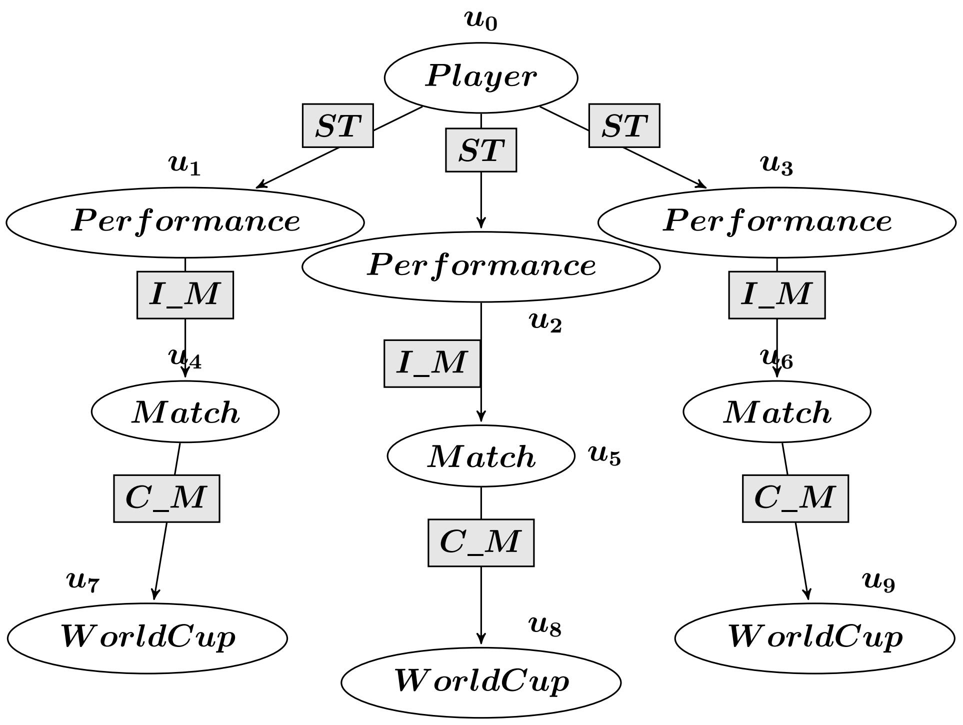


Figure 4.5: Graph of Query-5 in WorldCup Database

4.4.2 Bank Database

Table 4.8: Relationship abbreviations for Bank Database

L_I	LIVES_IN	H_C	HAS_CREDIT
H_A	HAS_ACCOUNT	B_O_C	BRANCH_OF_CREDIT
C_O_A	COUNTY_OF_ADDRESS	B_O_A	BRANCH_OF_ACCOUNT
C_O_B	COUNTY_OF_BRANCH	H_C_C	HAS_CREDIT_CARD

- **Query-1** : Customers who have an account from a branch brank which locates in the same county with the location of the customer address

  Figure 4.6 shows the graph of query-1 executed in Bank Database. It is a cyclic query consisting of 5 nodes and 5 relationships. In Table 4.9, it is seen that although BB-Graph's and Cypher's performances are very close, BB-Graph performs slightly better than Cypher. For this query, GraphQL's performance is quite lower than the other two.

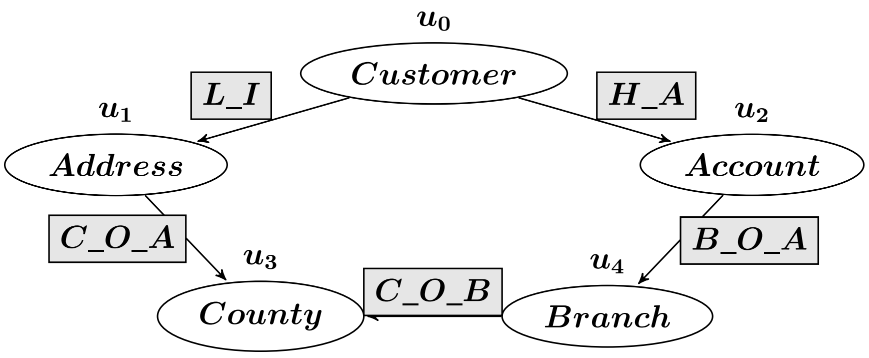


Figure 4.6: Graph of Query-1 in Bank Database

Table 4.9: Results for Query-1 in Bank Database

Cypher	GraphQL	BB-Graph	BB-Graph Matching Order
2193 ms	339038 ms	2071 ms	u_0, u_2, u_4, u_3, u_1

- **Query-2** : Customers who have at least 2 accounts from the same branch bank and at least 1 account from a different branch bank

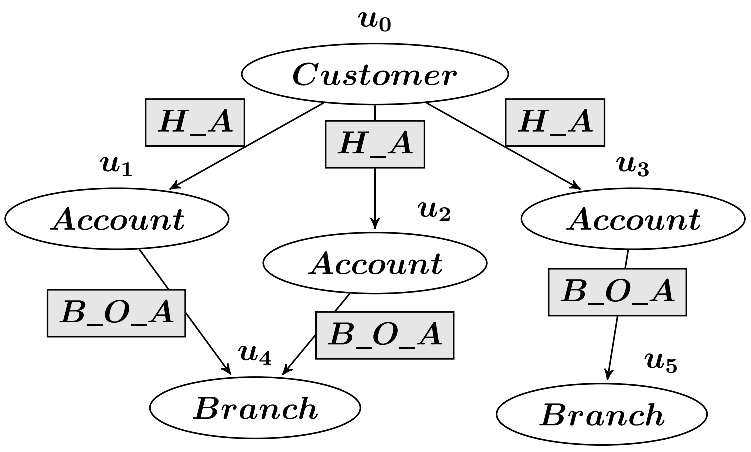


Figure 4.7: Graph of Query-2 in Bank Database

Figure 4.7 shows the graph of query-2 executed in Bank Database. It consists of 6 nodes and 6 relationships with 1 important cycle which is ($u_0 - u_1 - u_4 - u_2 - u_0$). Table 4.10 shows that BB-Graph performs the best whereas GraphQL performs the worst.

Table 4.10: Results for Query-2 in Bank Database

Cypher	GraphQL	BB-Graph	BB-Graph Matching Order
3692 ms	630825 ms	2784 ms	$u_0, u_3, u_5, u_2, u_4, u_1$

- **Query-3** : Customers who have an account and got credit from the same branch bank

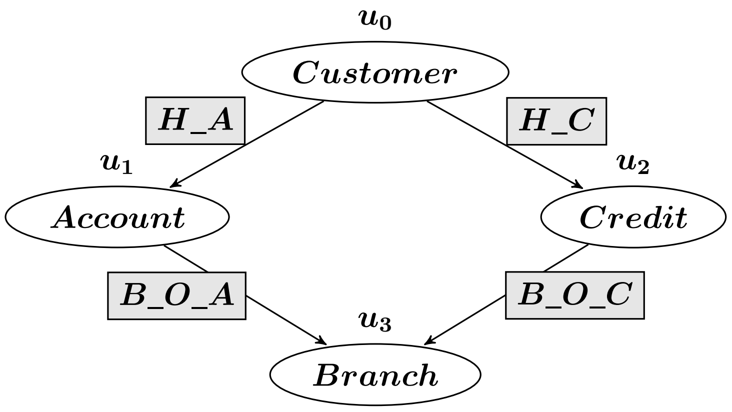


Figure 4.8: Graph of Query-3 in Bank Database

Figure 4.8 shows the graph of query-3 executed in Bank Database. It is a cyclic query consisting of 4 nodes and 4 relationships. Similar to query-1 results, Cypher and BB-Graph shows very close performances for this query also 4.11.

Table 4.11: Results for Query-3 in Bank Database

Cypher	GraphQL	BB-Graph	BB-Graph Matching Order
1593 ms	238963 ms	1435 ms	u_0, u_2, u_3, u_1

- **Query-4** : Cases consisting of 3 customers x,y,z who satisfy the followings:
  - For the same branch bank A, x has an account and y got a credit
  - For the same branch bank B, y has an account and z got a credit
  - For the same branch bank C, z has an account and x got a credit
  - A, B and C are all different branch banks

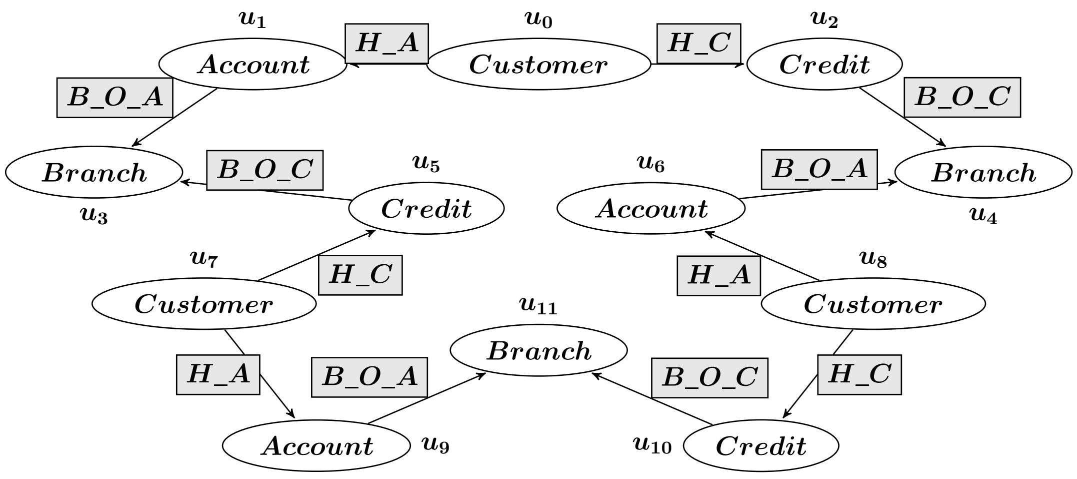


Figure 4.9: Graph of Query-4 in Bank Database

Figure 4.9 shows the graph of query-4 executed in Bank Database. It is a big cyclic query consisting of 12 nodes and 12 relationships. This time, contrary to query-1 and query-3 which are cyclic also, BB-Graph shows quite better results than Cypher as it is seen in Table 4.12. For this query, GraphQL cannot complete its execution in a reasonable time.

Table 4.12: Results for Query-4 in Bank Database

Cypher	GraphQL	BB-Graph	BB-Graph Matching Order
226830 ms	> 30 mins	16025 ms	$u_0, u_2, u_4, u_6, u_8, u_{10}, u_{11}, u_9, u_7, u_5, u_3, u_1$

- **Query-5** : Customers who has account from the same branch bank and live in the same county, and one of them has at least 1 credit card

  Figure 4.10 shows the graph of query-5 executed in Bank Database. It is a query including 9 nodes and 9 relationships with 1 cycle. Table 4.13 shows that, BB-Graph performs better than Cypher for this query whereas GraphQL cannot perform in an acceptable time.

Table 4.13: Results for Query-5 in Bank Database

Cypher	GraphQL	BB-Graph	BB-Graph Matching Order
8966 ms	> 30 mins	6093 ms	$u_0, u_3, u_5, u_7, u_8, u_6, u_4, u_2, u_1$

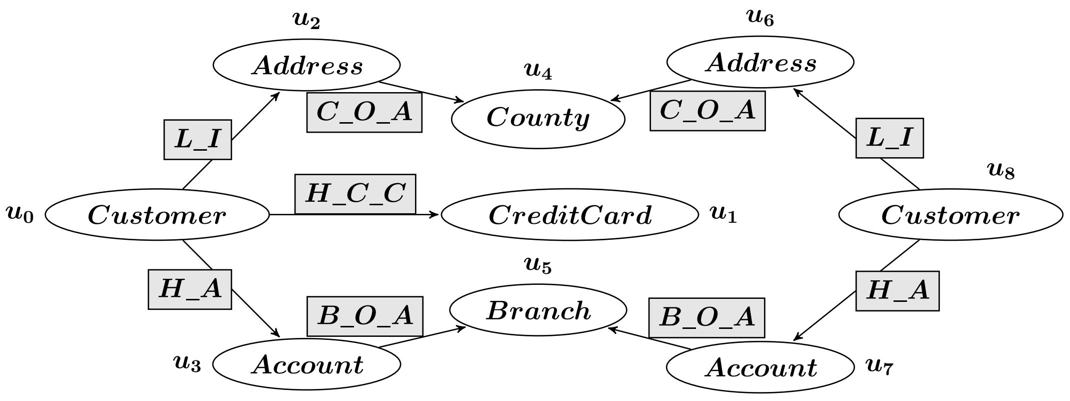


Figure 4.10: Graph of Query-5 in Bank Database

4.4.3 Population Database

Table 4.14: Relationship abbreviations for Population Database

M	MOTHER	L_I	LIVES_IN
F	FATHER	M_O	MEMBER_OF
S	SPOUSE	Of	FLAT_OF_APARTMENT
O_S	OLD_SPOUSE		

- **Query-1** : Families with at least 3 children and all (mother + father + 3 children) living in the same address

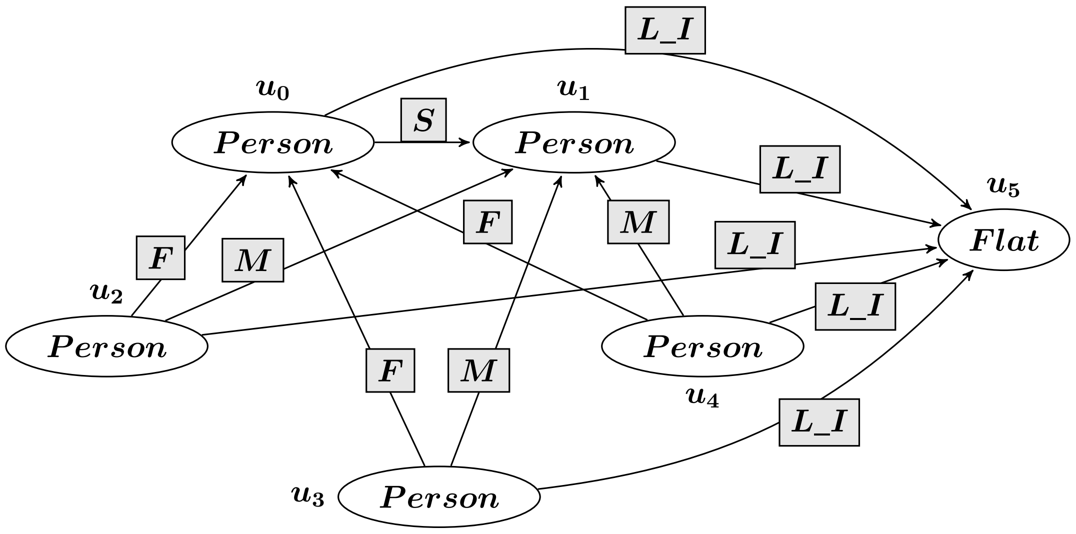


Figure 4.11: Graph of Query-1 in Population Database

Figure 4.11 shows the graph of query-1 executed in Population Database. It is a highly connected query graph where degree of each node is at least 3. It includes 6 nodes and 12 relationships with many cycles. Table 4.15 shows that BB-Graph has a better performance than Cypher.

Table 4.15: Results for Query-1 in Population Database

Cypher	GraphQL	BB-Graph	BB-Graph Matching Order
189940 ms	> 30 mins	138193 ms	$u_0, u_4, u_3, u_2, u_5, u_1$

- **Query-2** : Extended families consisting of mother, father, son and son's wife and all living in the same address

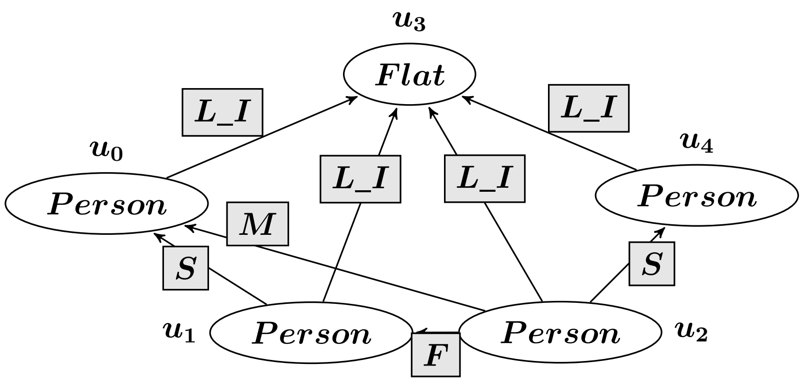


Figure 4.12: Graph of Query-2 in Population Database

Figure 4.12 shows the graph of query-2 executed in Population Database. It contains 5 nodes and 7 relationships. Table 4.16 shows that Cypher and BB-Graph have close performances; however, BB-Graph performs slightly better than Cypher.

Table 4.16: Results for Query-2 in Population Database

Cypher	GraphQL	BB-Graph	BB-Graph Matching Order
24289 ms	> 30 mins	22354 ms	u_0, u_2, u_4, u_1, u_3

- **Query-3** : Married couples whose mothers are sisters

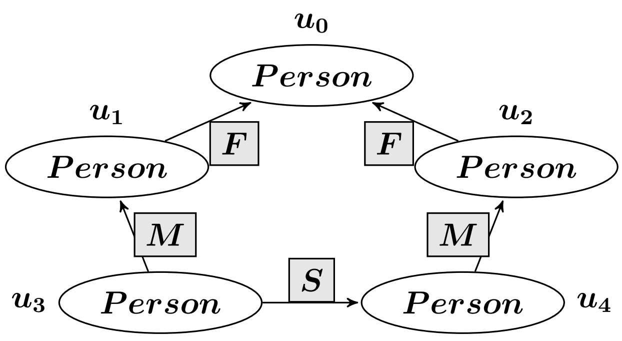


Figure 4.13: Graph of Query-3 in Population Database

Figure 4.13 shows the graph of query-3 executed in Population Database. It is a cyclic query consisting of 5 nodes and 5 relationships. We have done BB-Graph experiments with 2 different matching orders for this query. For both matching orders, we have obtained very different results 4.17. Although the performance is almost doubled in the second version of matching, Cypher shows better performance than BB-Graph for this query.

Table 4.17: Results for Query-3 in Population Database

Cypher	GraphQL	BB-Graph	BB-Graph Matching Order
14865 ms	> 30 mins	33374 ms	u_0, u_2, u_4, u_3, u_1
		17352 ms	u_3, u_1, u_0, u_2, u_4

- **Query-4** : Housemates whose fathers are brothers

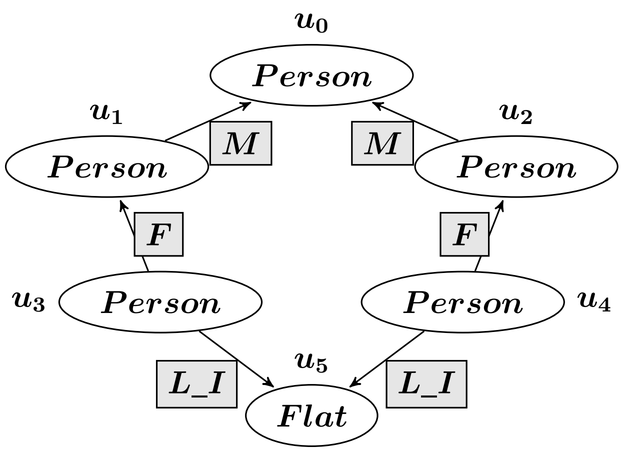


Figure 4.14: Graph of Query-4 in Population Database

Figure 4.14 shows the graph of query-4 executed in Population Database. It is a cyclic query with 6 nodes and 6 relationships. Although it has very similar structural features with query-4, performance results of both Cypher and BB-Graph for this query quite differ than the results obtained in query-3. This is just because in Population Database the number of nodes for each label changes quitely. In Table 4.19, it is observed that for the first matching order, the performance of BB-Graph is slightly worse than Cypher's whereas for the second matching order BB-Graph has performed twice as well the previous one and thus, the results have become much better than Cypher's results.

Table 4.18: Results for Query-4 in Population Database

Cypher	GraphQL	BB-Graph	BB-Graph Matching Order
92444 ms	> 30 mins	101401 ms	$u_0, u_2, u_4, u_5, u_3, u_1$
		54552 ms	$u_1, u_3, u_5, u_4, u_2, u_0$

- **Query-5** : Families in which a man, his wife, his at least 2 children; one from his ex-wife and the other from his wife, are all living in the same address

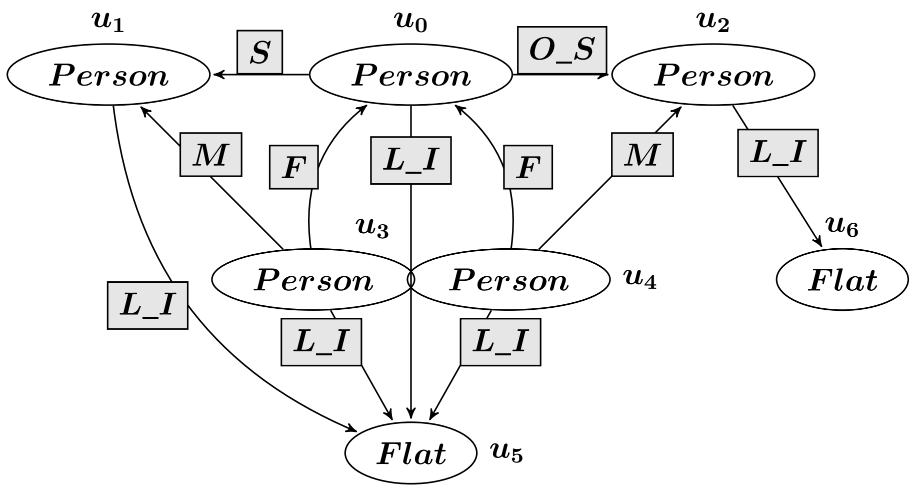


Figure 4.15: Graph of Query-5 in Population Database

Figure 4.15 shows the graph of query-5 executed in Population Database. It has 7 nodes and 11 relationships. Table 4.19 shows that BB-Graph has a better performance than Cypher.

Table 4.19: Results for Query-5 in Population Database

Cypher	GraphQL	BB-Graph	BB-Graph Matching Order
14465 ms	> 30 mins	8554 ms	$u_0, u_4, u_2, u_6, u_3, u_5, u_2, u_1$

- **Query-6** : Families in which a woman, her child from her ex-husband, her mother and father are all living in the same address

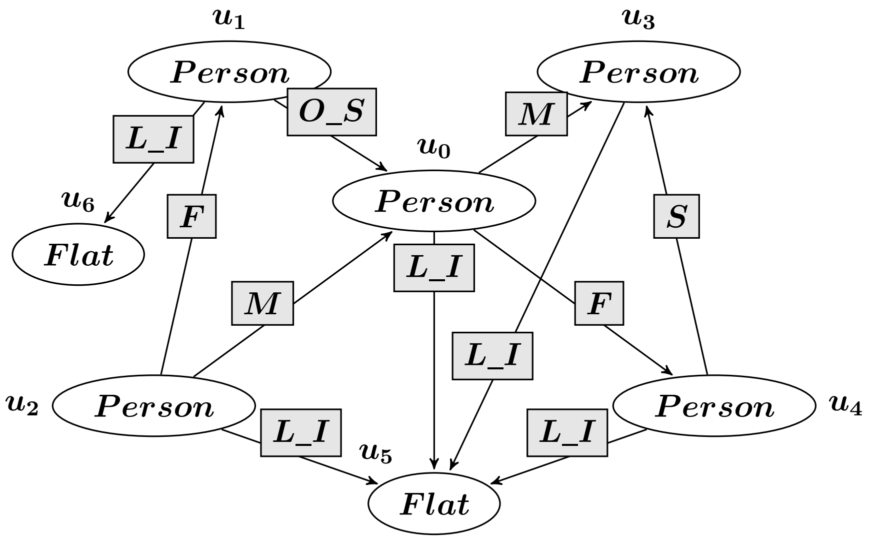


Figure 4.16: Graph of Query-6 in Population Database

Figure 4.16 shows the graph of query-6 executed in Population Database. It consists of 7 nodes and 11 relationships similar to query-5. However, this time, although results obtained with both are close to eachother, Cypher performs slightly better than BB-Graph 4.20. Here, the performance difference between BB-Graph experiments with query-5 and query-6 is caused by the different neighbourhoods of the same node labels. Since the neighbourhood of a node, in other words types of adjacent relationship and degree, specializes it, how much a node is well-described, the number of false candidates for that node decreases in the same manner.

Table 4.20: Results for Query-6 in Population Database

Cypher	GraphQL	BB-Graph	BB-Graph Matching Order
13713 ms	> 30 mins	14036 ms	$u_0, u_2, u_1, u_6, u_5, u_4, u_3$

- **Query-7** : Couples whose state registers are different

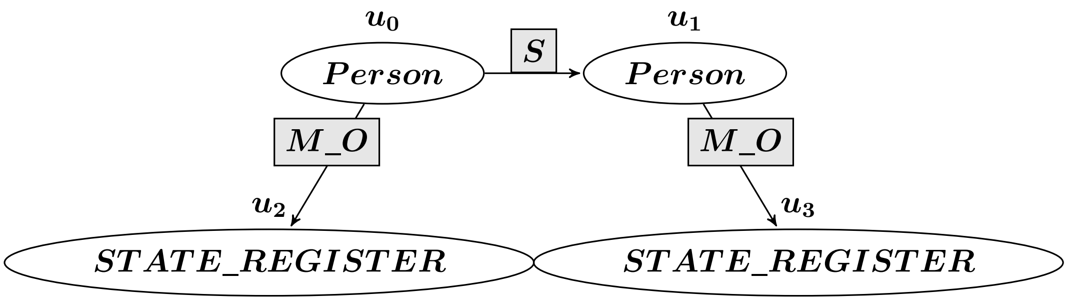


Figure 4.17: Graph of Query-7 in Population Database

Figure 4.17 shows the graph of query-7 executed in Population Database. It is a short path query including 4 nodes and 3 relationships. For such a query, Table 4.21 shows that Cypher's performance results are better than the both performance results that BB-Graph produced with different matching orders.

Table 4.21: Results for Query-7 in Population Database

Cypher	GraphQL	BB-Graph	BB-Graph Matching Order
12541 ms	> 30 mins	24103 ms	u_0, u_2, u_1, u_3
		18984 ms	u_2, u_0, u_1, u_0

- **Query-8** : The families living in different flats; A, B, C, of the same apartment such that:
  - The grandmother x and grandfather y lives in flat A
  - Son of x and y lives with his wife and his 2 children in flat B
  - Daughter of x and y lives with her husband and her child in flat C

Figure 4.18 shows the graph of query-8 executed in Population Database. It is a highly complex structured query consisting of 13 nodes and 25 relationships

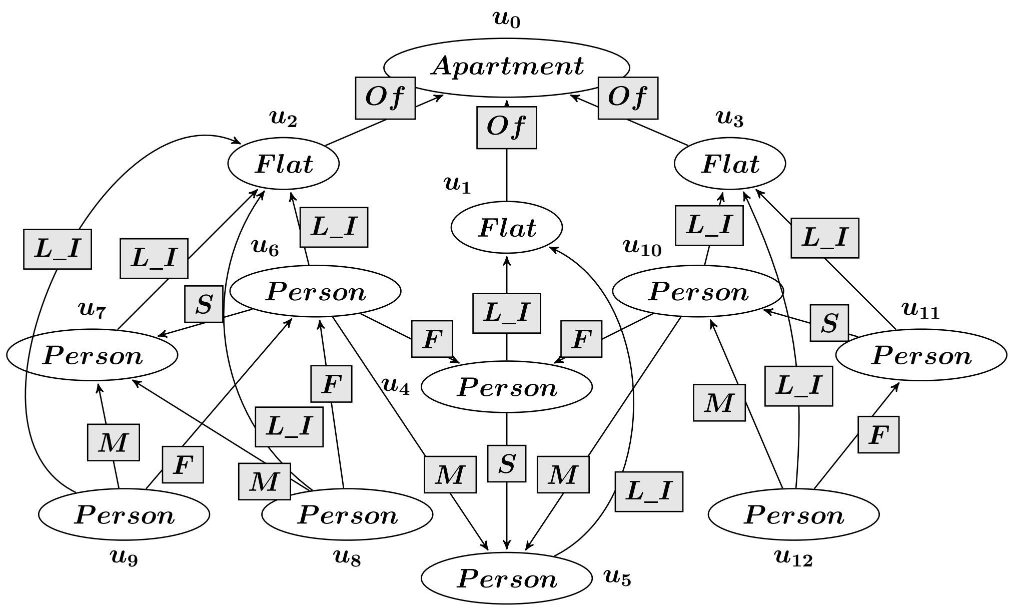


Figure 4.18: Graph of Query-8 in Population Database

with many cycles. As it is seen in Table 4.22, Cypher fails at returning all the matches in a reasonable time whereas BB-Graph returns all of the results in approximately 8 minutes.

Table 4.22: Results for Query-8 in Population Database

Cypher	GraphQL	BB-Graph	BB-Graph Matching Order
> 30 mins	> 30 mins	502721 ms	$u_0, u_3, u_{12}, u_{11}, u_{10}, u_5, u_6, u_9, u_8, u_7, u_4, u_2, u_1$

- **Query-9** : Fathers and their sons along 8-degree-generation

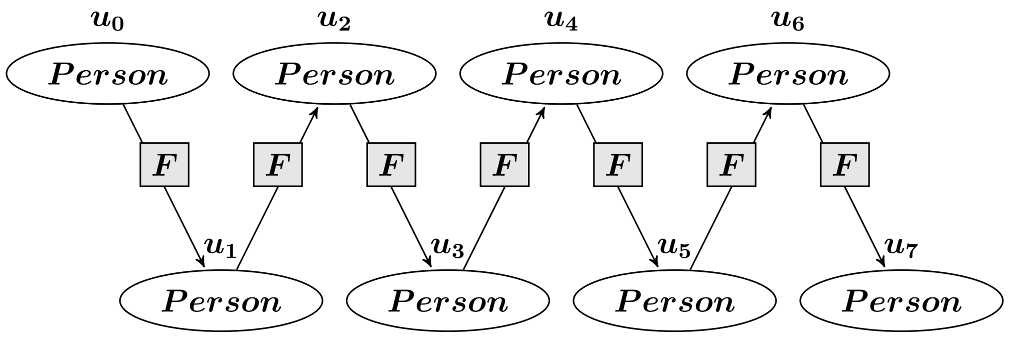


Figure 4.19: Graph of Query-9 in Population Database

Figure 4.19 shows the graph of query-9 executed in Population Database. It is a path query with 8 nodes and 7 relatinships. We have run BB-Graph with 2 different matching orders for this query. While BB-Graph performs worse than Cypher with the first mathching order, it shows a much higher performance with the second matching order and performs quite better than Cypher 4.23. This is caused by the smaller number of candite nodes occuring with the second matching order due to the semantic features of Population Database; that is, almost every node with label $\boldsymbol{Person}$ has an outgoing relationship of type $\boldsymbol{F}$, but does not have an incoming relationship of the same type.

Table 4.23: Results for Query-9 in Population Database

Cypher	GraphQL	BB-Graph	BB-Graph Matching Order
26204 ms	> 30 mins	41421 ms	$\boldsymbol{u_0, u_1, u_2, u_3, u_4, u_5, u_6, u_7}$
		2204 ms	$\boldsymbol{u_7, u_6, u_5, u_4, u_3, u_2, u_1, u_0}$

- **Query-10** : Twins who live in different flats of the same apartment which is different from the apartment where their parents live

  Figure 4.20 shows the graph of query-10 executed in Population Database. It is a query containing 10 nodes and 14 relationships with many cycles.For this query, it is seen in Table 4.23 that BB-Graph shows a better performance than Cypher.

Table 4.24: Results for Query-10 in Population Database

Cypher	GraphQL	BB-Graph	BB-Graph Matching Order
226770 ms	> 30 mins	73435 ms	$\boldsymbol{u_0, u_3, u_7, u_9, u_6, u_5, u_2, u_4, u_8, u_1}$

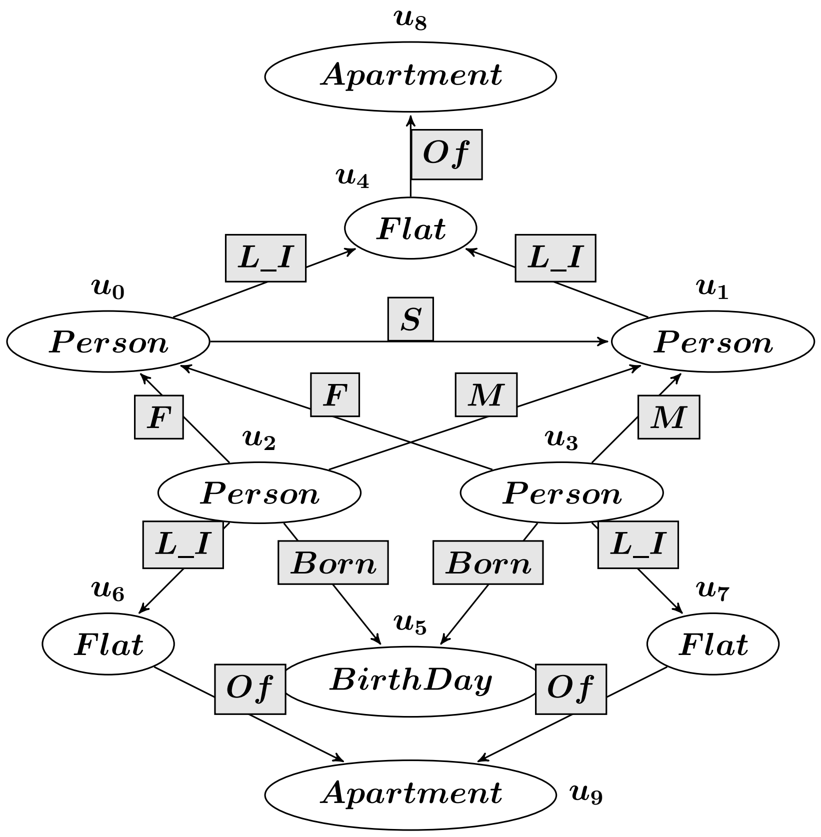


Figure 4.20: Graph of Query-10 in Population Database

CHAPTER 5

CONCLUSION & FUTURE WORK

5.1 Conclusion

In this thesis, a new algorithm, BB-Graph, for Subgraph Isomorphism Problem is introduced. Similar to existing approaches in literature, BB-Graph uses branch-and-bound technique to match each query node and relationship with its candidate and backtracks for trying the other possible candidates. However, contrary to the current algorithms that try to find candidates of each query node all across the database graph, after matching the first query node, BB-Graph searches candidates for other query nodes and relationships in local regions of the first matched database node. Our experiments conducted with different types of queries on 3 different real-world databases, Population Database, Bank Database and WorldCup Datbase, show that BB-Graph has the best performance among GraphQL and Cypher for most of the cases. According to our experimental results, GraphQL is not capable of querying in very large databases like Population Database since it couldn't complete its execution in a reasonable time. Although GraphQL has showed very close performance to BB-Graph for one of the queries in WorldCup Database, generally it performed worse than Cypher and BB-Graph due to its matching strategy of the irrelevant candidates from all over the whole database. When it comes to Cypher and BB-Graph, the experimental results show that BB-Graph has much better performance than Cypher for most of the time. Moreover, for the cases BB-Graph has performed worse than Cypher, it could achieve completing its execution in very close length of duration with Cypher. Therefore, we strongly suggest our algorithm, BB-Graph, for both large and small databases and for different types of queries because of its high performance compared to Cypher and GraphQL.

5.2 Future Work

According to our experimental results, we have seen the performance of the algorithms that we implemented, especially our algorithm BB-Graph, does not depend on just one factor like the number of nodes or relationships. Actually, many features such as frequency of query node labels and query relationship types in the database graph, number of cycles in query graph, structure of query graph (whether it is a path, a tree or something more complex) and also the semantic design of relationships like $\boldsymbol{1 - N}$ or $\boldsymbol{N - N}$ all affect the computation time of algorithms. Additionally, our experiments show that the order in which query nodes are matched also highly affect the algorithm performance since number of false cases can be eliminated earlier based on the situation appeared after matching some query node. For that reason, we are planning to improve the performance of our algorithm with some patches which design a query-dependent matching order by taking into consideration the query graph specifications mentioned above.

REFERENCES

[1] Bitnine Company Website, Relational Database vs Graph Database. `http://bitnine.net/rdbms-vs-graph-db/`. Accessed: Oct. 2016.

[2] A. Abboud, A. Backurs, T. D. Hansen, V. V. Williams, and O. Zamir. Subtree isomorphism revisited. arXiv preprint arXiv:1510.04622, 2015.

[3] S. Batra and C. Tyagi. Comparative analysis of relational and graph databases. International Journal of Soft Computing and Engineering (IJSCE), 2(2):509–512, 2012.

[4] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 26(10):1367–1372, 2004.

[5] R. Giugno and D. Shasha. Graphgrep: A fast and universal method for querying graphs. In Pattern Recognition, 2002. Proceedings. 16th International Conference on, volume 2, pages 112–115. IEEE, 2002.

[6] W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu. igraph: a framework for comparisons of disk-based graph indexing techniques. Proceedings of the VLDB Endowment, 3(1-2):449–459, 2010.

[7] H. He and A. K. Singh. Closure-tree: An index structure for graph queries. In Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on, pages 38–38. IEEE, 2006.

[8] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pages 405–418. ACM, 2008.

[9] V. Kolomičenko. Analysis and experimental comparison of graph databases. Master thesis, Charles University Department of Software Engineering, Prague, 2013.

[10] C. Lange, H. M. Sneed, and A. Winter. Comparing graph-based program comprehension tools to relational database-based tools. In Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on, pages 209–218. IEEE, 2001.

[11] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In Proceedings of the VLDB Endowment, volume 6, pages 133–144. VLDB Endowment, 2012.

[12] J. J. Miller. Graph database applications and concepts with neo4j. In Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA, volume 2324, 2013.

[13] A. Nayak, A. Poriya, and D. Poojary. Type of nosql databases and its comparison with relational databases. International Journal of Applied Information Systems, 5(4):16–19, 2013.

[14] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. Proceedings of the VLDB Endowment, 1(1):364–375, 2008.

[15] S. Srinivasa, M. Maier, M. R. Mutalikdesai, K. Gowrishankar, and P. Gopinath. Lwi and safari: A new index structure and query model for graph databases. In COMAD, pages 138–147, 2005.

[16] J. R. Ullmann. An algorithm for subgraph isomorphism. Journal of the ACM (JACM), 23(1):31–42, 1976.

[17] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins. A comparison of a graph database and a relational database: a data provenance perspective. In Proceedings of the 48th annual Southeast regional conference, page 42. ACM, 2010.

[18] D. W. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on, pages 976–985. IEEE, 2007.

[19] L. Wycislik and L. Warchal. A performance comparison of several common computation tasks used in social network analysis performed on graph and relational databases. In Man-Machine Interactions 3, pages 651–659. Springer, 2014.

[20] X. Yan and J. Han. Closegraph: mining closed frequent graph patterns. In Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, pages 286–295. ACM, 2003.

[21] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In Proceedings of the 2004 ACM SIGMOD international conference on Management of data, pages 335–346. ACM, 2004.

[22] S. Zhang, M. Hu, and J. Yang. Treepi: A novel graph indexing method. In Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on, pages 966–975. IEEE, 2007.

[23] S. Zhang, S. Li, and J. Yang. Gaddi: distance index based subgraph matching in biological networks. In Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, pages 192–203. ACM, 2009.

[24] P. Zhao and J. Han. On graph query optimization in large networks. Proceedings of the VLDB Endowment, 3(1-2):340–351, 2010.

[25] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: tree+ delta<= graph. In Proceedings of the 33rd international conference on Very large data bases, pages 938–949. VLDB Endowment, 2007.