BB-PLUS: AN EFFICIENT APPROACH FOR SUBGRAPH ISOMORPHISM PROBLEM IN BIG GRAPH DATABASES

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

BY

EZGI TAŞKOMAZ

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

JUNE 2019

Approval of the thesis:

BB-PLUS: AN EFFICIENT APPROACH FOR SUBGRAPH ISOMORPHISM PROBLEM IN BIG GRAPH DATABASES

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

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Halit Oğuztüzün
Head of Department, **Computer Engineering** _____

Prof. Dr. Adnan Yazıcı
Supervisor, **Computer Engineering, METU** _____

Examining Committee Members:

Prof. Dr. Ahmet Coşar
Computer Engineering, UTAA _____

Prof. Dr. Adnan Yazıcı
Computer Engineering, METU _____

Assoc. Prof. Dr. Ahmet Oğuz Akyüz
Computer Engineering, METU _____

Date:

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Ezgi Taşkomaz

Signature :

ABSTRACT

BB-PLUS: AN EFFICIENT APPROACH FOR SUBGRAPH ISOMORPHISM PROBLEM IN BIG GRAPH DATABASES

Taşkomaz, Ezgi
M.S., Department of Computer Engineering
Supervisor: Prof. Dr. Adnan Yazıcı

June 2019, 138 pages

Graph databases are flexible NoSQL databases used to efficiently store and query complex dataset. The problem of subgraph isomorphism, finding a pattern in a given graph, is one of the biggest problem of graph databases. Therefore, the goal of this study is to introduce a new approach called BB-Plus, which consists of heuristics to find best matching order using the volatility and size of the database, the type and size of the query as an input in order to improve the performance of the queries. BB-Plus approach trims candidate nodes at high level and effectively reduces the size of the problem. The approach is implemented using the Java programming language and graph data structures of Neo4j GDBMS and compared to the state-of-the-art subgraph isomorphism algorithms, namely BB-Graph, Cypher, DualIso, GraphQL, TurboIso and VF3 with three different dataset within the same programming environment. The results of the performance tests show that BB-Plus is an average on 10%, 37% and 4% faster than the other algorithms based on different queries in public WorldCup, Pokec and non-public Population dataset, respectively.

Keywords: Subgraph Isomorpishm Problem, Matching Order Selection, Graph Database, Neo4j

ÖZ

BB-PLUS: BÜYÜK ÇİZGE VERİTABANLARINDA ALTÇİZGE EŞYAPILILIK PROBLEMİNE ETKİN BİR YAKLAŞIM

Taşkomaz, Ezgi
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Adnan Yazıcı

Haziran 2019 , 138 sayfa

Çizge veritabanları, karmaşık veri setlerini daha etkin bir şekilde depolama ve sorgulamada kullanılan esnek NoSQL veritabanlarıdır. Altçizge eşyapılılık problemi yani verilen bir çizgede örüntülerin bulunması ise çizge veritabanlarındaki en büyük problemlerden biridir. Bu nedenle bu çalışmanın amacı, sorgunun performanasını artırmak için veritabanın büyüyklüğü, değişkenliği, sorgunun büyüklüğü ve tipini girdi olarak alan sezgisel yöntemler kullanarak en iyi eşleşen sırayı bulan BB-Plus adında yeni bir yaklaşım sunmaktır. BB-Plus, üst seviyelerde aday düğümlerin eler ve arama uzayının boyutunun düşürür. Yaklaşım, Java programlama dili ve Neo4j çizge veri yapılarını kullanılarak geliştirilmiştir ve aynı programlama ortamında üç farklı boyutta veri seti kullanılarak güncel altçizge eşyapılılık algoritmaları olan BB-Graph, Cypher, DualIso, GraphQl, TurboIso ve VF3 ile karşılaştırılmıştır. BB-Plus, farklı sorgular baz alındığında diğer algoritmalardan kullanıma açık WorldCup, Pokec ve kullanıma açık olmayan Population veritabanlarında sırasıyla ortalama %10, %37 ve %4 daha hızlı çalışmaktadır.

Anahtar Kelimeler: Altçizge Eşyapılılık Problemi, Eşleşen Sıra Seçimi, Çizge Veri tabanı, Neo4j

To my family and friends

ACKNOWLEDGMENTS

First of all, I would like to thank and express my special appreciation and gratitude to my thesis advisor, Prof. Dr. Adnan YAZICI, for his supervision, patience and encouragement throughout the research. He always keeps me motivated to explore new ideas and solutions in my research and contributed to this work with his great experience, knowledge and energy.

I would like to thank Merve Asiler for allowing me to improve her work. I wish to express my sincere gratitude to her for being always open to share her experience on the area, to guide me in exploring new ideas and never hesitate to spending time with me to improve my work.

In addition, I want to thank my company TÜBİTAK for supporting me to continue my studies. I would like to thank my colleagues in my project and especially to my old teammates, Anıl Özberk, Fatma Emül, Murat Burak Yıldırım and Nazlı Ece Uykur, who never stop supporting me and staying motivated at all times.

I would like to thank especially KALE YAZILIM A.Ş. for sharing their Population dataset which is very important for me to complete my experiments. I would like to thank Başar DALKILIÇ for his help and finding quick solutions to my problems with the dataset.

Finally, I would like to thank my family: my mother Yüksel TAŞKOMAZ, my father Erkan TAŞKOMAZ and my sister Özge TAŞKOMAZ. Words cannot express how grateful I am for their support, love and sacrifices that they have made throughout my life. They deserve a special thanks for their limitless support and encouragement. This work would not have been possible without them.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

DB	Database
GDBMS	Graph Database Management System
BB	Branch-and-Bound
DC	Degree Centrality
CC	Closeness Centrality
BC	Betweenness Centrality
EC	Eigenvector Centrality
CNS	Candidate Node Selection
CES	Candidate Edge Selection
DFS	Depth First Search
BFS	Breadth First Search
CR	Candidate Region
NEC	Neighborhood Equivalence Class
s.t.	Such That
i.e.	In Other Words
e.g.	For Example
query node/vertex	Node/Vertex In Query Graph
data node/vertex	Node/Vertex In Data Graph
query edge/relationship	Edge/Relationship In Query Graph
database edge/relationship	Edge/Relationship In Data Graph

CHAPTER 1

INTRODUCTION

Over the past decade, the amount of data collected has increased with technological developments and all research done in computer science. With social networks, e-commerce websites, web applications, bioinformatics, communication, etc., we produce "Big Data", which is complex, useful, structured/unstructured, very fast with questionable veracity [4].

With the rise of Big Data, NoSQL databases have been created to effectively manage Big Data in order to overcome limitations of the relational database model such as fixed schema or data with a consistent structure [5]. NoSQL databases, which are document, key-value, column store and graph databases, have their own advantages and disadvantages of solving different Big Data problems with their different data models [6].

Column stores (e.g., Cassandra, MariaDB, ClickHouse) are highly scalable, fast databases that store data by each column, unlike relational databases, which store data by each row. Therefore, they are good at aggregation queries, big-data analysis, and data mining applications. Document stores (e.g., MongoDB, CouchDB, Couchbase) are schemaless, flexible and highly available databases that can store, retrieve or manage large volumes of unstructured/semi-structured data in documents in XML, JSON, YAML, PDF etc. format and execute queries rapidly with their strong indexing capabilities. Key-value stores (e.g., Riak, DynamoDB, BerkeleyDB) are schemaless and capable of mass storage databases that keep data as key-value pairs as simple hash tables. Therefore, they can offer operational simplicity, fast lookups, and high concurrency. Additionally, they are good at horizontal scaling comparing to the other NoSQL databases. [7].

Graph databases (e.g., Neo4j, OrientDB) are flexible databases that use basic graph structures to store and query complex data sets, such as graphs [5]. They are good for frequently modified schemas, recursive queries, semantic search and queries with expensive join operations in complex and highly connected entities. However, GDMBSs are not as mature as relational database management systems. As a result, they do not have a feature-rich environment and a standard query language. In addition, the graph databases introduce high-memory consumption. Therefore, they should instead be used for more complex database applications rather than for very large sets. [8].

The subgraph isomorphism problem is an NP-Complete problem that can simply be described as detection of the patterns of a query graph in a data graph. The subgraph isomorphism is used in many areas like pattern recognition, computer vision, computer-aided design, image processing, biocomputing, graph grammars and transformation. Subgraph isomorphism problem plays a big role when we try to execute a query to find a pattern in a graph database. Therefore, the approach to solving the problem can directly affect the query performance of the big graph databases [1].

1.1 Motivation

In the literature, there are two types of techniques used in subgraph isomorphism algorithms, *filtering-and-verification* and *branch-and-bound* techniques. The filtering-and-verification technique aims to decrease the number of candidate dataset by creating an index of small graphs (features) and then eliminates irrelevant candidate data nodes with respect to indexed features. GraphGrep [9], GIndex [10], Labeled Walk Index (LWI) [11], Closure-Tree [2], Graph Decomposition Indexing [12], TreePi [13], TreeDelta [14] are based on the filtering-and-verification algorithm. On the other hand, the branch-and-bound technique find matching candidate data node for each query node in the graph by following branches connected to the matching the query and data node in the graph database. If the algorithm matches with all constraints, it continues the search, but if it does not match any data node, the algorithm backtracks and returns to the previous match [15]. VF2 [16], QuickSI [15], GADDI [17], GraphQL [18], SPath [19], TurboIso [3], DualIso [20], BB-Graph [1] and VF3 [21] are the most known and efficient branch-and-bound algorithms in the literature.

The existing algorithms of these two techniques have some problems with finding exact matches in big graph databases. More specifically, the *filtering-and-verification* techniques are designed for graphs with multiple parts and they are incapable of finding all exact matches in the graph databases. On the other hand, the *branch-and-bound (BB)* techniques can find all the exact matches. However, they are not effective for large datasets due to the recursive calls. Although most BB-based algorithms have introduced their own pruning methods, they are not efficient enough to eliminate mismatch patterns in higher levels; therefore, they must look in all the nodes and all edges. Additionally, using large data structures or customizing indexes increase the memory consumption. The BB-Graph [1] algorithm was developed based on these application problems by introducing its own pruning method using the Neo4j's graph data structures. It searches for exact matches from a starting node and continues to search for nodes connected to the already matching nodes. It has been proven that BB-Graph works with large data sets. However, complex queries cause performance issues in big graph databases.

In order to improve the performance of *branch-and-bound* algorithms, we examined all the subgraph isomorphism algorithms existing in the literature. We realized that all of these algorithms use various approaches to efficiently search for patterns in a data graph. However, the matching order selection methods have generally not been considered to improve query performance in large sets of graph databases. Defining a good matching order helps algorithms to find patterns in fewer attempts. Therefore, we examine matching order selection methods of all the algorithms in the literature as shown in Table 1.1. Only GraphQl, QuickSI, TurboIso, VF2 and its descendants VF2Plus and VF3 algorithms indicate matching order selection to able to execute queries more efficiently. GraphQl always selects a query node with the smallest candidate size connected to the query nodes already matching in each iteration. QuickSI uses a minimum spanning tree called *QI-Sequence* to order the nodes based on their label frequency. TurboIso searches for *neighborhood equivalence class* for each query node and then applies a selection order to them. On the other hand, VF2, VF2Plus and VF3 introduce a "node exploration sequence" which first uses the rarest and the most constrained nodes when searching for patterns [18, 15, 3, 22, 21].

Table 1.1: Matching Order Selection Usage in the Literature

Algorithm	**Matching Order Selection Method**
QuickSI	QI-Sequence
GraphQl	Candidate Size For Query Nodes
SPath	Candidate Size of the Path
Turboiso	Candidate Region Exploration
VF2, VF2-Plus, VF3	Node Exploration Sequence

Motivated by this, we introduced the *BB-Plus* approach that consists of heuristics for automatically selecting the best matching order selection method using volatility (real-time and historical) and size of the database, size and type of query (path, cyclic and others) as an input to improve the performance of subgraph isomorpishm queries. The BB-Plus executes time efficient queries in large graph dataset by using these heuristics to find the best matching order that eliminates redundant candidate nodes at high level in order to reduce the search space.

Within the scope of this study, we first offered five matching order selection method to improve the performance of subgraph isomorphism algorithms. Four of the selection methods were developed using fundamental measures of the graph centrality, such as the *degree*, *closeness*, *betweenness* and *eigenvector centrality*. The other one was developed based on *candidate node size*. Each of them can be applied to any subgraph isomorpishm algorithm to increase the query performance.

After developing these five matching order selection method, we decided to combine these matching order selection methods to a branch and bound algorithm. We choose BB-Graph algorithm because it gives the best performance among all the state-of-the-art subgraph isomorphism algorithm. Therefore, *Matching Order Based On Degree Centrality*, *Matching Order Based On Closeness Centrality*, *Matching Order Based On Betweenness Centrality*, *Matching Order Based On Eigenvector Centrality* and *Matching Order Based On Candidate Node Selection* methods are emerged and we examined their performance against various queries and with different dataset. We see that some methods give greater results in some queries and decide to create rules for determining the matching order selection method automatically for different query types.

We used decision tree in the process of generating rules for determining the matching order selection method. Decision tree created some rules based on the volatility and size of database, size and type of query as attributes and best matching order selection methods as output in the queries. Therefore, we used these rules to introduce our approach the BB-Plus. The BB-Plus decides which matching order selection method should use while executing a subgraph isomorphism query according to these rules.

1.2 Contributions and Novelties

The main contributions of this thesis are:

1. We introduced a new approach called *BB-Plus* that consists of heuristics for automatically selecting the best matching order selection method volatility (real-time, historical etc.) and size of the database, size and type of query (path, cyclic, recursive etc.) as an input to improve the performance of subgraph isomorpishm queries. It combines the best aspects of five different matching order selection methods along with BB-Graph algorithm.

2. In order to improve the performance of subgraph isomorphism algorithms, we offered five different matching order selection methods called *Degree Centrality*, *Closeness Centrality*, *Betweenness Centrality*, *Eigenvector Centrality* and *Candidate Node Selection*. We compared these matching order selection methods and found their strengths and weaknesses in different types of database and queries. Therefore, we generated basic rules that decide which matching order selection method to use in which query or databases and we used these rules while developing the *BB-Plus* approach.

3. We compared the state-of-the-art subgraph isomorphism algorithms in the literature such as *GraphQl*, *DualIso*, *TurboIso*, *VF3*, *BB-Graph* and *Neo4j's Cypher* using the same programming language and graph data structures (for storing nodes, edges and properties or indexing etc.) of Neo4j's graph database with our approach BB-Plus. A number of experiments have been performed in three different dataset. Most of the queries, the *BB-Plus* performs better than other algorithms.

1.3 The Outline of the Thesis

Organization of the rest of this thesis is as in the following:

Chapter 2 presents the background and the related work on the subgraph isomorphism algorithms. Chapter 3 presents our approach, which is called the BB-Plus, and how its developed using different matching order selection methods. Chapter 4 presents the experimental results of the comparison of the state-of-the-art subgraph isomorphism algoritmhs, GraphQl, DualIso, TurboIso, VF3, BB-Graph and Neo4j's Cypher, with our approach, BB-Plus. Finally, the conclusion and future work are given in Chapter 5.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Graph Databases

Graph database is a database that uses graph data structures for semantic queries like nodes, edges and properties to represent and store data. On the other hand, graph database management system (GDBMS) are used to manage graph databases. Graph databases employ nodes, edges and their properties. Nodes represents entities that we want to keep track and edges connect each nodes. With properties, we can keep information on both nodes and edges [5]. Library graph database is given as an example in Figure 2.1. In this example, we see that graph databases keep nodes such as books, persons etc. and its properties such as name, title etc., edges such as wrote, purchased etc. and its properties such as date etc.

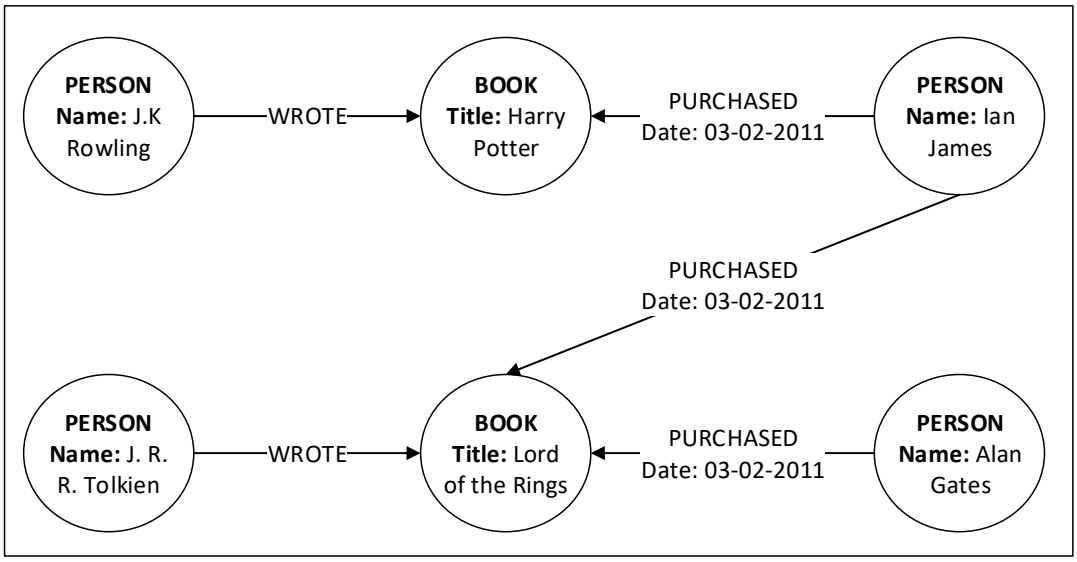

Figure 2.1: Library Graph Database Example

Graph databases built on three essential properties. Firstly, they use property graph as a data model [23]. Secondly, they use path traversal as a query language. Lastly, they satisfies index-free adjacency for physical level organization.

Index-free adjacency means there is no global or external index are needed for searching the existence of an edge between two nodes. Therefore, no index lookups are required in graph databases [24]. As shown in Figure 2.2, if we want to find *"People who writes Harry Potter"* in the example Library Relational Database, we need to create three indexes for *"Title"* attribute in *"Book"* table, *"BookId"* attribute in *"Writes"* table and *"Id"* attribute in *"Person"* table. However, in the graph databases, we only need an index to find the root node which is a book with title *"Harry Potter"* in our example an then just do traversal to find the people who writes it.

PERSON

Id	Name
1	J.K. Rowling
2	J.R. Tolkien
3	Edgar Allan Poe
..	..
4	George Orwell

WRITES

PersonId	BookId
1	1
2	2
4	3
..	..
4	4

BOOK

Id	Title
1	Harry Potter
2	Lord of the Rings
3	1984
..	..
4	Animal Farm

Figure 2.2: Library Relational Database Example

We need to consider using graph databases rather than relational database when we have a large and complex data set, expensive join operations, our data is highly connected (e.g. social networks) or in a recommendation systems (e.g. Netflix) [8].

Graph databases are good at managing ad-hoc and changing data with evolving schemas, recursive queries, semantic search and managing data set within graph structure, graph traversal or graph like queries (e.g. shortest path between two given nodes in a graph) [8].

Although graph databases have great features, they have some existing issues. Modeling a graph database and migrating data, queries, indexes etc. from an existing database is a big problem because there is no well-defined way to do it. Additionally, some of graph databases are lack of fully ACID properties and standard query language. On the other hand, relational databases are more mature than the graph databases and they have feature rich environment. Therefore, graph databases create trust issues for large businesses. Lastly, their memory consumption is high when they compare to other databases [25].

Graph databases are used in different applications in social, information, technological and biological networks [5, 6]. Graph databases are mostly used in social networks because it is easy to represent people, groups and their relationships like friendship etc. with graph data structures. Additionally, graph databases are used for social network analysis [26].

Graph databases are also used in recommendation systems to recommend a product, user etc. to its users by predicting based on user data. Associations between items or users can be easily defined as a graph model and recommendation can be done by graph traversal. For example, Google uses graph database to show relevant ads, Facebook uses for friend suggestions and Amazon uses to recommend products [27].

In bioinformatics, graph databases are used for modelling metabolic pathways, chemical structures, genes, proteins and enzims etc. For example, Bio4j is using graph database to store and query proteins [28].

Also in technological networks like Geographic Information Systems or spatial databases, graph databases can be used for geographic, topological and metric operations [26].

2.1.1 Comparison of Graph Databases

Although most of the GDBMSs exist for less than ten years and developments are still coming within their roadmaps, they provide different capabilities and features [6].

Neo4j, AllegroGraph and Sones have their own query language, which are Cypher, SPARQL and GraphQL respectively. The others usually use SQL-based language for querying [6].

Each graph database can use different data structures to represent data. For example, Hypergraph and SonesDB use hypergraph to store data [5] that is useful in knowledge representation, artificial intelligence and bioinformatics [29]. OrientDB, AllegroGraph and ArangoDB are multi-model databases that can use different data models for solving different problems. For example, ArangoDB is also a RDF store database that makes it meet with the Semantic Web standards [5].

Most of the graph databases use master/slave architecture for replication. When a slave needs to write, it needs to synchronize with the master to preserve consistency [30]. These updates will be eventually visible to other slaves by master node that could lead master node to bottleneck. OrientDB solves the bottleneck problem by multi-master (master-master) replication. In OrientDB, all the nodes in the cluster are master so they can both read and write. This also allows each cluster to scale up horizontally [31].

All of the graph databases use different data structure for indexing as shown in Table 2.1. However, they usually use a text search engine called Apache's Lucene for full text indexing.

Triggering feature only comes up with Neo4j, OrientDB, Titan and AllegroGraph for catching events. Graph databases have high throughput like other NoSQL databases so using triggering could also create performance problems.

Geospatial queries are available on Neo4j, OrienDB, Titan, AllegroGraph and ArangoDB. Neo4j and Titan have external geospatial libraries to handle geospatial queries that are called Neo4j Spatial and Titan Geo respectively. OrientDB, Titan and ArangoDB have spatial indexes to faster geospatial queries.

Table 2.1: Comparison of graph database management systems

	Neo4j	OrientDb	Titan	AllegroGraph	ArangoDB	InfiniteGraph
Database Model	GDBMS	Document Store GDBMS	GDBMS	RDF Store GDBMS	Document Store GDBMS Key-value store	GDBMS
Year	2007	2010	2012	2004	2012	2010
Price	Min 12K/Year	Free	Free	Commercial	Free	Commercial
Query Language	Cypher	SQL-Based	Gremlin	SPARQL	Gremlin	Gremlin
Trigerring	Event Handler	Hooks	EventGraph	Yes	-	-
Indexing	Schema Indexes Legacy Indexes Lucene Full Text Indexing	SB-Tree Index Hash Index Lucene Full Text Index Lucene Spatial Index	Graph Indexes Vertex-Centric Indexes Lucene Full Text Index Lucene Spatial Index	Solr Text Indexing	Hash Index Skiplist Index Geo Index Full Text Index	-
Property Graph	Yes	Yes	Yes	No (RDF)	Yes	Yes
Replication	Master/Slave	Multi Master	Master/Slave	Master/Slave	Master/Slave	Master/Slave
Partitioning	No	Sharding	Balanced Partitioning	with Federation	Sharding	Sharding
Consistency	Fully ACID	Fully ACID	ACID or Eventually Consistent	ACID	ACID	Fully ACID
Supported Programming Languages	.Net, Clojure, Go, Groovy, Java, Perl, PHP, Pyhton, JavaScript, Ruby	.Net, C, CSharp, Clojure, Java, Scala JavaScript, PHP, Pyhton, Ruby, Scala	Clojure, Java, Pyhton	CSharp, Clojure, Java, Lisp, Perl, Pyhton, Ruby, Scala	CSharp, Java, JavaScript, PHP, Python, Ruby, Clojure	Java, Blueprints
Security	-	User and role authentication Record level security	User authentification Rexster Graph Server	User authentification	-	-
MapReduce	BatchImporter	Yes	Yes (via Fanus)	No	No	Yes
Geospatial Queries	Neo4j Spatial	Yes	Titan Geo	SPARKQL	Yes	No

2.2 Centrality Measures in Graphs

Centrality is a quantative measure that determines the most important (or central) node in a graph. The most central node can be changed based on how a node is identified as the most important. The most important node could be node with largest degree or the one that closest to the other nodes or the one that most shortest paths pass through or the one that has dominant eigenvector. With different centrality measures, we can detect different properties of a graph. Degree, betweenness, closeness and eigenvector centralities are one of most popular centrality measures in the literature.

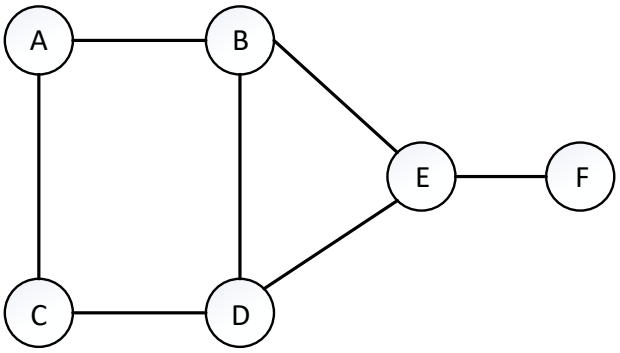

Figure 2.3: Example graph G for calculating centrality measures

2.2.1 Degree Centrality

Degree centrality is the first and the simplest centrality measure that defines the importance of a node based on its degree that can be calculated with the number of incoming and outgoing edges of nodes. If the node's degree is bigger than the degree of other nodes, then the importance of the node is higher than the other nodes. Degree centrality is good at searching for most connected nodes or the nodes with most information that can easily connect with other nodes in a graph. The pseudo code [32] for calculating degree centrality for graph is given in Algorithm 1.

Algorithm 1 DegreeCentrality

Input: Q : Query graph

Output: $C_d(Q)$: Degree Centrality values for nodes of graph Q

1: **procedure** DEGREECENTRALITY(Q)
2: $C_d(Q) \leftarrow \emptyset$
3: **for** i `from 1 to` $|Q|$> **do**
4: **for** j `from 1 to` $|Q|$> **do**
5: **if** $matrix[i][j]$!= 0 **then**
6: $degree[i]++$
7: $C_d(v_i) = degree[i]++$
8: Push $C_d(v_i)$ to $C_d(Q)$
9: **return** $C_d(Q)$

Degree Centrality can also be expressed as in Equation 21. According to the equation, the degree centrality of a node can be calculated with incoming and outgoing edges that touches to the node which can be also describes as degree of the node.

$$C_{degree}(v_k) = degree(v_k) \tag{21}$$

Let's calculate degree centrality for each nodes in graph G. Therefore, we need to find the degrees of each node which are $2, 3, 2, 3, 3, 1$ for A, B, C, D, E, F respectively.

2.2.2 Closeness Centrality

Closeness centrality is determined for each node in the connected graph based on their closeness to other nodes in the graph that can be calculated with sum of the shortest paths between the node and the other nodes. If a node is more central, then its distance between the other nodes is lower than the other nodes in the graph. With closeness information of graph, we can determine the time for nodes to spread information from itself to the all other nodes. Therefore, it is good at searching for nodes that deploy in the best location to easily affect other nodes in the graph. The pseudo code [32] for calculating betweenness centrality for graph is given in Algorithm 3.

Algorithm 2 ClosenessCentrality

Input: Q : Query graph

Output: $C_c(Q)$: Closeness Centrality values for nodes of graph Q

1: **procedure** CLOSENESSCENTRALITY(Q)
2: $C_c(Q) \leftarrow \emptyset$
3: **for** i `from 1 to` $|Q|$> **do**
4: $Path[i]$ = Execute $Dijkstra(i)$
5: **for** j `from 1 to` $|Q|$> **do**
6: $PathLength[i]$ += $Path[i][j]$
7: $C_c(v_i) = 1/PathLength[i]$
8: Push $C_c(v_i)$ to $C_c(Q)$
9: **return** $C_c(Q)$

Closeness Centrality can also be expressed as in Equation 22. According to the equation, to find closeness value of node k (v_k), we need to find the distance between v_k and all other nodes ($d(v_i, v_k)$) in the graph and normalize it.

$$C_{closeness}(v_k) = \frac{1}{\sum_i^N d(v_i, v_k)} \tag{22}$$

Let's calculate the closeness value of each nodes in graph G. To do that we need find the shortest distance between all the nodes as in Table 2.2. For example; the distance between node E and all the other nodes is $2, 1, 2, 3, 1$, respectively. Therefore, the closeness value of E can be obtained by dividing total distance (9) by 1 which is 0.11. The closeness value for all the other nodes can be seen in Table 2.3.

Table 2.2: Distance matrix of example query graph G

	A	**B**	**C**	**D**	**E**	**F**
A	-	1	1	2	2	3
B	1	-	2	1	1	2
C	1	2	-	1	2	3
D	2	1	1	-	1	2
E	2	1	2	3	-	1
F	3	2	3	2	1	-

Table 2.3: Calculating closeness centrality of example query graph G

Nodes (v)	**Total # of shortest paths between v and the other nodes**	$C_c(v)$ **(Closeness Centrality of v)**
A	1 + 1 + 2 + 2 + 3 = 9	1 / 9 = 0.10
B	1 + 2 + 1 + 1 + 2 = 7	1 / 7 = 0.14
C	1 + 2 + 1 + 2 + 3 = 9	1 / 9 = 0.11
D	2 + 1 + 1 + 1 + 2 = 7	1 / 7 = 0.14
E	2 + 1 + 2 + 3 + 1 = 9	1 / 9 = 0.11
F	3 + 2 + 3 + 2 + 1 = 11	1 / 11 = 0.09

2.2.3 Betweenness Centrality

Betweenness centrality is defined by how many times a node is included on the shortest path that is calculated among all the nodes in a graph. The most important node in the betweenness centrality acts as a bridge between all the nodes in a graph. Therefore, it is good at defining a node that most affect the flow in a graph. The pseudo code [32] for calculating betweenness centrality for graph is given in Algorithm 3.

Algorithm 3 BetweennessCentrality

Input: Q : Query graph

Output: $C_b(Q)$: Betweenness Centrality values for nodes of graph Q

1: **procedure** BETWEENNESSCENTRALITY(Q)
2: $C_b(Q) \leftarrow \emptyset$
3: **for** i `from 1 to` $|Q|$> **do**
4: $VectorPath[i]$ = Execute $Dijkstra(i)$
5: **for** j `from 1 to` $|Q|$> **do**
6: $VectorAllPath$ += $Path[i][j]$
7: Add $Path[i]$ into $VectorAllPath$
8: **for** i `from 1 to` $|Q|$> **do**
9: **for** j `from 1 to` $VectorAllPath.size()$> **do**
10: **if** i in $VectorAllPath[j]$ **then**
11: $PathNumber[i] + +$
12: $C_b(v_i)$ = $PathNumber[i]/2/((|Q| - 1)(|Q| - 2)/2)$
13: Push $C_b(v_i)$ to $C_b(Q)$
14: **return** $C_b(Q)$

Betweenness Centrality can also be expressed as in Equation 23 where g_{ij} is the number of shortest path between v_i and v_j and $g_{ij}(v_k)$ is the number of shortest path between v_i and v_j that contains node v_k.

$$C_{betweenness}(v_k) = \sum_{i}^{N} \sum_{j}^{N} \frac{g_{ij}(v_k)}{g_{ij}} \tag{23}$$

Table 2.4: Calculating betweenness centrality of example query graph G

Nodes (v)	**Shortest path passes through the node**	$C_b(v)$ **(Betweenness Centrality of v)**
A	C-B (C-A-B, C-D-B)	1 / 2 = 0.50
B	A-D (A-B-D, A-C-D), A-E (A-B-E), A-F (A-B-E-F)	1 / 2 + 1 + 1 = 2.50
C	A-D (A-B-D, A-C-D)	1 / 2 = 0.50
D	C-B (C-A-B, C-D-B), C-E (C-D-E), C-F (C-D-E-F)	1 + 1 + 1 / 2 = 2.50
E	A-F, C-F, B-F, D-F	1 + 1 + 1 + 1 = 4
F	-	0

To be able to normalize the betweenness value, we need to divide results by the number of pairs of vertices not including v, which for directed graphs is $(n-1)(n-2)$ and for undirected graphs is $\frac{(n-1)(n-2)}{2}$.

Let's calculate the betweenness value of each nodes in graph G. To do that we need find all the shortest paths and the shortest path that contains each nodes like in Table 2.4. For example, the betweenness value of node E is calculated as for 4. Because four shortest paths in the graph which are $A-F, C-F, B-F$ and $D-F$ passes through the node E.

Table 2.5: Adjacency matrix of example query graph Q

	A	**B**	**C**	**D**	**E**	**F**
A	-	1	1	0	0	0
B	1	-	0	1	1	0
C	1	0	-	1	0	0
D	0	1	1	-	1	0
E	0	1	0	1	-	1
F	0	0	0	0	1	-

2.2.4 Eigenvector Centrality

Eigenvector centrality is calculated for a node with its degree and the quality of its edges between other nodes. It is more about calculating the measure of the influence of a node in the graph. If the edges connected to nodes with high score nodes than it consider more important than the other nodes. Additionally, Google's PageRank and Katz centrality measures derived from eigenvector centrality [33]. The pseudo code [32] for calculating degree centrality for graph is given in Algorithm 4.

Algorithm 4 EigenvectorCentrality

Input: Q : Query graph

Output: $C_e(Q)$: Eigenvector Centrality values for nodes of graph Q

1: **procedure** EIGENVECTORCENTRALITY(Q)
2: $C_e(Q) \leftarrow \emptyset$
3: **for** i `from 1 to` $|Q|$> **do**
4: $eigenvector[i] = 1$
5: **for** n `from 1 to` $MaxTimes$> **do**
6: **for** i `from 1 to` $|Q|$> **do**
7: $TmpEigen[i] = 0$
8: **for** j `from 1 to` $|Q|$> **do**
9: $TmpEigen[i]$ += $matrix[i][j] * eigenvector[j]$
10: $NormSq = 0$
11: **for** j `from 1 to` $|Q|$> **do**
12: $NormSq = TmpEigen[i] * TmpEigen[i]$
13: $Norm = sqrt(NormSq)$
14: **for** j `from 1 to` $|Q|$> **do**
15: $Eigenvector[i] = TmpEigen[i]/Norm$

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

$$(A-\lambda I)C_e = 0 \Rightarrow \begin{bmatrix} 0-\lambda & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0-\lambda & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0-\lambda & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0-\lambda & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0-\lambda & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0-\lambda & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0-\lambda & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0-\lambda \end{bmatrix} \begin{bmatrix} u1 \\ u2 \\ u3 \\ u4 \\ u5 \\ u6 \\ u7 \\ u8 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\det(A-\lambda I) = \begin{vmatrix} 0-\lambda & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0-\lambda & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0-\lambda & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0-\lambda & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0-\lambda & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0-\lambda & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0-\lambda & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0-\lambda \end{vmatrix} = 0$$

$$\Rightarrow \lambda_1 = 2.60 \quad \lambda_2 = -2.06 \quad \lambda_3 = -1.72 \quad \lambda_4 = 1.34 \quad \lambda_5 = -1.60 \quad \lambda_6 = 1.70 \quad \lambda_7 = 0.37 \quad \lambda_8 = 0.21$$

*The largest eigenvalue is **2.60**. So the corresponding eigenvector:*

$$\begin{bmatrix} -2.6 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & -2.6 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & -2.6 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -2.6 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & -2.6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & -2.6 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & -2.6 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -2.6 \end{bmatrix} \begin{bmatrix} u1 \\ u2 \\ u3 \\ u4 \\ u5 \\ u6 \\ u7 \\ u8 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\Rightarrow \lambda_1 = -4.66 \quad \lambda_2 = -4.32 \quad \lambda_3 = -3.60 \quad \lambda_4 = -2.97 \lambda_5 = -2.38 \quad \lambda_6 = -1.60 \quad \lambda_7 = -1.25 \lambda_8 = -0.01$$

*The largest eigenvalue is **-0.01** and the eigenvector associated to it:*

$$\begin{bmatrix} 0.54 \\ 0.39 \\ 0.46 \\ 0.27 \\ 0.21 \\ 0.24 \\ 0.35 \\ 0.13 \end{bmatrix} \Rightarrow$$

A B C D E F G H

Ce = {0.54, 0.39, 0.46, 0.27, 0.21, 0.24, 0.35, 0.13}

Figure 2.4: Calculating eigenvector centrality of example query graph G

Eigenvector Centrality can also be expressed as in Equation 24 where λ is a constant, A is the adjacency matrix of a graph such that $a_{ij} = 1$ if node v_i is connected to v_j and $a_{ij} = 0$ if not and x_i is leading eigenvector of node v_i.

$$C_{eigenvector}(v_k) = \frac{1}{\lambda} \sum_{t \in G} a_{vt} x_t \tag{24}$$

Let's calculate the eigenvector value of each nodes in graph G. Firstly, we need to obtain the adjacency matrix of Q that given in Table 2.5 and then we need follow the steps that given in Figure 2.4.

2.3 The Subgraph Isomorphism Problem

In the thesis, the graph G is defined as (V, E) where V is the set of nodes and E is the set of edges. L_v and P_v denote the label set and property set of node v respectively. Similarly, for an edge $e =< u, v >$, L_e, P_e and dir_e^u denote the label set, property set and the direction of the edge e with respect to node u, respectively.

Given $Q : (V, E)$ and $G : (V', E')$, G is an exact match of Q, if there is a one-to-one and onto function $f : V \leftarrow V'$ such that $L_v \subseteq L_f(v)$, $P_v \subseteq P_f(v)\ \forall v \in Q$ and $s =< f(u), f(v) >\in E'\ \forall r =< u, v >\in E$ 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 subgraph isomorphism problems is to find all distinct exact matches of Q in G.

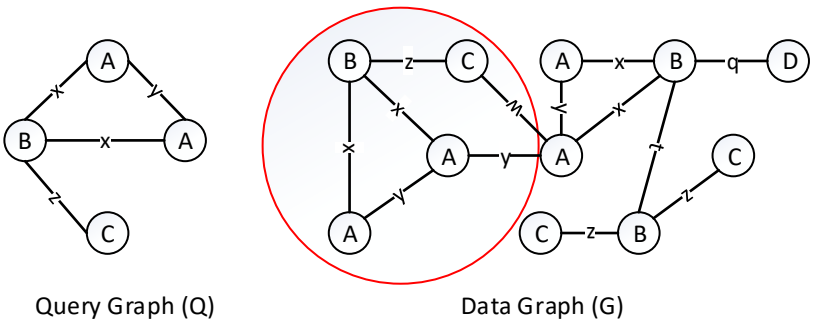

Figure 2.5: Example for Subgraph Isomorphism Problem

The matched nodes must satisfy the label comparison and degree comparison test. In other words, matching couple (u, u') must satisfy $L_u \subseteq L'_u$ and $u.deg \leq u'.deg$ [1].

As shown in Figure 2.5, subgraph isomorpishm problem can be described as, finding all the exact matches of query graph Q in data graph G.

2.4 The Subgraph Isomorphism Algorithms

In the literature, subgraph isomorphism algorithms are divided into *filtering-and-verification* and *branch-and-bound* techniques according to their strategies. Filtering and verification algorithms filter the data graph based on non-matched index of query graph and verify the matches. They are good at reducing the candidate data nodes and graphs with disconnected parts. However, they cannot find all exact matches of query graph. An example for filtering and verification algorithms (Closure-Tree) is given in Figure 2.6.

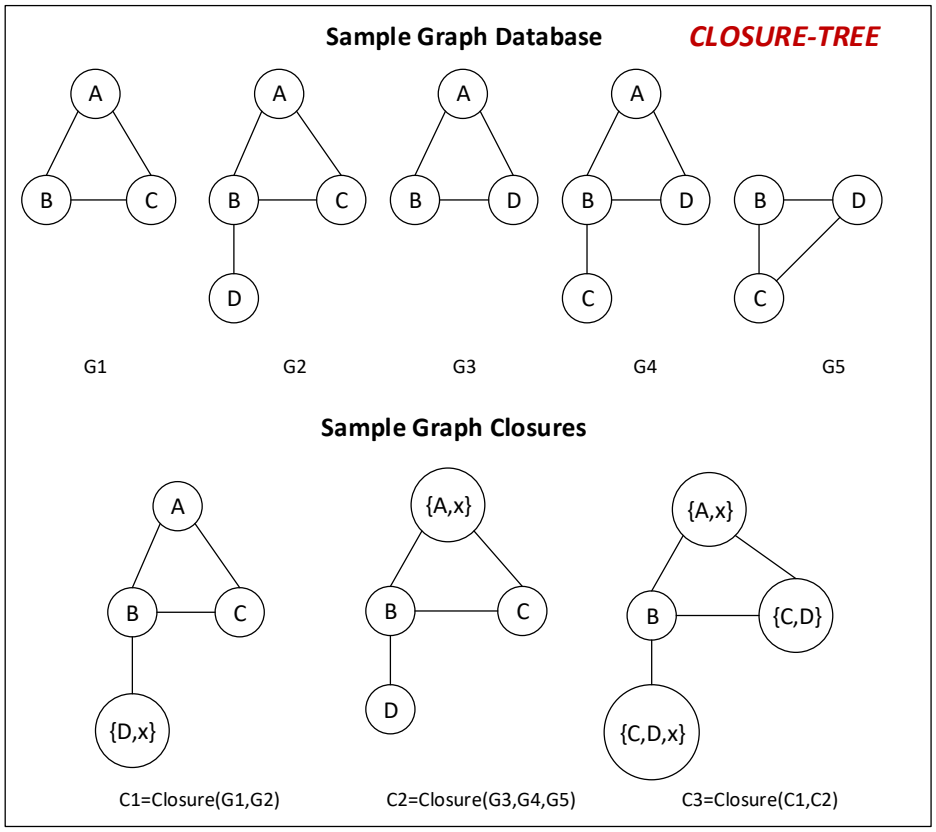

Figure 2.6: Example for Filtering and Verification Algorithms - Closure Tree [2]

On the other hand, branch and bound algorithms can find all the exact matches based on finding candidate data nodes for each query node and continue with new query nodes by branching from already matched query nodes. Although they are good at finding exact matches unlike filtering-and-verification technique algortihms, they are not computationaly efficient. Because, they search globally on the data graph and tries to find all candidate nodes by branching on irrelevant relationships and they use excessive memory based on the need of large data structures [1]. Therefore, prunning and matching order selection methods are important in these kind of algorithms to increase the performance. They eliminate irrevelant candidate nodes at early stepts in order to find exact matches more efficiently. The comparison of the all subgraph isomorpishm algorithms are shown in Table 2.6.

2.4.1 Ullmann's Algorithm

Ullmann proposed the first subgraph isomorphism algorithm that introduces the essential concepts of the problem and provides a basis for development of subsequent algorithms. The algorithm uses branch and bound technique to find all exact matches of a query graph. The algorithm is incapable to execute efficient queries on the large datasets and cannot compete with subsequent algorithms. Becuase the algorithm is in the immature state compared to the others and it has inadequate pruning rules and do not apply any matching order strategies. However, it created a basic structure for subsequent algorithms to develop new techniques to increase the query performance.

Ullmann's algorithm filters the candidate data nodes first for each query nodes by applying label and degree comparison tests. After finding the candidate nodes, it randomly picks a start query node and continue to find matching data nodes for a query node from randomly ordered query nodes. After finding a matching couple, the algorithm applies *IsJoinable* procedure that controls all the relationships between the matching query node and already matched query nodes is defined between matching data node and already matched data nodes [34]. If the matching couple successfully passes the procedure, then the matching couple added to the partial solution. However, the procedure fails then the algorithm backtracks and continue with the other candidate nodes [35].

2.4.2 VF2

The VF2 algorithm was built on the structure of the Ullmann's algorithm and it designed to achieve better performance on large graph datasets. The algorithm uses tree-based index structure to find all the exact matches.

The algorithm starts with an initial empty state and then adds a start query node from given order of the input. It continues with a query node that connected to already matched query nodes that satisfies all feasibility rules to create intermediate states. Therefore, the algorithm does not need to apply *IsJoinable* procedure and can prune out some candidate data nodes that are not connected to the already matched nodes. Finally, the algorithm stops when it covers all the nodes in the query graph.

Although the VF2 does not apply matching order selection, it has three pruning rules to increase the performance. The first one helps to prune out the data node that is not connected to already matched data nodes. The second one checks that the number of the intersection of adjacent query vertices and non-matched query vertices is bigger than the number of the intersection of adjacent data nodes and non-matched data nodes. The last one checks that the number of query nodes is bigger than data nodes that can be calculated as the difference between the sum of non-matched and the matched nodes from the adjacent nodes.

The VF2 algorithm performs better than the Ullmann's algorithm especially the size of query graph is bigger than twenty nodes. The execution time of finding patterns for Ullmann algorithm increase exponentially with the size of query graph and it causes a low performance when it compares to the VF2 algorithm [16].

2.4.3 QuickSI

The QuickSI is a subgraph isomorphism algorithm based on filtering and verification and it uses feature-based index technique, which is called Swift-Index to reduce computational costs and gain the ability of working on the large datasets.

Additionally, the algorithm has a matching order selection method, which called QI-Sequence. QI-Sequence is a minimum spanning tree that can be created by the

weighted edges. The weight of edges can calculated by the label frequency of both incoming and outgoing nodes based. QI-Sequence is ordered by ascending label frequency. Therefore the algorithm continues with a low frequency node, both the number of recursive calls and the query execution time are decreased.

On the other hand, QuickSI applies *IsJoinable* procedure to able to prune out the data vertices that has no edges between the already matched data vertices. QuickSI is also compared with Ullmann's algorithm and it collects query results in less time, especially with the increasing size of query graph [15].

2.4.4 GraphQL

The GraphQL is a query language that is developed to be able to manipulate graph databases that has attributes on both nodes and edges. GraphQL uses different pruning rules and matching order selection method to increase performance.

The GraphQL algorithm use two different pruning rules that are neighborhood signature based pruning and pseudo subgraph isomorphism test based pruning. Neighborhood signature based pruning ensures that labels of a query node's neighbors are a subset of label of candidate data node. On the other hand, pseudo isomorphism test helps to prune out data vertex if the breadth first search tree of the query vertex is not contained by the breadth first search tree of data vertex.

In addition to, the GraphQL also differs from other algorithms with its matching order selection method. The algorithm always selects a query node with the smallest candidate size that is connected with the already matched query nodes in each iteration of finding matches of query graph [18].

2.4.5 GADDI

The GADDI algorithm is emerged from the need for indexing large graphs in biological data. Until the GADDI algorithm was developed, there is no algorithm that can index large data graphs. Therefore, the algorithm uses distance based indexing that uses neighboring discriminating substructure (NDS) distance in large graphs, which

can be calculated with the number of exact patterns in a partial subgraph.

The GADDI does not apply matching order selection. It starts with the first query vertex in the input and continue with a query node that first appears in the depth-first search.

Additionally, the GADDI has three pruning rule. Firstly, it checks the query node's labels is a a subset of data vertex's labels. Secondly, it checks NDS distance of query vertex is smaller or equal than the data vertex's NDS distance. Finally, the third rule checks the shortest distance between query node and its neighbors is greater than or equal to the shortest distance between data node and its neighbors [17].

2.4.6 SUMMA

The SUMMA algorithm also noted that the database size is the biggest challange for the subgraph isomorphism problem. The algorithm uses an novel index based structure that generates local and global indexes to reduce the disk accesses and redundant calculations to able to execute queries in large graph datasets. Memory consumption of the algorithm is only based on the number of nodes.

Local indexes keeps infrequent label combinations which is sorted label list by lexicographic order. Infrequent label combinations are better than the frequent ones. Because, it can keep more information within less space. With local indexes, the algorithm can determine the matching order.

As for global indexes, the algorithm prefers distance based indexing like the GADDI. However the space complexity is high for the large graph datasets, it prefers shortest path trees. Because the size changes linearly with the node size.

Zhang et al. compare the SUMMA algorithm with the GADDI algorithm based on index construction time, index size, induced subgraphs and model generated graphs. Although, it was shown that the GADDI performs better in small datasets, the SUMMA beats the GADDI in large graphs dataset [36].

2.4.7 SPath

The SPath algorithm is built on a pattern based indexing structure that uses shortest path trees as indexes to be able to handle large graph datasets like the SUMMA algorithm. The algorithm divides query graph into shortest path and finds candidate paths for them in the data graph. Then, it brings all the paths together to find all the embeddings. In other words, the algorithm does not match nodes with candidate nodes, it matches paths with candidate paths at a time.

The selection order of query paths affect the query performance directly. The selection order of the SPath is about finding the most selective path that depends on the node with smallest candidate size and size of all vertices in a path. Additionally, the SPath applies neighborhood signature based pruning on the query path so that neighbor data path satisfies labels of the neighbor query path.

The SPath was compared GraphQL with thousand queries and the average query processing time for the queries are at least four times better than the GraphQl [19].

2.4.8 TurboIso

The TurboIso algorithm has emerged from low query performance of existing subgraph isomorphism algorithms based on lack of a good mathing order selection method. TurboIso comes with two important concept, which are *candidate region exploration* and *combine and permute strategy (Comb/Perm)*.

Candidate region exploration procedure searches candidate regions (CR) in the data graph and it helps to generate a matching order for each candidate region. Candidate region exploration exploits from *neighborhood equivalence class (NEC)*, which reduce the size of candidate region. The procedure finds all candidate data vertices for each NEC node and keeps them in a candidate subregion and then, the algorithm performs DFS in NEC tree to repeat this procedure for each ordered NEC node. The

matching order for each NEC node can be calculated as in Equation 25.

$$\sum_{v \in CR(P(u'),-)} \binom{|CR(u',v)|}{|u'.NEC|} \tag{25}$$

Comb/Perm strategy can find combinations for the query nodes in the same NEC instead of do permutation for all possible enumerations. However if the algorithm detects that a combination is not going to involve in the final solution, algorithms prunes out all the permutations for the combination.

The TurboIso algorithm was compared with QuickSI, GADDI, Spath, VF2 and algorithms with large datasets like AIDS, Human, NASA dataset. According to the paper, no algorithms other than GraphQl have completed most of the queries. Additionally, TurboIso performs 3.22 and 1836 times better than GraphQL in NASA and Human dataset respectively [3].

Algorithm 5 The TurboIso Algorithm

Input:: Q : query graph, G : data graph

Output: all embeddings of Q in G

1: **procedure** TURBOISO(Q, G)
2: $u_s \leftarrow ChooseStartVertex$
3: $Q' \leftarrow RewriteNecTree(Q, u_s)$
4: **for all** v_s in $v|(v \in V(G))$ and $(L(u_s) \subseteq L(v_s))$ **do**
5: **if** $ExploreCR(u'_s, v_s, CR = FAIL)$ **then**
6: **continue**;
7: $order \leftarrow DetermineMatchingOrder(Q', CR)$
8: $UpdateState(M, F, u_s, v_s)$
9: $SubgraphSearch(Q, Q', G, order, 1)$
10: $RestoreState(M, F, u_s, v_s)$

2.4.9 DualIso

The DualIso is a tree search based algorithm that uses effiecient pruning rules to be able to execute queries using less time and memory in large graph datasets. The DualIso consists of two basic pruning method that are simple and dual simulation.

Simple simulation checks that if there is a neighbor data node of the matched data node with the same label as the neighbor node's label of the matched node as shown in Equation 26. Simple simulation is very close to the refinement procedure of the Ullmann's algorithm and therefore insufficient for pruning in large graph dataset.

$$\forall(u, u') \in E_q, \forall v \in \phi(u), \exists v' \in \phi(u') s.t. (v, v') \in E \tag{26}$$

In contrast to simple simulation, dual simulation not only controls child nodes, but also parent nodes. For each query node, which is a neighbor of a matched query node, it checks neighbor nodes of the data node, which has any intersection with neighbor query node as shown in Equation 27. Dual simulation prunes out the nodes early state, therefore it minimizes the search space and increases the query performance.

$$\forall(u, u') \in E_q, \forall v' \in \phi(u'), \exists v \in \phi(u) s.t. (v, v') \in E \tag{27}$$

Firstly, the algorithm tries to find the feasible matches of nodes by finding all data nodes with the same label as query node and then it applies dual simulation to prune out unwanted data nodes. Then, it tries to find matching couples with applying depth-first search procedure until reach the maximum depth of the query graph. If the algorithm reach an unfeasible state then it backtracks.

DualIso compared with GraphSimIso, VF2 and GraphQl with amazon-2008 and enwiki-2018 dataset. In the experiment, the effect of dataset size, query size, labels and data graph density was investigated and DualIso gives the best results under this circumstances [20].

Algorithm 6 The DualIso Algorithm

```
1:  procedure DUALISO(G, Q, Φ)
2:      changed ← true
3:      while changed do
4:          changed ← false
5:          for all u ← V_q do
6:              for all u' ← Q.adj(u) do
7:                  Φ'(u') ← ∅
8:                  for all v ← Φ do
9:                      Φ_v(u') ← G.adj(v) ∩ Φ(u')
10:                     if Φ_v(u') = ∅ then
11:                         remove v from Φ(u)
12:                         if Φ(u) = ∅ then
13:                             return empty Φ
14:                         changed ← true
15:                     Φ'(u') ← Φ'(u') ∪ Φ_v(u')
16:                     if Φ'(u') = ∅ then
17:                         return empty Φ
18:                     if Φ'(u') < Φ(u') then
19:                         changed ← true
20:                     Φ(u') = Φ(u') ∩ Φ'(u')
21:     return Φ
```

2.4.10 BB-GRAPH

The BB-Graph algorithm has emerged from inadequate pruning techniques and excessive memory usage due to the large data structure needs of other algorithms. The BB-Graph algorithm uses branch and bound technique with backtracking strategy. Additionaly, the algorithm has efficient memory usage because it is built on data structures of Neo4j. The psuedo code for the algorithm is given in Algorithm 7.

Algorithm 7 The BB-Graph Algorithm

Input:: Q : query graph, G : data graph

Output: M : all embeddings of Q in G

1: **procedure** BBGRAPH(Q, G)
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** u_s has property **then**
7: $(C_u)_s \leftarrow FilterByProperty(u_s, (C_u)_s)$
8: **for all** $<v_s \in (C_u)_s>$ **do**
9: $M_{node} \leftarrow \emptyset$, $M_{rel} \leftarrow \emptyset$, $S \leftarrow \emptyset$
10: S.push $M_< u_s, v_s >$
11: Add $M_< u_s, v_s >$ into M_{node}
12: TransitionState()
13: **return** C_u

The algorithm starts with a query node and find its potential candidate nodes by checking their label, degree, property and both incoming and outgoing relationships. *FilterByLabel* method helps to find all candidates nodes that have the same label with query nodes as described in Algorithm 28. After finding the candidate nodes with the same label, the algorithm checks the incoming and outgoing edges of candidates node are matching with the query nodes with *FilterByRelationship* method as described in Algorithm 29. The method makes sure that edges of candidate nodes have the same type and direction. This step takes linear time based on the size of candidate nodes list. Finally, the algorithm narrows the size of candidate nodes list down by calling

FilterByProperty method to checks that the candidate node's properties and their values are matched with the query nodes that described in Algorithm 30. This step calls if the query nodes has properties, if not then the step is passed and filtering process ends up. This step also takes linear time based on the candidate nodes list.

After finding the final candidate node list, the algorithm continue with calling *the reciprocal node branching process* recursively to obtain new matchings. In the process, the algorithm traverses on the other query nodes which is connected to the already matched query nodes. The algorithm backtracks to traverse on other possible matches on the data graph until find all the exact matches of the query graph. The branching process is described in Algorithm 8 and 9.

Algorithm 8 Transition State

Input: S, M, M_{node}, M_{rel}

Output: -

1: **procedure** TRANSITIONSTATE(S, M, M_{node}, M_{rel})
2: **if** $S \neq \emptyset$ **then**
3: $M_{<u,v>} = S.pop()$
4: $BranchAndMatch(M_{<u,v>})$
5: $S.push(M_{<u,v>})$
6: **else**
7: $M.add(M_{node}, M_{rel})$
8: **return**

Matched query node u and graph node v is stored in $M(< u, v >)$ stack and the algorithm branches from $M(< u, v >)$. But if there is another options exists for $M(< u, v >)$, they are handled by backtrack mechanism later.

The algorithm needs candidate relationships for node couple $< u, v >$ to branch from $M(< u, v >)$ and uses *FindCandidateRelationships* method at this point which is described in Algorithm 10. The method eliminate candidate relationships with checking the direction and type of candidate graph edges that matches with query edge.

The algorithm uses *IsMatchValid* method to make three types of conflict control on $M_{<u,v>}$ that described in Algorithm 11. The first one checks that if u_i was matched

Algorithm 9 Branch and Match

Input: S, $M_{u,v}$, M_{node}, M_{rel}

Output: -

1: **procedure** BRANCHANDMATCH(S, $M_{u,v}$, M_{node}, M_{rel})
2: R_u = [relationship r of u | $\exists r_x \in R_g$ s.t. $M < r, r_x > \in M_{rel}$]
3: **if** $R_u \neq \emptyset$ **then**
4: **for** $r_i \in R_u$ **do**
5: $C_{r_i} = FindCandidateRelationship(M_{u,v}, r_i)$
6: $ind_i = 0$
7: $size_i = |C_{r_i}|$
8: $i = 0$
9: **while** $i >= 0$ **do**
10: **while** $ind_i < size_i$ **do**
11: **if** $i == |R_u|$ **then**
12: TransitionState()
13: $i--$
14: Take back M_{node}, M_{rel} and S to the previous values
15: $s_i = C_{r_i}.get(ind_i)$
16: ind_i++
17: **if** $IsMatchValid(M < r_i, s_i >, M < u, v >)$ **then**
18: $i++$
19: $ind_i = 0$
20: $i--$
21: Take back M_{node}, M_{rel} and S to the previous values
22: **return**

Algorithm 10 Find Candidate Relationships

Input: $M_{<u,v>}$: The matched node that is branching

r_i: The query relationship adjacent to u whose candidate are searched

Output: C_{r_i}: Set of candidate relationships for r_i

1: **procedure** FINDCANDIDATERELATIONSHIPS($M_{<u,v>}$, r_i, C_{r_i})

2: **if** $r_i.dir^u == OUTGOING$ **then**

3: $C_{r_i} = [s_i =< v, v' > | s_i.type = r_i.type, v' \in V_G]$

4: **else**

5: $C_{r_i} = [s_i =< v', v > | s_i.type = r_i.type, v' \in V_G]$

6: **return** C_{r_i}

with v_i before. The second one checks that if u_i was not matched but v_i is already matched. The last one uses *FilterByRelationships* and *FilterByProperty* to checks that two nodes is matching. After all the checks, $< u_i, v_i >$ is considered as a valid match. The algorithm continues to branch until validity of all the candidate relationship are evaluated.

The BB-Graph algorithm offers different pruning techniques like *matching node principal* for the nodes and *matching relationship principal* for the relationships. Matching node principal ensures that all the candidate nodes of a query node has the same label, degree and property with the query node. On the other hand, matching relationship principal ensures that all the candidate edges have the same label, property and direction with the query edge.

The BB-Graph was compared with the GraphQL and Cypher on WorldCup, Bank and Population graph databases and it was shown that BB-Graph gives the best results among the all databases for most of the query types [1].

The detailed working process of BB-Graph is given in Figure 2.7 and 2.8. It shows how the algorithm finds query graph Q in data graph G, step-by-step.

Algorithm 11 IsMatchValid

Input: $M_{<r_i,s_i>}$: The matched relationship to be checked

$M_{<u,v>}$ The matched node currently being brached

Output: -

1: **procedure** ISMATCHVALID($M_{<r_i,s_i>}$, $M_{<u,v>}$)
2: $u_i = r_i.getOtherNode(u)$
3: $u_i = s_i.getOtherNode(u)$
4: **if** $\exists u_x \neq u_i s.tM < u_x, v_i > \in M_{node}$ **then**
5: **return** $false$
6: **if** $\exists v_x \neq v_i s.tM < u_i, v_x > \in M_{node}$ **then**
7: **return** $false$
8: **if** $M < u_i, v_x > \notin M_{node}$ **then**
9: **if** FilterByRelationships($u_i, v_i \neq \emptyset$) and FilterByProperty($u_i, v_i \neq \emptyset$) **then**
10: $M_{node}.add(M(< u_i, v_i >))$
11: $S_{node}.add(M(< u_i, v_i >))$
12: **else**
13: **return** $false$
14: $M_{rel}.add(M(< r_i, s_i >))$
15: **return** $true$

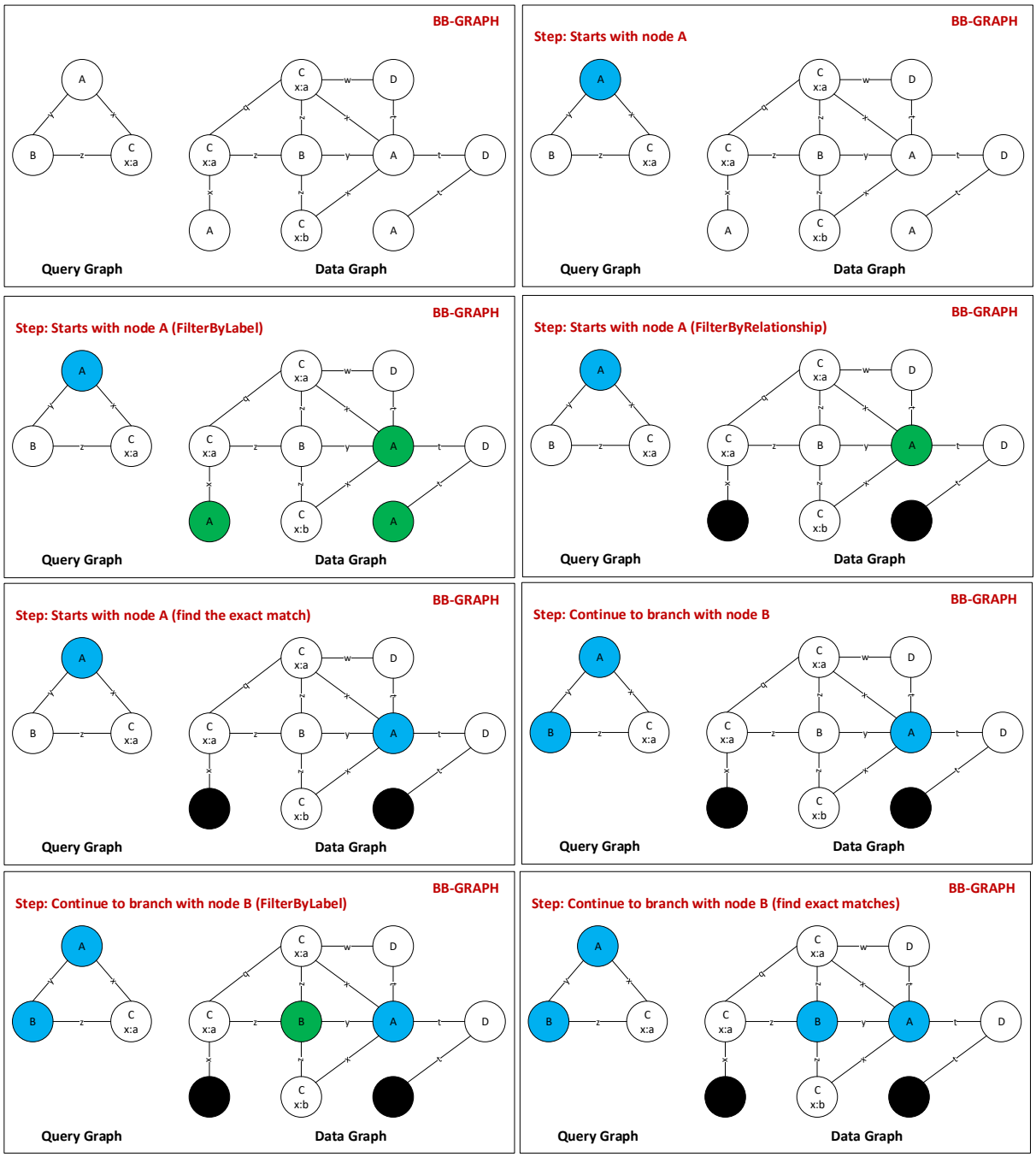

Figure 2.7: An Example for the working process of BB-Graph (Part-1)

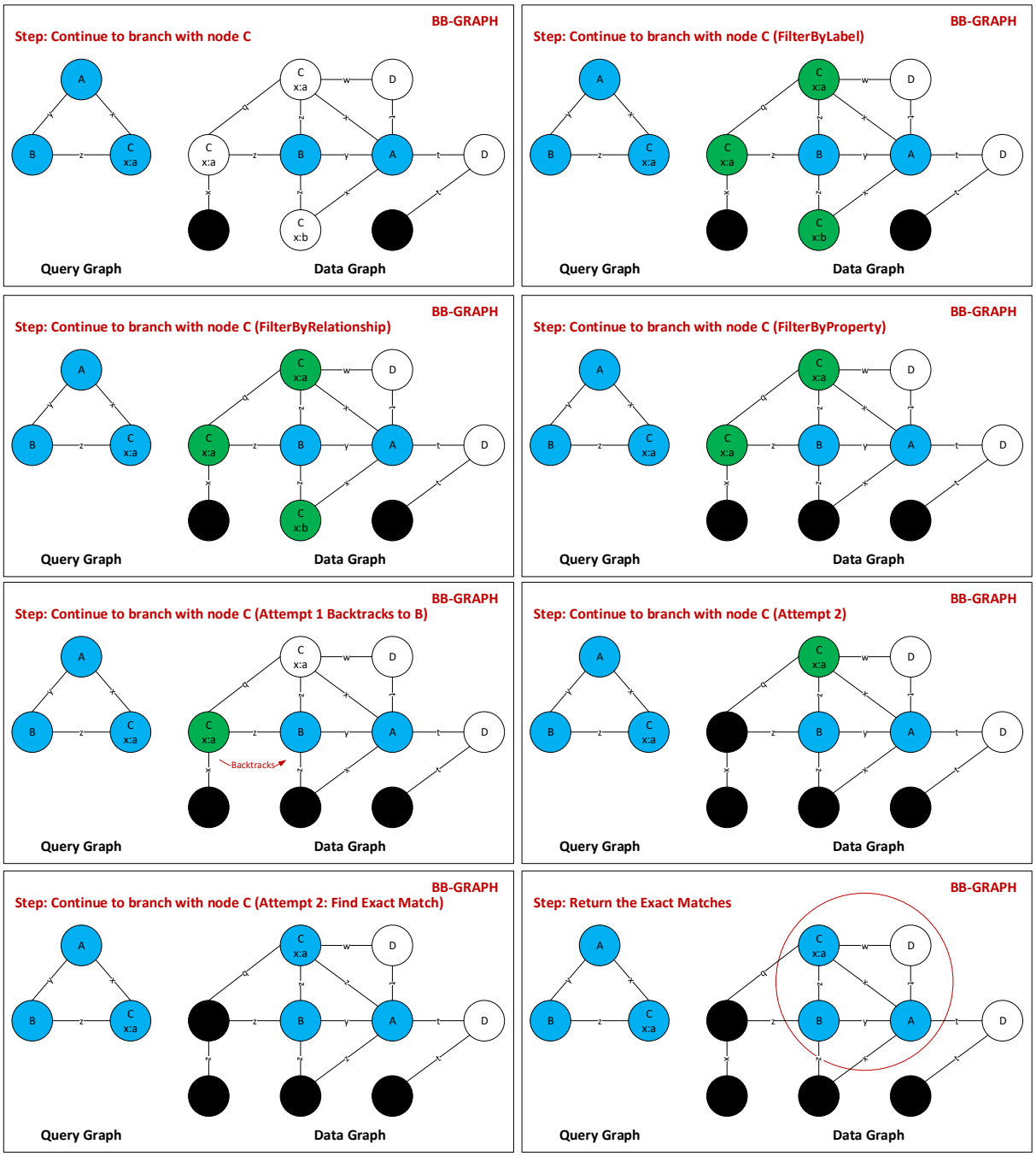

Figure 2.8: An Example for the working process of BB-Graph (Part-2)

2.4.11 VF2-Plus

The VF2-Plus algorithm is an improvement of VF2 algorithm by adding new pruning rules and a new matching order selection method to be able to execute query efficiently. The algorithm aims to generate a good matching order in order to prune infeasible branches at higher level and reduce the search space.

The algorithm introduces *node exploration sequence* for the matching order selection. Node exploration sequence tries to create an order based on the rareness of each nodes. In order to calculate their rareness, it calculates the probability of finding the node with same label and degree in data graph for each node in the query graph. In addition, it consider *node mapping degree* when defining the order. Node mapping degree is number of edges between the remaining node of query graph and the node in the node exploration sequence.

The VF2 algorithm applies *state space exploration* that starts with an empty state. A new pair of nodes that are a candidate node from the query graph and the matched node from data graph is added to create a new state. The algorithm applies feasibility rules to the new state to understand whether adding a new state will create a consistent state or not. It continues to adding new pair of nodes until there is no candidate node left and the goal state is reached. When no pair of nodes remains, the algorithm backtracks to the its parent state and try different pair of nodes.

There are three feasibility rules are defined in the algorithm which are “core rule”, “1-level look-ahead rule” and “2-level look ahead rule”. Core rules satisfies that the neighbors of matched couple nodes are needs to be matched with each other. 1-level and 2-level look ahead rule check the number of remaining neighbors that connected with incoming and outgoing edges, respectively [22].

2.4.12 VF3

The VF3 algorithm is an improvement of the VF2-Plus algorithm and it is very similar to its ancestor methods the VF2 and VF2-Plus algorithms. VF3 differs from VF2Plus with the way of preprocessing the query graph and selecting the next candidate couples. In the preprocessing, a tree is created based on node exploration sequence and the edges between nodes. For selecting the next candidate couple, the algorithm selects a node from node exploration sequence based on order for the query node. For the data node, it selects a node unmatched nodes based on the neighbors of matched query node [21].

Algorithm 12 The VF3 Algorithm

Input:: Q : query graph, G : data graph

Output: M : all embeddings of Q in G

1: **procedure** VF3(Q, G)
2: $M \leftarrow \emptyset$
3: $P_f = ComputeProbabilities(G, Q)$
4: $N_Q = GenerateNodeSequence(Q, P_f)$
5: $ClassifyNodes(Q, G)$
6: $(s_0, Parent) = PreprocessPatternGraph(Q, N_Q)$
7: $Match(s_0, G, Q, N_Q, Parent, M)$
8: **return** M

Table 2.6: Comparison of subgraph isomorphism algorithms

	Authors	Release Year	Indexing	Pruning Methods	Matching Order Selection
Ullmann's Algorithm	Ullmann	1976	-	Label comparison test Degree comparison test	-
VF2	Cordella et al.	2002	Tree-based Indexing	- IsJoinable Test - Intersection of Adjacent and Non-matched Query Vertices - Difference of the Non-matched and Matched Nodes from the Adjacent Nodes	-
QuickSI	Shang et al.	2008	Feature-based Indexing (SwiftIndex)	IsJoinable Test	QI-Sequence
GraphQL	He et al.	2008	Label-based Indexing	Neighborhood Signature-based Pruning Pseudo Subgraph Isomorphism Test	Candidate Size for Query Node
GADDI	Zhang et al.	2009	Distance-based Indexing (NDS Distance)	- Label Comparison Test - NDS Distance - Shortest distance between nodes and its neighbors	-
SUMMA	Zhang et al.	2010	Infrequent Label Combination (for Local Indexing) Distance-based Indexing (for Global Indexing)	Vertex must appear in one shortest path tree	-
SPath	Zhao et al.	2010	Pattern-based Indexing (Shortest Path Tree)	Neighborhood Signature-based Pruning	Candidate Size of the Path
TurboIso	Han et al.	2013	Pattern-based Indexing	Neighborhood Equivalence Class	Candidate Region Exploration
DualIso	Saltz et al.	2014	-	Simple Simulation Dual Simulation	-
BB-Graph	Asiler et al.	2014	-	Matching Node Principal Matching Relationship Principal	-
VF2-Plus	Jüttner et al.	2016	-	Core rule 1-level look-ahead rule 2-level look-ahead rule	Node Exploration Seqeunce
VF3	Carletti et al.	2017	-	Core rule 1-level look-ahead rule 2-level look-ahead rule	Node Exploration Seqeunce

2.5 Matching Order Selection in the Subgraph Isomorpishm Problem

In subgraph isomorpishm problem, matching order selection defines the order of query nodes that match with the candidate data nodes. Matching order selection is critical for the subgraph isomorpishm algorithms, because it directly affects the query performance by reducing the number attempts to find patterns in data graph.

The importance of matching order selection is shown in the figure 2.9. Let's consider that we have two data graphs, G_1 and G_2, like in the figure. If we use the matching order O(1) = (u_1, u_2, u_3) for G_1 then it only requires three attempts to find all the patterns. However, for G_2, we need 1003 attempts to find them. On the other hand, if we use O(2) = (u_1, u_2, u_3), we need 1003 and 3 attempts to find patterns in data graph G_1 and G_2 respectively [3].

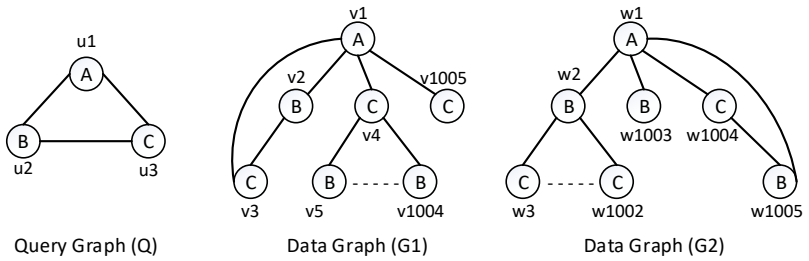

Figure 2.9: The Importance of Matching Order Selection in the Subgraph Isomorpishm Problem [3]

In the literature, most of the algorithms do not consider matching order selection as we mentioned before. They randomly picks a query node to find all the matching patterns. Therefore, if we can define a good matching order selection method, we can avoid useless computation and execute time-effecient queries.

CHAPTER 3

BB-PLUS:AN APPROACH FOR SUBGRAPH ISOMORPHISM IN BIG GRAPH DATABASE

The BB-Plus approach consists of heuristics for automatically selecting the best matching order selection method using information of database (volatility and size) and query (type and size) as an input to improve the performance of subgraph isomorpishm queries. The BB-Plus uses these heuristics to find the best matching order that eliminates redundant candidate nodes at high level in order to reduce the search space and execute queries with less time in large graph datasets. The pseudocode for the approach is given in Algorithm 13.

Algorithm 13 The BB-Plus Approach

Input: Q : All query nodes, G : All data nodes,
$volatilityOfDb$: Volatility of database (Historical/Real-Time)
Output: M : All embeddings of Q in G

1: **procedure** BBPLUS(Q, G, $volatilityOfDb$)
2: **if** $IsBeforeQuery$ **then**
3: **if** $volatilityOfDb$!= $RealTime$ **then**
4: $StoreCandidateNodeSizes(Q, G)$
5: **else**
6: **if** $containsSameNode(Q)$ **then**
7: $O(Q) \leftarrow Q$
8: **else**
9: $O(Q) \leftarrow FindMatchingOrder(Q, volatilityOfDb))$
10: $M \leftarrow BBGraph(O(Q), G)$
11: **return** M

The rules for selecting the best matching order selection methods are created based on experiments. We created a *"Matching Order Selection"* dataset based on executing different kinds of queries on three different databases which are the *WorldCup*, *Pokec* and *Population* dataset. *"Matching Order Selection"* dataset contains information about queries such as size of the databases (which they are applied), size of query (nodes and edges) and type of query. Also it contains the matching order selection method that gives the best performance. After we obtain this dataset, we create a decision tree using this dataset and create rules to determine best matching order selection method based on the type and size of the query as shown in in Figure 3.1. The details of creation of this dataset and the decision tree is given in later in Chapter 3.8.

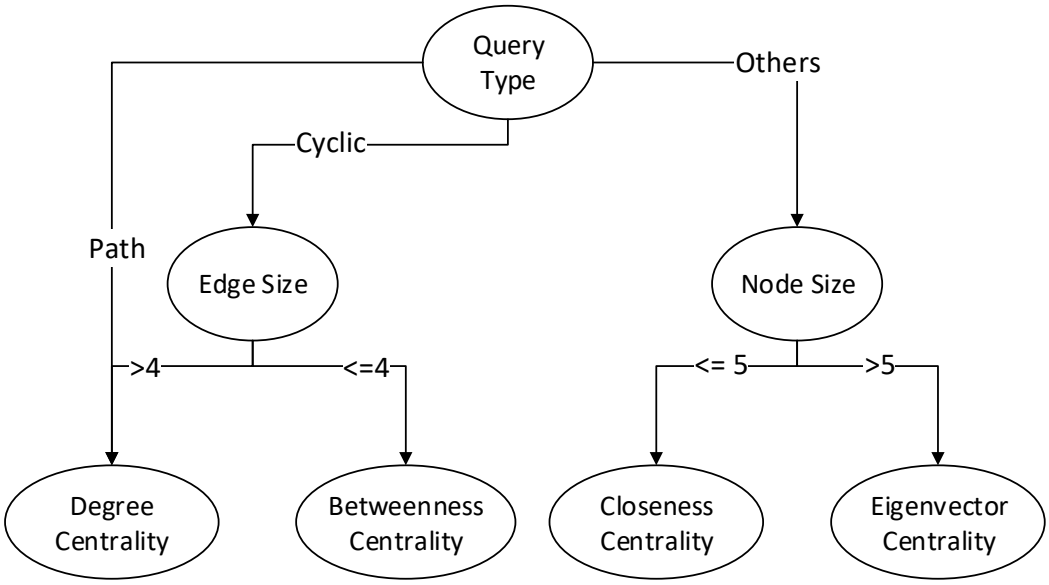

Figure 3.1: Decision Tree for Determining Best Matching Order

According to these rules, we shaped the our approach, the BB-Plus, to perform efficiently based on any queries on any databases. Therefore, we divided our approach into two parts which are *before querying* and *executing a query* which is described in the flow chart of the approach as shown in Figure 3.2.

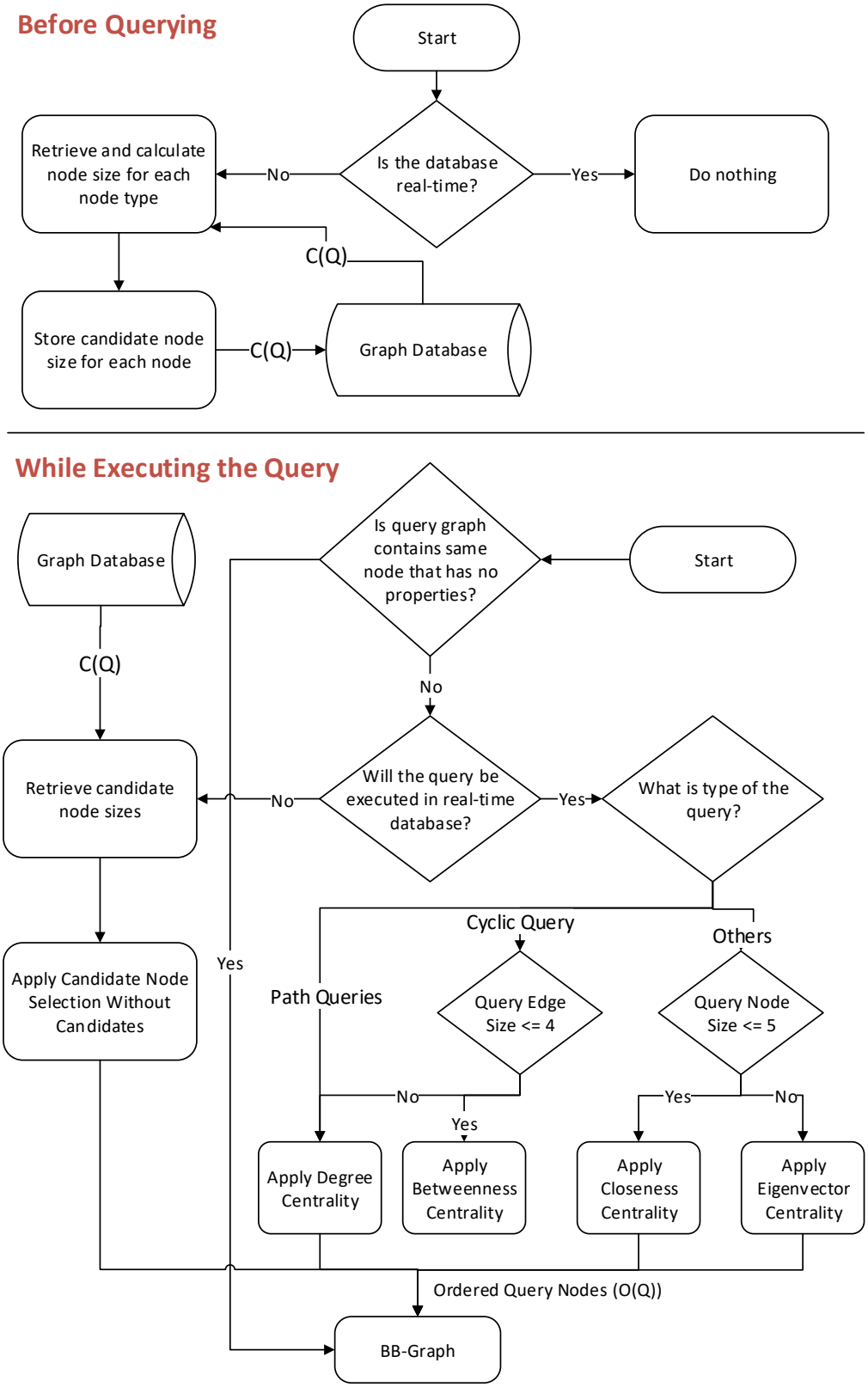

Figure 3.2: Flowchart of the BB-Plus Approach

Before querying, the approach uses the volatility of database as an input and checks the database is real-time or not. If the database is not real-time (historical), then the approach calls *StoreCandidateNodeSizes* method to retrieve and calculate the candidate node size for each query node and store them in the database for further uses. Since calculating candidate node size takes a lot of time, calculating it before querying helps to improving the performance of subgraph isomorpishm queries.

Algorithm 14 Store Candidate Node Sizes

Input: Q : All query nodes, G : All data nodes

Output: $C(Q)$: Candidate Node Size for Q

1: **procedure** STORECANDIDATENODE(Q, G)
2: **for all** $<u \in Q>$ **do**
3: $C_u \leftarrow \emptyset$ ▷ C_u = Candidate node size of u
4: $C_u \leftarrow$ FilterByLabel(u, G)
5: $C_u \leftarrow$ FilterByRelationships(u, C_u, G)
6: **if** u has property **then**
7: $C_u \leftarrow$ FilterByProperty(u, C_u, G)
8: Push C_u to $C(Q)$
9: Store $C(Q)$ in database

However, the approach does not do anything if the database is real-time. Because candidate node size could be changing based on the volatility of the database. Based on this, the approach does not use candidate node sizes in real-time databases. It uses a matching order that can be easily calculated at the time of query execution.

Algorithm 15 ContainsSameNode

Input: Q : All query nodes

Output: $containsSameNode$: true/false

1: **procedure** CONTAINSSAMENODE(Q)
2: **if** $distinctNodeLabels(Q) = 1$ **and** $distinctEdgeLabels(Q) = 1$ **and** $nodesHasSameProperties(Q) = 1$ **then**
3: **return true**
4: **else**
5: **return false**

While executing a query, the approach first checks if the query graph contains same nodes with *ContainsSameNode* method. If all the nodes are the same then the approach does not calculate matching order. Because, if all the nodes have the same importance and putting anyone into the first place does not change the query performance. Therefore, if the approach knows all nodes are the same, then it calls *BB-Graph* algorithm at the first place to find all exact matches.

If all the nodes are not the same, the approach calls *Find Matching Order* to look for the volatility of database and checks if the database is real-time or not again. If the database is not real-time, then the approach calls the *Candidate Node Selection Without Calculating Candidates* which is an extension of the *Candidate Node Selection* that uses already calculated candidate nodes size instead of calculating candidate node size while executing a query. Therefore, it eliminates the time for calculating candidate node size and performs more efficient queries.

Algorithm 16 Decision Tree

Input: Q : All query nodes

Output: $matchingOrder$: Matching Order Selection Method (DC-BC-CC-EC)

1: **procedure** DECISIONTREE(Q)
2: **if** $typeOfQuery(Q) = Path$ **then**
3: $matchingOrder = DegreeCentrality$
4: **else if** $typeOfQuery(Q) = Cyclic$ **then**
5: **if** $numberOfEdges(Q) <= 4$ **then**
6: $matchingOrder = BetweennessCentrality$
7: **else**
8: $matchingOrder = DegreeCentrality$
9: **else if** $typeOfQuery(Q) = Others$ **then**
10: **if** $numberOfNodes(Q) <= 5$ **then**
11: $matchingOrder = ClosenessCentrality$
12: **else**
13: $matchingOrder = EigenvectorCentrality$
14: **return** $matchingOrder$

Algorithm 17 Find Matching Order

Input: Q : All query nodes, $volatilityOfDb$: Volatility of database

Output: $O(Q)$: Ordered Query Graph

```
1:  procedure FINDMATCHINGORDER(Q)
2:      O(Q) ← ∅
3:      if volatilityOfDb == RealTime then
4:          if decisionTree(Q) = DegreeCentrality then
5:              O(Q) ← MosBasedOnDC(Q)
6:          else if decisionTree(Q) = BetweennessCentrality then
7:              O(Q) ← MosBasedOnBC(Q)
8:          else if decisionTree(Q) = ClosenessCentrality then
9:              O(Q) ← MosBasedOnCC(Q)
10:         else
11:             O(Q) ← MosBasedOnEC(Q)
12:     else
13:         Retrieve C(Q) from database
14:         O(Q) ← MosBasedOnCNSWithoutCandidates(Q, C(Q))
15:     return O(Q)
```

On the other hand, if the database is real-time, the approach calls *DecisionTree* to determine matching order selection based on the type and size of the query. *DecisionTree* method uses *TypeOfQuery* to determine the type of query. It uses *Tarjan's Algorithm* and *IsPathQuery* method to determine whether the type of query is *path query*, *cyclic query* or the *others* like recursive query.

After determining the type of query, the approach calls a matching order selection method that performs best with these query types. These matching order selection methods does not use data graph to give an order. Therefore, these matching order selection methods does not give more accurate order like *Candidate Node Selection*. However, they can be easily calculate and give great results when they match with the right types of query.

Algorithm 18 Type of Query

Input: Q : All query nodes

Output: $queryType$: Type of query

1: **procedure** TYPEOFQUERY(Q)
2: **if** $Tarjan(Q)$!= $\emptyset$ **then**
3: $queryType \leftarrow Cyclic$
4: **else if** $IsPathQuery(Q) == true$ **then**
5: $queryType \leftarrow Path$
6: **else**
7: $queryType \leftarrow Others$
8: **return** $queryType$

Algorithm 19 Is Path Query

Input: Q : All query nodes

Output: $isPathQuery$: true/false

1: **procedure** ISPATHQUERY(Q)
2: $N_1(Q) = 0$ ▷ $N_0(Q)$: Number of nodes that has 1 degree in Q
3: $N_2(Q) = 0$ ▷ $N_1(Q)$: Number of nodes that has 2 degree in Q
4: **for all** $<u \in Q>$ **do**
5: **if** $degree(u)$ = 1 **then** $N_1(Q)++$
6: **else if** $degree(u)$ = 2 **then** $N_2(Q)++$
7: **else**
8: **return** $false$
9: **if** $N_1(Q) = 2$ && $N_2(Q) = numberOfNodes(Q) - 2$ **then**
10: **return** $true$
11: **else**
12: **return** $false$

Therefore, if the type of query is a "path query", then the approach applies the *Degree Centrality* to get ordered query graph. If the type of query is a "cyclic query", then the approach looks for edge size of query graph. If the edge size is smaller than four, than it applies *Betweenness Centrality*. If not than it applies *Degree Centrality* to get ordered query graph. If the type of query is an another type of query such as "recursive query", then the approach looks for the node size of query graph. If the query graph node size is smaller than five, then the approach calls for the *Closeness Centrality*. If not, then it calls for *Eigenvector Centrality* to get ordered query graph.

If we take a more closer look to *TypeOfQuery* method, it uses *Tarjan's Algorithm* and *IsPathQuery* method to determine the query type. The method first applies *IsPathQuery* to check if the query is a type of path query. *IsPathQuery* checks the graph that has two nodes of degree 1 and the other n-2 nodes of degree 2. If the graph matches with that structure, then we consider our query graph is a path query. If it is not path query, then the method uses *Tarjan's algorithm* which is the most efficent algorithm to find if a graph contains any cycles. If Tarjan's Algorithm returns any cycles, then we consider our query graph is a cyclic query. If the query graph is not "path" or "cyclic" type, then the graph type is consider as "others". The pseudocode for *TypeOfQuery* and *IsPathQuery* methods can be seen in Algorithm 18 and 19, respectively.

Finally, after the approach getting the best matching order based on the information of database and query graph, it calls for the *BB-Graph* algorithm with ordered query graph to find all the exact matches. Normally, the *BB-Graph* algorithm does not use matching order selection. It matches query nodes with data nodes based on the order of nodes in the query. However, with *BB-Plus* approach, the algorithm works based on a calculated matching order. Therefore, the *BB-Plus* approach gives better results than *BB-Graph* for most of the time.

Let's explain the *BB-Plus* approach and show its difference with other algorithms with a simple example. Let's assume that we have query graph Q and data graph $G1$ as shown in Figure 3.3. We want to find exact matches of Q in G, which is only $(v7, v6, v8, v9, v10)$.

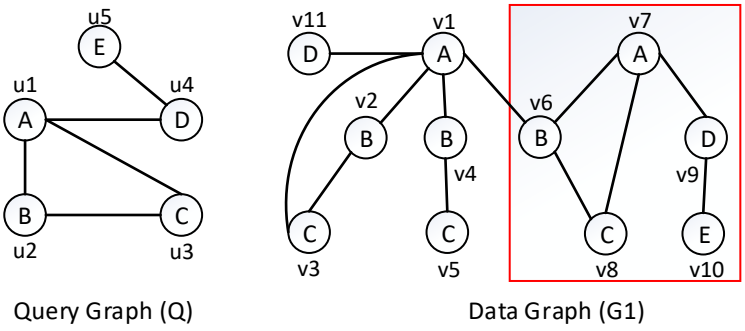

Figure 3.3: An Example of Difference of BB-Plus from the algorithms

First, let's see how many attempts are required for finding exact matches of Q in G with a branch-and-bound algorithm without calculating matching order selection. In this situtation, the algorithm uses a random matching order such as $(A - B - C - D - E)$. If the algorithm uses $(A - B - C - D - E)$ order, then it needs to search at $(v1, v2, v3, v11)$, $(v1, v4, v5)$, $(v1, v6, v8)$ and $(v7, v6, v8, v9, v10)$. Therefore, it requires 4 attemps to find the exact match. The algorithm loses time for cannot prunning out $(v1, v2, v3, v11)$, $(v1, v4, v5)$ and $(v1, v6, v8)$ at early steps of query.

On the other hand, if we use BB-Plus approach, we can exploit from matching order. Let's say BB-Plus approach decided to use *Degree Centrality* in this example. Therefore, the approach uses $(E-D-A-B-C)$ order. In this situation, the approach only search for $(v7, v6, v8, v9, v10)$ which only requires 1 attempt. $(E - D - A - B - C)$ order helps to pruning at early steps and query efficiently with reduced search space.

In this chapter, new matching order selection methods that we defined in the developing process of BB-Plus approach will be described and visualized with examples using the query graph Q and data graph G which are shown in Figure 3.4 and Figure 3.5 respectively. Matching order selection methods based on *Degree Centrality*, *Closeness Centrality*, *Betweenness Centrality*, *Eigenvector Centrality* and *Candidate Node Selection* are examined in this chapter.

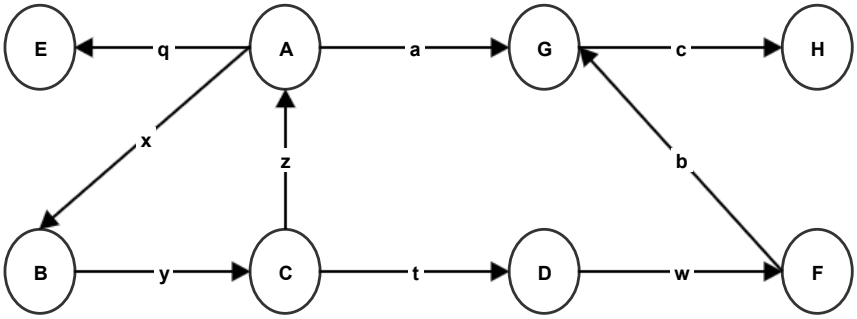

Figure 3.4: The Example Query Graph

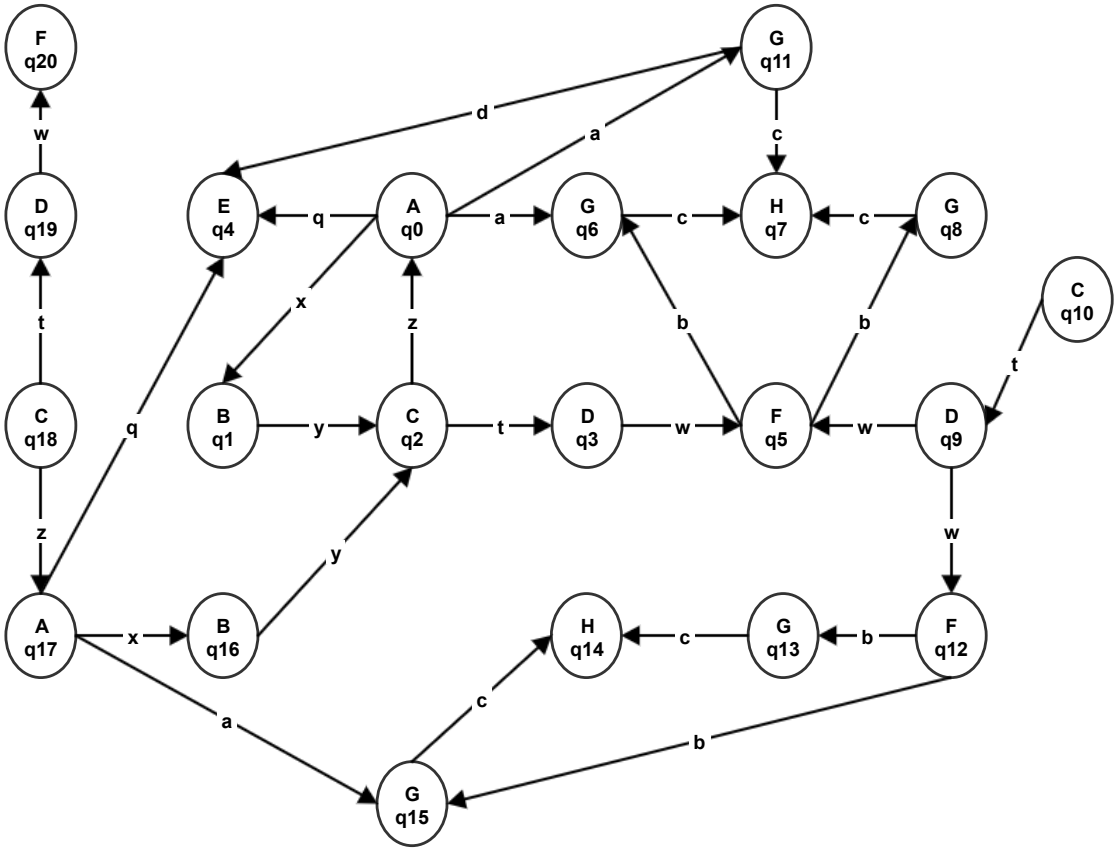

Figure 3.5: The Example Data Graph

3.1 Matching Order Selection Based On Degree Centrality

We use *Degree Centrality* to create a matching order in order to find exact matching couples $< u, v >$ efficiently by combining this order with a branch-and-bound algorithm. The pseudocode of calculating a *Matching Order Selection based on Degree Centrality* is given in Algorithm 20.

Algorithm 20 Matching Order Selection Based On Degree Centrality

Input: Q : All query nodes, G : All data nodes

Output: $O(Q)$: Query nodes in matching order

1: **procedure** MOSBYDC(Q, G)
2: $O(Q) \leftarrow \emptyset$ ▷ $O(Q)$: Query nodes in matching order
3: $O_d(Q) \leftarrow DegreeCentrality(Q)$ ▷ $O_d(Q)$: List of degree of query nodes
4: $O_d(Q) \leftarrow Sort_{asc}(O_d(Q))$ ▷ Sort $O_d(Q)$ in ascending order
5: $O(Q) \leftarrow CreateMatchingOrder(O_d(Q))$
6: **return** $O(Q$

Algorithm 21 Create Matching Order

Input: Q : Query nodes

Output: $O(Q)$: Ordered query nodes

1: **procedure** CREATEMATCHINGORDER(Q)
2: $O(Q) \leftarrow \emptyset$
3: **for all** $<u \in Q>$ **do**
4: $nextNode \leftarrow u$
5: **for all** $<u' \in Q>$ **do**
6: **if** $order(nextNode) = order(u')$ **then**
7: **if** $order_{min}(neighbor(nextNode)) > order_{min}(neighbor(u'))$ **then**
8: $nextNode \leftarrow u'$
9: Add $nextNode$ to $O(Q)$
10: Remove $nextNode$ from Q
11: **return** $O(Q)$

First, the algorithm calls the *DegreeCentrality* method that given in Algorithm 1 to obtain degree centrality values for all query nodes in query graph Q. *DegreeCentrality* looks incoming and outgoing edges of each nodes to find degrees of each nodes and assings degree centrality value to each query node. After calling *DegreeCentrality*, the algorithm sorts query nodes ascendingly by their degree centrality values to obtain $O_d(Q)$ and become ready for calling the *CreateMatchingOrder* method which is described in Algorithm 21. The method looks at $O_d(Q)$ and if there are nodes with same minimum degree then the algorithm prioritize the one that has neighbor nodes with minimum degree centrality value and creates an order that called $O(Q)$.

The *BB-Plus* approach uses *Matching Order Selection based on Degree Centrality* when the database is real-time and the query type is path query. Also it uses for cyclic queries that has more than 4 edges. The approach first find a matching order with *Matching Order Selection based on Degree Centrality* and then call *BB-Graph* algorithm to find the exact matches efficiently. The algorithm starts with the node that has the first place in the order and it continues to branch from first matched node in the order that connected to the already matched query nodes. The algorithm branches for all query nodes in the order until all of them find matches in data graph.

Let's assume that we want to find the exact matches of Q query graph in data graph G using the *BB-Plus* approach and the approach uses *Matching Order Selection based on Degree Centrality* for efficient querying. The algorithm first calls *DegreeCentrality* to find the degree centrality values of query nodes which are $(4, 2, 3, 2, 1, 2, 3, 1)$ for (A, B, C, D, E, F, G, H) as explained in Figures 3.6 and 3.7.

After finding the degrees of each query nodes, the algorithm orders the query nodes in ascending order and obtain $O_d(Q) = (E, H, B, D, F, C, G, A)$. Because of the degree of E and H are the same, the algorithm looks for the degree of their neighbor nodes. Since the degree of G (the neighbor of H) is smaller than A (the neighbor of E), the algorithm prioritize H when putting it into the order. Then, the algorithm continues with G which is connected to H. Then it adds F, D, C, B, A and E respectively to the order based on their degrees. Therefore, the final version of the matching order for the algorithm is defined as $O(Q) = (H, G, F, D, C, B, A, E)$. All the steps for finding *Matching Order based on Degree Centrality* can be seen in Figure 3.8, 3.9.

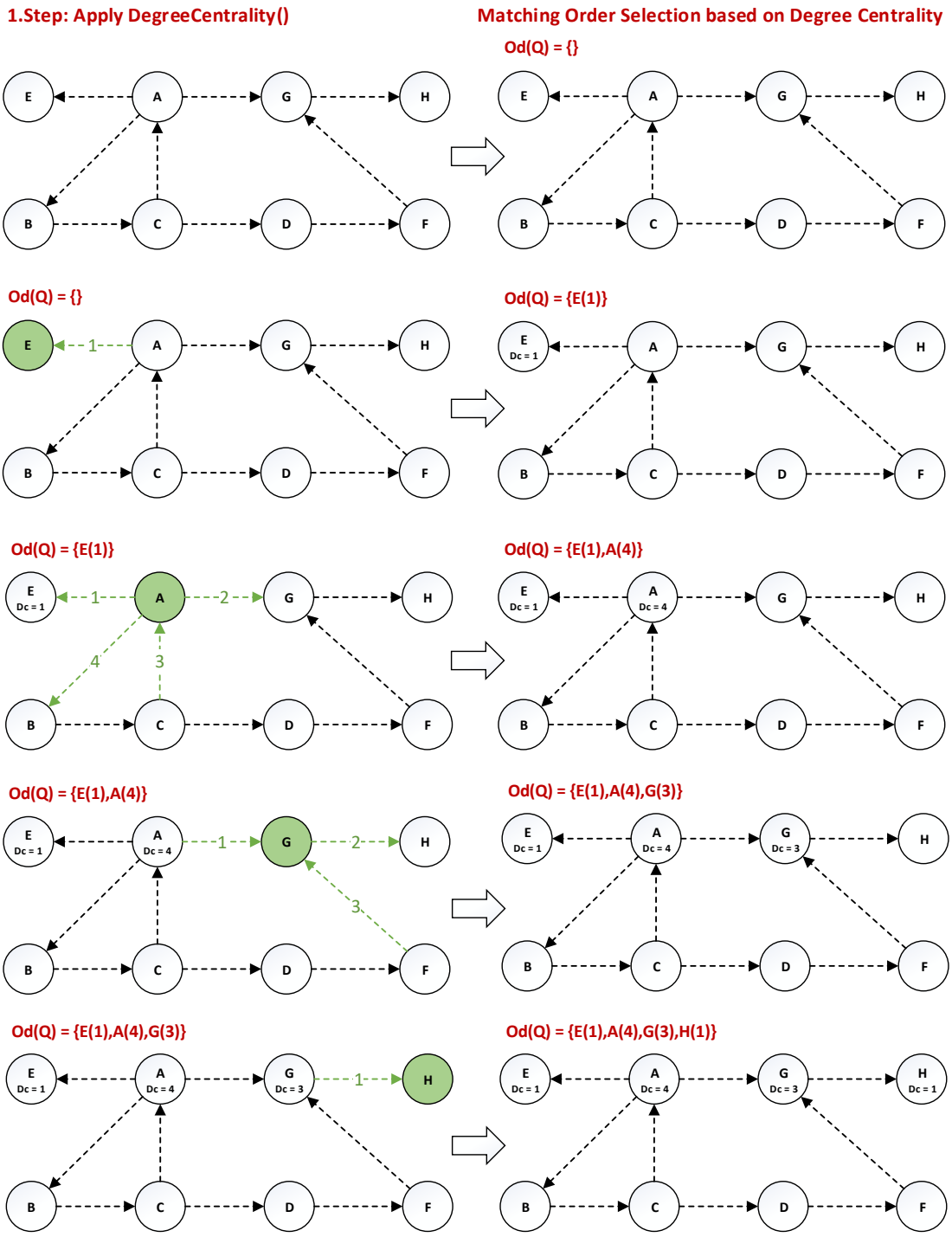

Figure 3.6: Calculating Degree Centrality for Query Graph Q (Part-1)

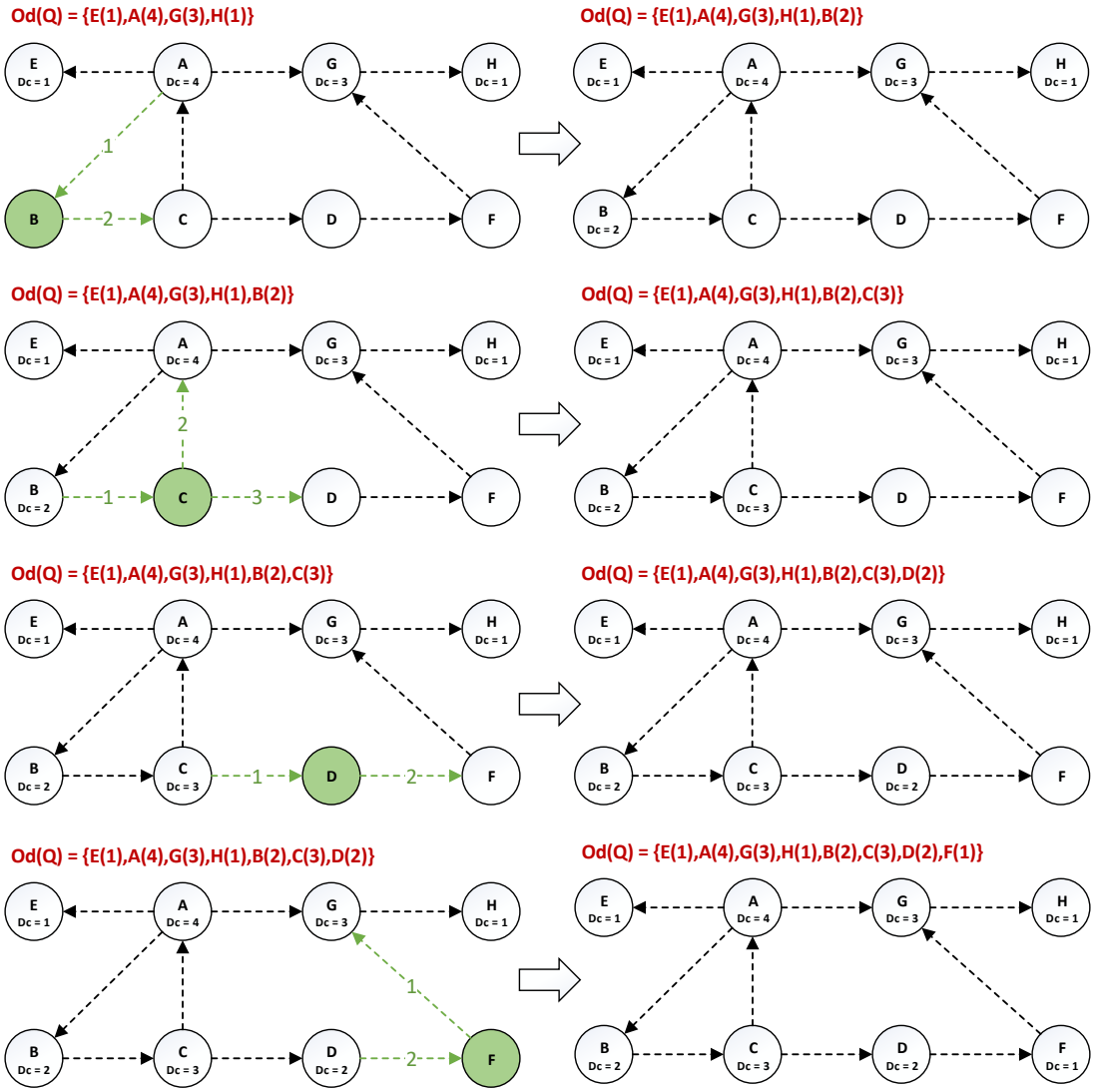

Figure 3.7: Calculating Degree Centrality for Query Graph Q (Part-2)

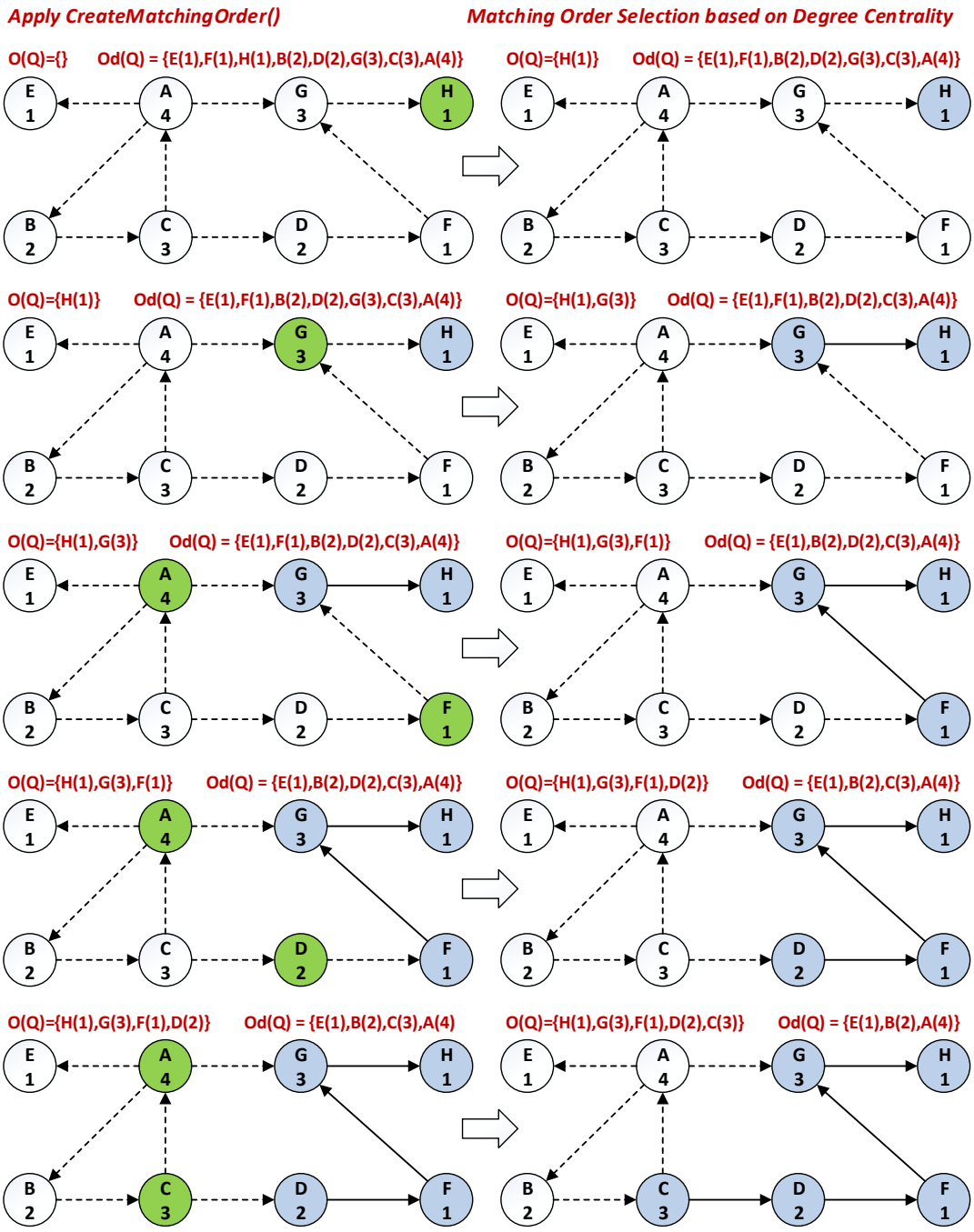

Figure 3.8: An Example of Creating Matching Order with Degree Centrality Method (Part-1)

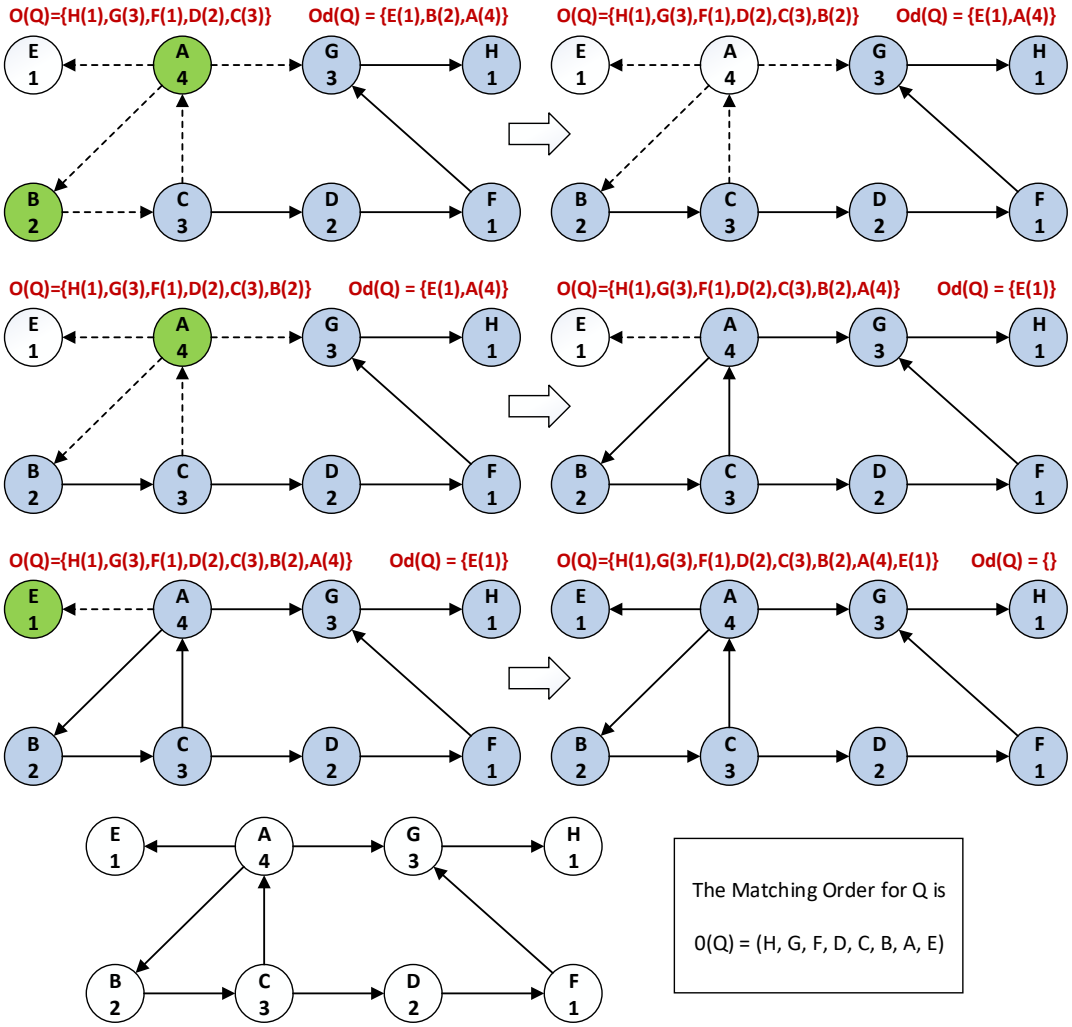

Figure 3.9: An Example of Creating Matching Order with Degree Centrality Method (Part-2)

After finding the matching order for the Q, the approach calls the *BB-Graph* algorithm with the ordered query graph $O(Q)$ to find all the exact matches of the data graph G. As shown in Figure 3.10, *BB-Graph* branches for two graphs which are $G1$ and $G2$. Although $G1$ prunes out after cannot find a matching for the query node C, $G2$ gives the exact solution for Q.

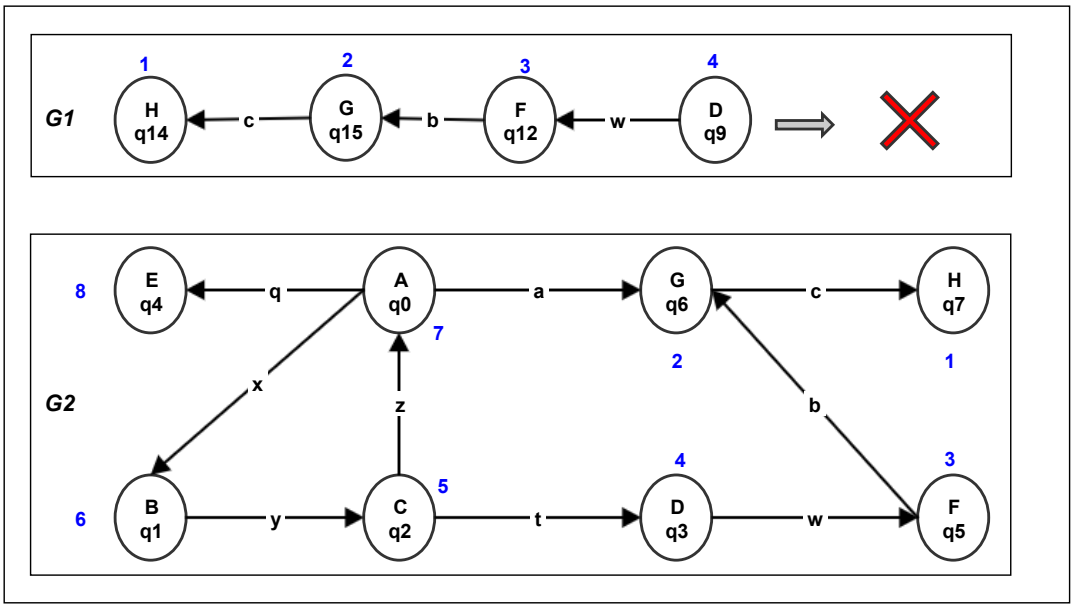

Figure 3.10: Finding Matches for Q in G with the BB-Graph

3.2 Matching Order Selection Based On Closeness Centrality

We use *Closeness Centrality* to create a matching order in order to find exact matching couples $< u, v >$ efficiently by combining this order with a branch-and-bound algorithm. The pseudocode of calculating a *Matching Order Selection based on Closeness Centrality* is given in Algorithm 22.

At the first step, the algorithm calls the *ClosenessCentrality* method that given in Algorithm 2 to generate closeness centrality values for all query nodes. *ClosenessCentrality* calculate the sum of the shortest paths between the query node and the other nodes to find closeness value of each query nodes and assigns the closeness centrality value to each query node. After calling *ClosenessCentrality*, the algorithm sorts query nodes in ascending order by their closeness centrality value to obtain $O_c(Q)$

Algorithm 22 Matching Order Selection Based On Closeness Centrality

Input: Q : All query nodes, G : All data nodes

Output: $O(Q)$: Query nodes in matching order

1: **procedure** MOSBASEDONCC(Q, G)
2: $O(Q) \leftarrow \emptyset$ ▷ $O(Q)$: Query nodes in matching order
3: $O_c(Q) \leftarrow ClosenessCentrality(Q)$ ▷ $O_c(Q)$: List of query nodes that assigned closeness value
4: $O_c(Q) \leftarrow Sort_{asc}(O_c(Q))$ ▷ Sort $O_c(Q)$ with ascending order
5: $O(Q) \leftarrow CreateMatchingOrder(O_c(Q))$
6: **return** $O(Q)$

and become ready for calling the *CreateMatchingOrder* method which is described in Algorithm 21. The method looks at $O_c(Q)$ and if there are nodes with same minimum closeness value then the algorithm picks the one that has neighbor nodes with minimum closeness centrality value and puts them into an order that called $O(Q)$.

The *BB-Plus* approach uses *Matching Order Selection based on Closeness Centrality* when the database is real-time and the query type is defined as others (such as recursive queries) and query node size is bigger than 5. The approach first find a matching order with *Matching Order Selection based on Closeness Centrality* and then call *BB-Graph* algorithm to find the exact matches efficiently. The *BB-Graph* algorithm starts with the node that has the first place in the order and continues to branch from it based on the order until finding all the exact matches for the query graph in the data graph.

Let's assume that we want to find the exact matches of Q in G using the *BB-Plus* approach and the approach uses *Matching Order Selection based on Closeness Centrality* for efficient querying. The algorithm first calls *ClosenessCentrality* to find the closeness centrality values of query nodes. In order to that, the algorithm first find shortest distance between all the nodes as shown in Table 3.1. For all the query nodes, the algorithm calculate the total shortest distance and divide it to 1 like shown in Table 3.2. Therefore, it obtains the closeness value for the query nodes called $O_c(Q)$ which consists of (A, B, C, D, E, F, G, H) with $(0.10, 0.07, 0.08, 0.07, 0.06, 0.07, 0.09, 0.05)$ values. After obtaining the closeness centrality values for query graph Q, the algorithm sorts $O_c(Q)$ in ascending order and the $O_c(Q)$ become (H, E, B, D, F, C, G, A).

The algorithm calls *CreateMatchingOrder* method to get matching order $O(Q)$ after obtaining the $O_c(Q)$. The algorithm first pushes H, the node with minimum closeness value, into $O(Q)$ and then push G because it is the only node that connected with H. It continues with F which is the node with the minimum closeness value that connected to G. The algorithm push D, C, B, A and E into $O(Q)$ respectively with following the minimum closeness value rule. Therefore, the final version of the matching order for the algorithm is defined as $O(Q) = (H, G, F, D, C, B, A, E)$. The steps for finding *Matching Order based on Closeness Centrality* can be seen in Figure 3.11, 3.12.

Table 3.1: Distance matrix of query graph Q

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**
A	-	1	1	2	1	2	1	2
B	1	-	1	2	2	3	2	3
C	1	1	-	1	2	2	2	3
D	2	2	1	-	3	1	2	3
E	1	2	2	3	-	3	2	3
F	2	3	2	1	3	-	1	2
G	1	2	2	2	2	1	-	1
H	2	3	2	3	3	2	2	-

Table 3.2: Calculating closeness centrality of example query graph G

Nodes (v)	**Total # of shortest paths between v and the other nodes**	$C_c(v)$ **(Closeness Centrality of v)**
A	1 + 1 + 2 + 1 + 2 + 1 + 2 = 10	1 / 10 = 0.10
B	1 + 1 + 2 + 2 + 3 + 2 + 3 = 14	1 / 14 = 0.07
C	1 + 1 + 1 + 2 + 2 + 2 + 3 = 12	1 / 12 = 0.08
D	2 + 2 + 1 + 3 + 1 + 2 + 3 = 14	1 / 14 = 0.07
E	1 + 2 + 2 + 3 + 3 + 2 + 3 = 16	1 / 16 = 0.06
F	2 + 3 + 2 + 1 + 3 + 1 + 2 = 14	1 / 14 = 0.07
G	1 + 2 + 2 + 2 + 2 + 1 + 1 = 11	1 / 11 = 0.09
H	2 + 3 + 2 + 3 + 3 + 2 + 2 = 17	1 / 17 = 0.05

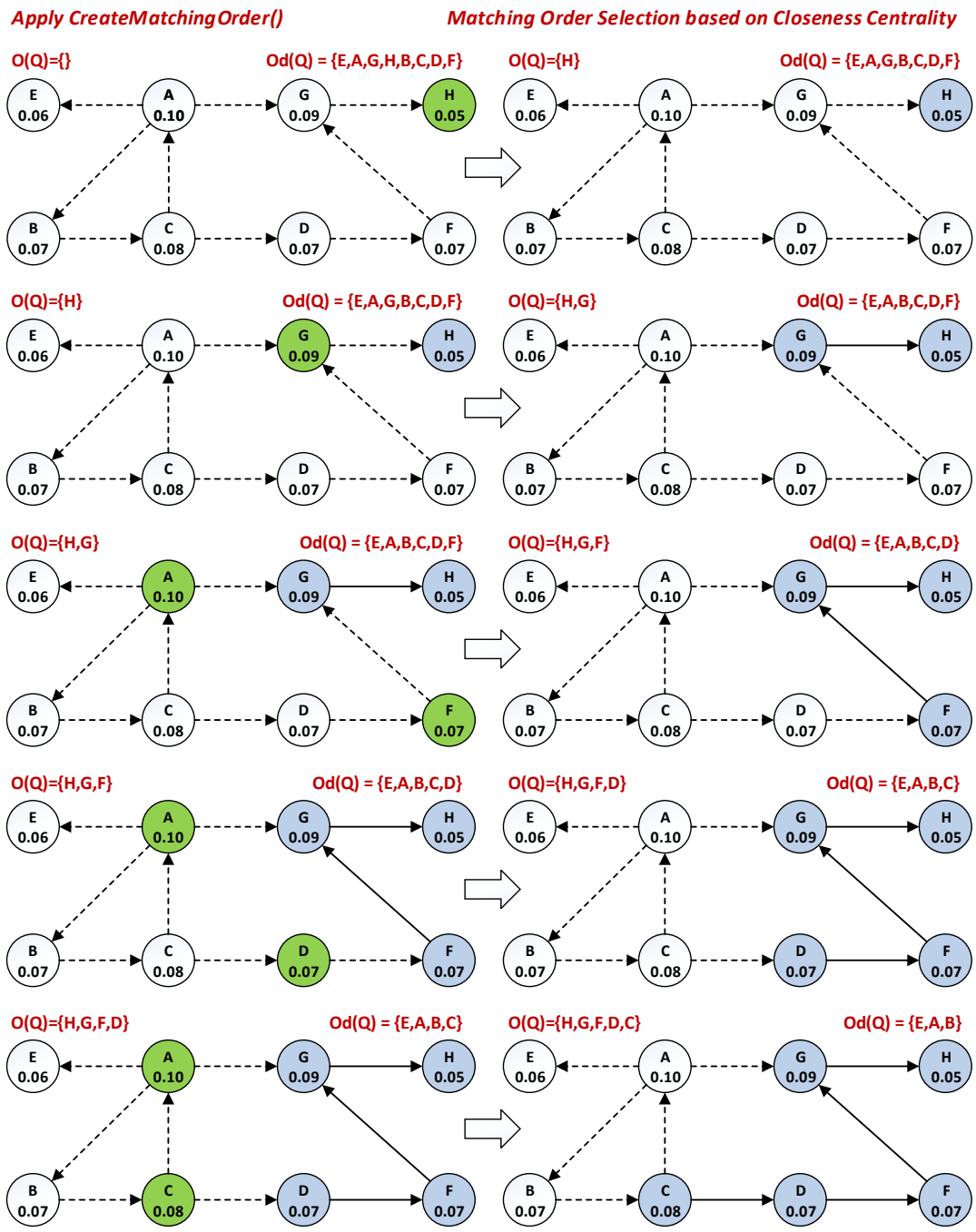

Figure 3.11: An Example of Creating Matching Order with Closeness Centrality Method (Part-1)

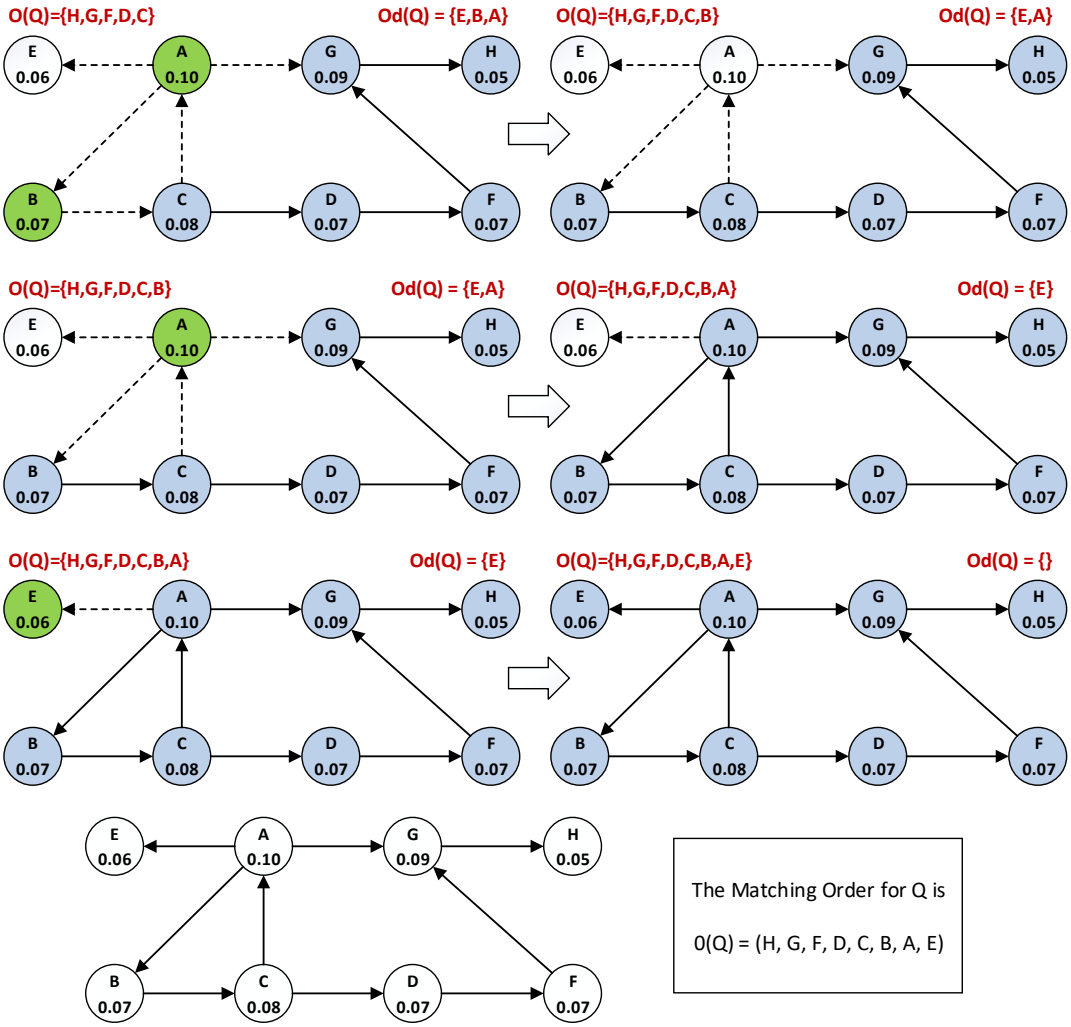

Figure 3.12: An Example of Creating Matching Order with Closeness Centrality Method (Part-2)

After finding the matching order for the Q, the approach calls the *BB-Graph* algorithm with ordered query graph $Q(O)$ to find all the exact matches of the data graph G. As shown in Figure 3.13, the approach branches for two graphs which are $G1$ and $G2$. Although $G1$ prunes out after cannot find a matching for the query node C, $G2$ gives the exact solution for Q.

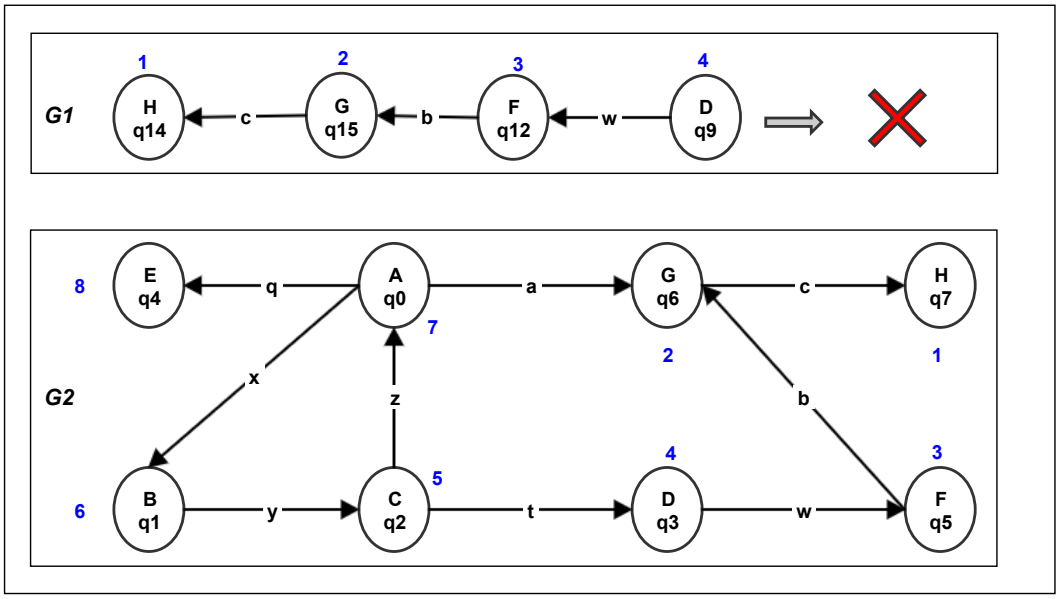

Figure 3.13: An Example of Finding All Exact Matches with BB-Graph

3.3 Matching Order Selection Based On Betweenness Centrality

We use *Betweenness Centrality* to create a matching order in order to find exact matching couples $< u, v >$ efficiently by combining this order with a branch-and-bound algorithm. The pseudocode of calculating a *Matching Order Selection based on Betweenness Centrality* is given in Algorithm 23.

At the first step, the algorithm calls the *BetweennessCentrality* method that given in Algorithm 3 to assign betweenness centrality values for all query nodes in query graph Q. *BetweennessCentrality* tries to find how many times a node is included on a shortest path that is defined among all the nodes in the query nodes and assigns this value to each query nodes to generate betweenness centrality values.

Algorithm 23 Matching Order Selection Based On Betweenness Centrality

Input: Q : All query nodes, G : All data nodes

Output: $O(Q)$: Query nodes in matching order

1: **procedure** MOSBASEDONBC(Q, G)
2: $O(Q) \leftarrow \emptyset$ ▷ $O(Q)$: Query nodes in matching order
3: $O_b(Q) \leftarrow BetweennessCentrality(Q)$ ▷ $O_b(Q)$: List of query nodes that assigned betweenness value
4: $O_b(Q) \leftarrow Sort_{asc}(O_b(Q))$ ▷ Sort $O_b(Q)$ with ascending order
5: $O(Q) \leftarrow CreateMatchingOrder(O_b(Q))$
6: **return** $O(Q)$

After calling *BetweennessCentrality*, the algorithm sorts each query nodes in ascending order by their betweenness centrality value to obtain $O_b(Q)$ and become ready for calling the *CreateMatchingOrder* method which is described in Algorithm 21. The method looks at $O_b(Q)$ and it adds a start query node with the minimum betweenness value into matching order list $O(Q)$ and continue to add nodes that it is connected to the already added nodes. When adding nodes to the matching order, if the algortihm stuck between nodes that has the same betweenness value, it selects the one that has neighbors with the minimum betweenness value.

The *BB-Plus* approach uses *Matching Order Selection based on Betweenness Centrality* when the database is real-time, the query type is defined as cyclic query with edge size smaller than 4. The approach first find a matching order with *Matching Order Selection based on Betweenness Centrality* and then call *BB-Graph* algorithm to find the exact matches efficiently. The *BB-Graph* algorithm starts with the node that has the first place in the order and continues to branch from it based on the order until finding all the exact matches for the query graph in the data graph.

Let's assume that we want to find the exact matches of Q in G using the *Matching Order Selection based on Closeness Centrality* and *BB-Graph*. At the first step, the algortihm applies the *BetweennessCentrality* to assign betweenness value to each query node. In order to that, the algorithm tries to find all the shortest path in the query graph and determine how many times a nodes pases through them which is shown in Table 3.3. Finally, it finds the betweenness value $O_b(Q)$ for each nodes

Table 3.3: Calculating betweenness centrality of example query graph G

Nodes (v)	**Shortest path passes through the node**	$C_b(v)$ **(Betweenness Centrality of** v**)**
A	B-E (B-A-E), B-G (B-A-G), B-H (B-A-G-H), C-E (C-A-E), C-G (C-A-G), C-H (C-A-G-H), D-E (D-C-A-E), E-F (E-A-G-F), E-G (E-A-G), E-H (E-A-G-H)	10
B	-	0
C	A-D (A-C-D), B-D (B-C-D), B-F (B-C-D-F), D-E (D-C-A-E)	4
D	B-F (B-C-D-F), C-F (C-D-F)	2
E	-	0
F	D-G (D-F-G), D-H (D-F-G-H)	2
G	A-F (A-G-F), A-H (A-G-H), B-H (B-A-G-H), C-H (C-A-G-H), D-H (D-F-G-H), E-F (E-A-G-F), E-H (E-A-G-H), F-H (F-G-H)	8
H	-	0

which are $(21, 0, 17, 3, 0, 4, 17, 0)$ for (A, B, C, D, E, F, G, H) respectively. After finding the betweenness value of query nodes, the algorithm sorts them ascendingly by their betweenness value and put them in $O_b(Q)$ list. Therefore, the $O_b(Q)$ is become (B, E, H, D, F, C, G, A).

The algorithm calls *CreateMatchingOrder* method to get matching order $O(Q)$. Because of the similarity of betweenness value of the query nodes B, E, H, the algorithm checks the minimum betweenness value of neighbours of (B, E, H) which are $(17, 21, 17)$ and adds B into $O(Q)$ because B is the first node that has neighbors with the minimum betweenness value. The algorithm keeps searching for a query node that branching from B and continues to adding C, D, F, G, H, A, E query nodes to $O(Q)$ with respect to their betweenness value. The steps for finding *Matching Order based on Betweenness Centrality* can be seen in Figure 3.14, 3.15.

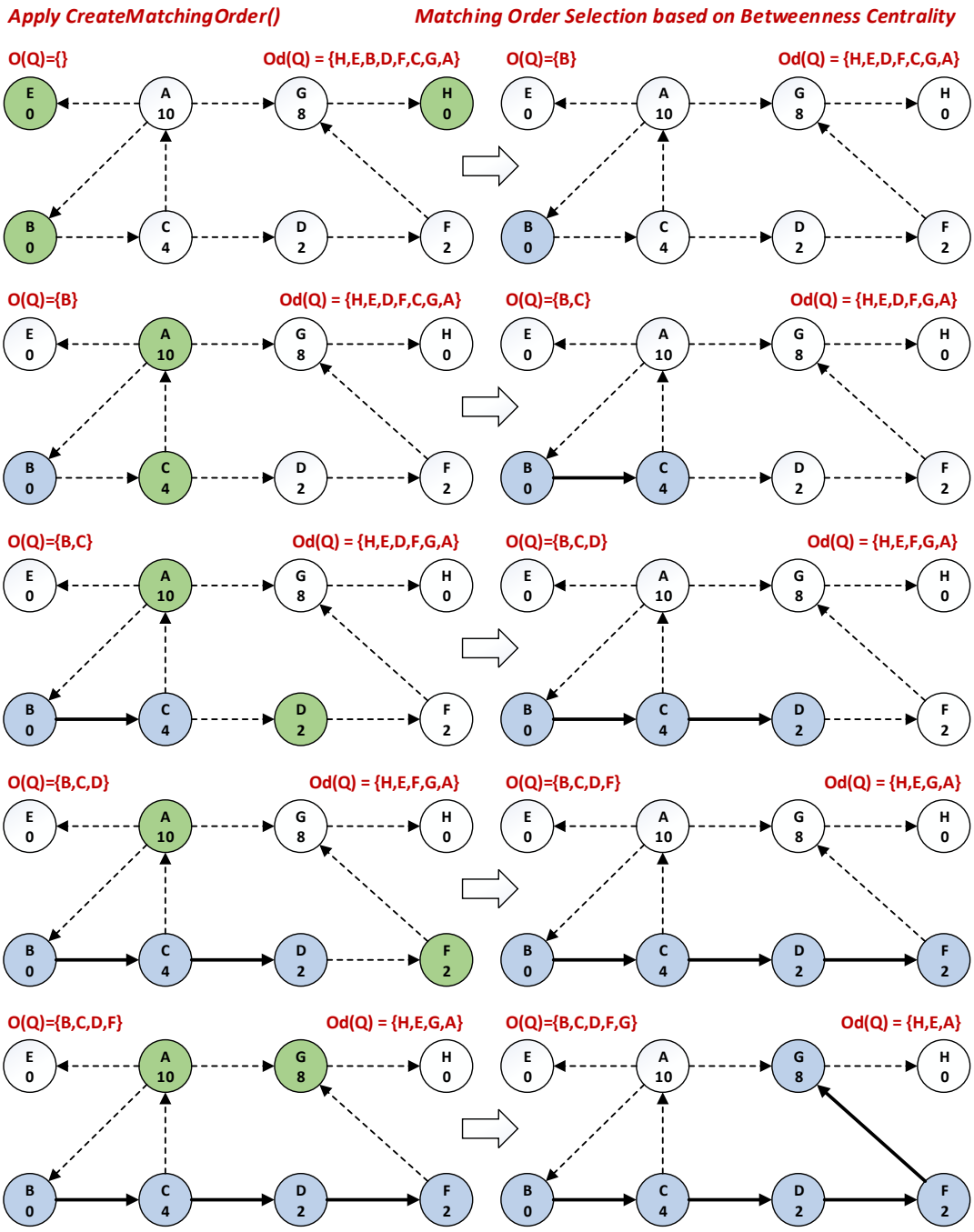

Figure 3.14: An Example of Creating Matching Order with Betweenness Centrality Method (Part-1)

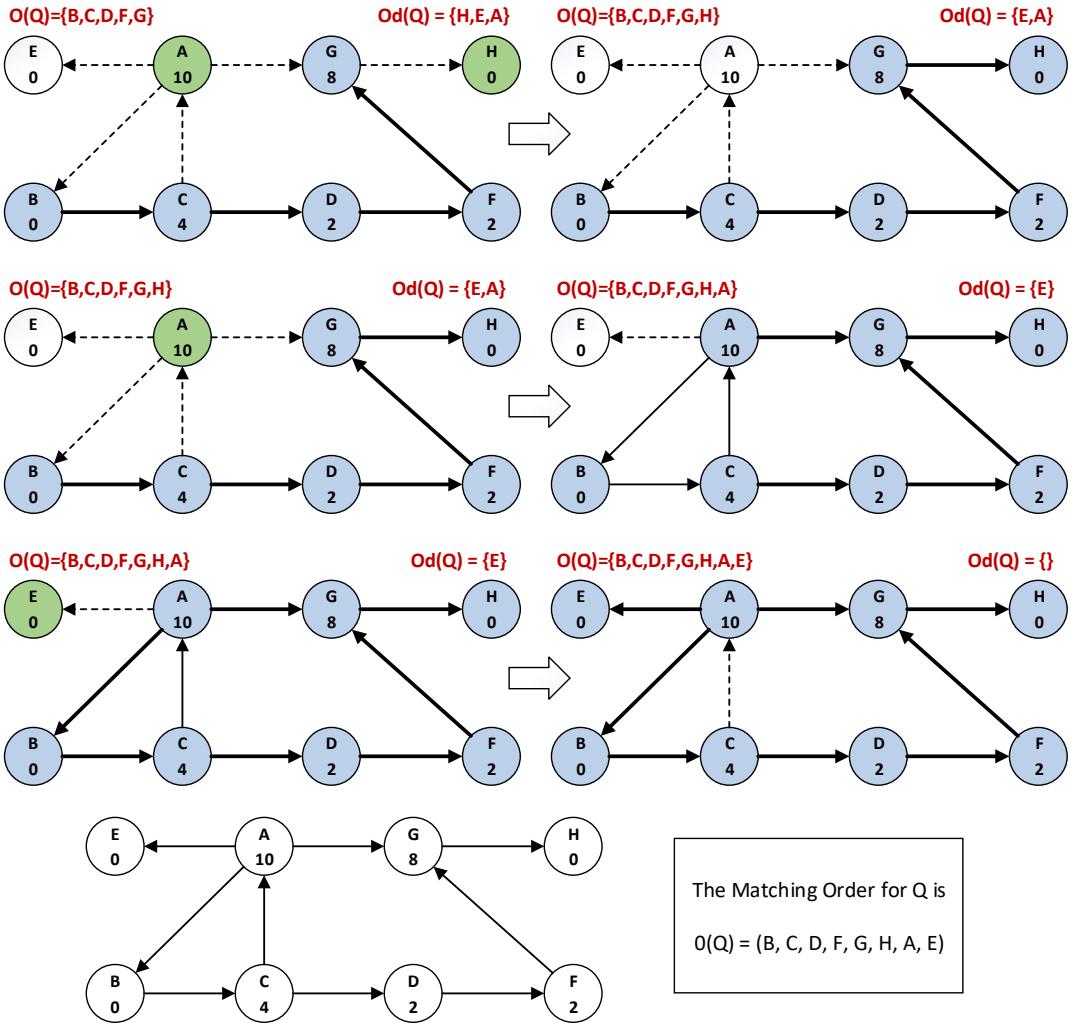

Figure 3.15: An Example of Creating Matching Order with Betweenness Centrality Method (Part-1)

At the final step, the algorithm calls the *BB-Graph* algorithm with ordered query graph $O(Q)$ and the data graph G. As shown in Figure 3.16, the algorithm branches for two graph which are $G1$ and $G2$. $G1$ prunes out after cannot find a matching for query node A. However, all query nodes find an exact matches in $G2$.

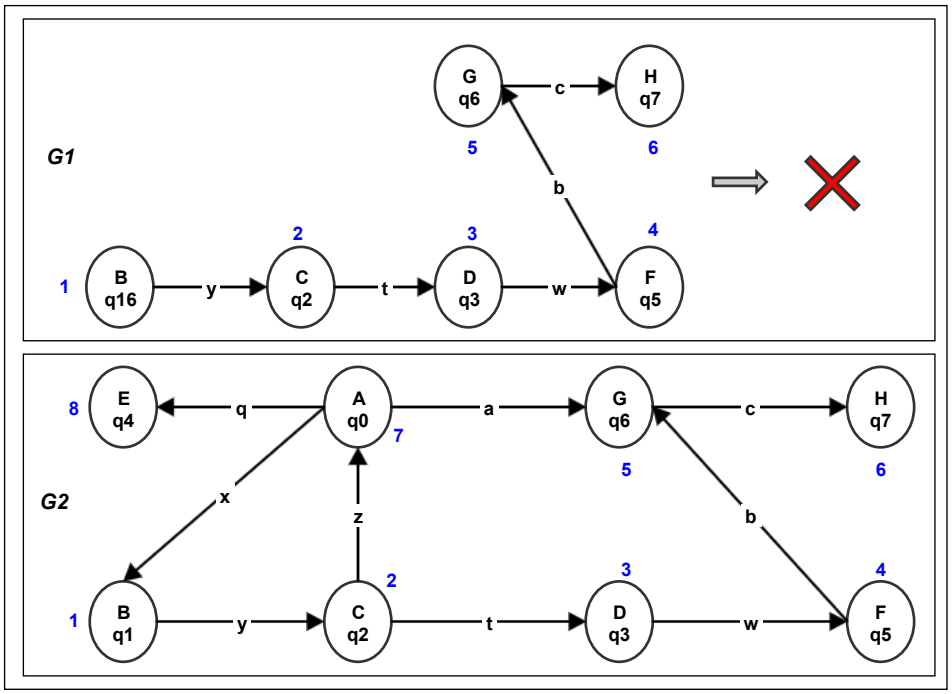

Figure 3.16: An Example of Finding All Exact Matches with the BB-Graph

3.4 Matching Order Selection Based On Eigenvector Centrality

We use *Eigenvector Centrality* to create a matching order in order to find similar patterns of query graph Q in data graph G efficiently by combining this order with a branch-and-bound algorithm. The pseudocode of calculating a *Matching Order Selection based on Eigenvector Centrality* is given in Algorithm 24.

The algorithm firstly applies the *EigenvectorCentrality* method that is given in Algorithm 4 to map eigenvector values with each query node in query graph. *Eigenvector-Centrality* tries to calculate the the measure of the influence of each query node in the graph. In order to do that, it creates and adjanceny matrix, calculates the eigenvalue and eigenvector values for the nodes based on this matrix and assigns eigenvector

Algorithm 24 Matching Order Selection Based On Eigenvector Centrality

Input: Q : All query nodes, G : All data nodes

Output: $O(Q)$: Query nodes in matching order

1: **procedure** MOSBASEDONEC(Q, G)
2: $O(Q) \leftarrow \emptyset$ ▷ $O(Q)$: Query nodes in matching order
3: $O_e(Q) \leftarrow EigenvectorCentrality(Q)$ ▷ $O_e(Q)$: List of query nodes that assigned eigenvector value
4: $O_e(Q) \leftarrow Sort_{asc}(O_e(Q))$ ▷ Sort $O_e(Q)$ with ascending order
5: $O(Q) \leftarrow CreateMatchingOrder(O_e(Q))$
6: **return** $O(Q)$

centrality values based on the eigenvector value that comes from maximum eigenvalue.

After calling *EigenvectorCentrality*, the algorithm sorts each query nodes in ascending order by their eigenvector centrality value to obtain $O_e(Q)$ and become ready for calling the *CreateMatchingOrder* method which is described in Algorithm 21. The method looks at $O_e(Q)$ and it adds a start query node with the minimum eigenvector value into matching order list $O(Q)$ and continue to add nodes that it is connected to the already added nodes. When adding nodes to the matching order, if the algortihm stuck between nodes that has the same betweenness value, it selects the one that has neighbors with the minimum eigenvector value.

The *BB-Plus* approach uses *Matching Order Selection based on Eigenvector Centrality* when the database is real-time and the query type is defined as "others" and the query node size is bigger than 5. The approach first find a matching order with *Matching Order Selection based on Eigenvector Centrality* and then call *BB-Graph* algorithm to find the exact matches efficiently. The *BB-Graph* algorithm starts with the node that has the first place in the order and continues to branch from it based on the order until finding all the exact matches for the query graph in the data graph.

For the query graph Q and data graph G example, let's explain *BB-Plus* approach and how it uses *Matching Order Selection based on Eigenvector Centrality* for efficient querying. In the beginning, the algorithm calls the *EigenvectorCentrality* method to find the eigenvector centrality values which is called $O_e(Q)$ for each query node

which are $(0.54, 0.39, 0.46, 0.27, 0.21, 0.24, 0.35, 0.35, 0.13)$, for (A, B, C, D, E, F, G, H) respectively. In order to calculate eigenvector centrality, the method creates an adjaceny matrix for query graph Q as shown in Table 3.4 and then applies all the steps that shown in Figure 3.17. After finding the centrality values, the algorithm sorts $O_e(Q)$ by ascending order and $O_e(Q)$ becomes (H, E, F, D, G, B, A, C).

Table 3.4: Adjacency matrix of query graph Q

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**
A	-	1	1	0	1	0	1	0
B	1	-	1	0	0	0	0	0
C	1	1	-	1	0	0	0	0
D	0	0	1	-	0	1	0	0
E	1	0	0	0	-	0	0	0
F	0	0	0	1	0	-	1	0
G	1	0	0	0	0	1	-	1
H	0	0	0	0	0	0	1	-

In order to obtain a matching order for the query graph, the algorithm applies *CreateMatchingOrder* and it finds $O(Q) = (H, G, F, D, C, B, A, E)$. Similar to the other algorithms, it looks at the $O_e(Q)$ and change the node places based on their eigenvector values. Nodes with minimum eigenvector values comes first in the method. However if the nodes have the same eigenvector centrality value, then it picks the one that has neighbors with minimum eigenvector centrality value. Each step for finding a matching order with the *Matching Order based on Eigenvector Centrality* is given in Figure 3.18, 3.19.

Finally, the algorithm calls the *BB-Graph* algorithm with ordered query graph $Q(Q)$ and the data graph G. As shown in Figure 3.20, the algorithm braches for two graph which are $G1$ and $G2$. $G1$ prunes out after cannot find a matching for query node A. However, all query nodes find an exact matches in $G2$.

$$A=\begin{bmatrix}0&1&1&0&1&0&1&0\\1&0&1&0&0&0&0&0\\1&1&0&1&0&0&0&0\\0&0&1&0&0&1&0&0\\1&0&0&0&0&0&0&0\\0&0&0&1&0&0&1&0\\1&0&0&0&0&1&0&1\\0&0&0&0&0&0&1&0\end{bmatrix}$$

$$(A-\lambda I)C_e=0 \Rightarrow \begin{bmatrix}0-\lambda&1&1&0&1&0&1&0\\1&0-\lambda&1&0&0&0&0&0\\1&1&0-\lambda&1&0&0&0&0\\0&0&1&0-\lambda&0&1&0&0\\1&0&0&0&0-\lambda&0&0&0\\0&0&0&1&0&0-\lambda&1&0\\1&0&0&0&0&1&0-\lambda&1\\0&0&0&0&0&0&1&0-\lambda\end{bmatrix}\begin{bmatrix}u1\\u2\\u3\\u4\\u5\\u6\\u7\\u8\end{bmatrix}=\begin{bmatrix}0\\0\\0\\0\\0\\0\\0\\0\end{bmatrix}$$

$$\det(A-\lambda I)=\begin{vmatrix}0-\lambda&1&1&0&1&0&1&0\\1&0-\lambda&1&0&0&0&0&0\\1&1&0-\lambda&1&0&0&0&0\\0&0&1&0-\lambda&0&1&0&0\\1&0&0&0&0-\lambda&0&0&0\\0&0&0&1&0&0-\lambda&1&0\\1&0&0&0&0&1&0-\lambda&1\\0&0&0&0&0&0&1&0-\lambda\end{vmatrix}=0$$

$$\Rightarrow \lambda_1=2.60 \quad \lambda_2=-2.06 \quad \lambda_3=-1.72 \quad \lambda_4=1.34 \quad \lambda_5=-1.60 \quad \lambda_6=1.70 \quad \lambda_7=0.37 \quad \lambda_8=0.21$$

*The largest eigenvalue is **2.60**. So the corresponding eigenvector:*

$$\begin{bmatrix}-2.6&1&1&0&1&0&1&0\\1&-2.6&1&0&0&0&0&0\\1&1&-2.6&1&0&0&0&0\\0&0&1&-2.6&0&1&0&0\\1&0&0&0&-2.6&0&0&0\\0&0&0&1&0&-2.6&1&0\\1&0&0&0&0&1&-2.6&1\\0&0&0&0&0&0&1&-2.6\end{bmatrix}\begin{bmatrix}u1\\u2\\u3\\u4\\u5\\u6\\u7\\u8\end{bmatrix}=\begin{bmatrix}0\\0\\0\\0\\0\\0\\0\\0\end{bmatrix}$$

$$\Rightarrow \lambda_1=-4.66 \quad \lambda_2=-4.32 \quad \lambda_3=-3.60 \quad \lambda_4=-2.97 \lambda_5=-2.38 \quad \lambda_6=-1.60 \quad \lambda_7=-1.25 \lambda_8=-0.01$$

*The largest eigenvalue is **-0.01** and the eigenvector associated to it:*

$$\begin{bmatrix}0.54\\0.39\\0.46\\0.27\\0.21\\0.24\\0.35\\0.13\end{bmatrix} \Rightarrow$$

A B C D E F G H

Ce = {0.54, 0.39, 0.46, 0.27, 0.21, 0.24, 0.35, 0.13}

Figure 3.17: Calculation of Eigenvector Centrality for each Node

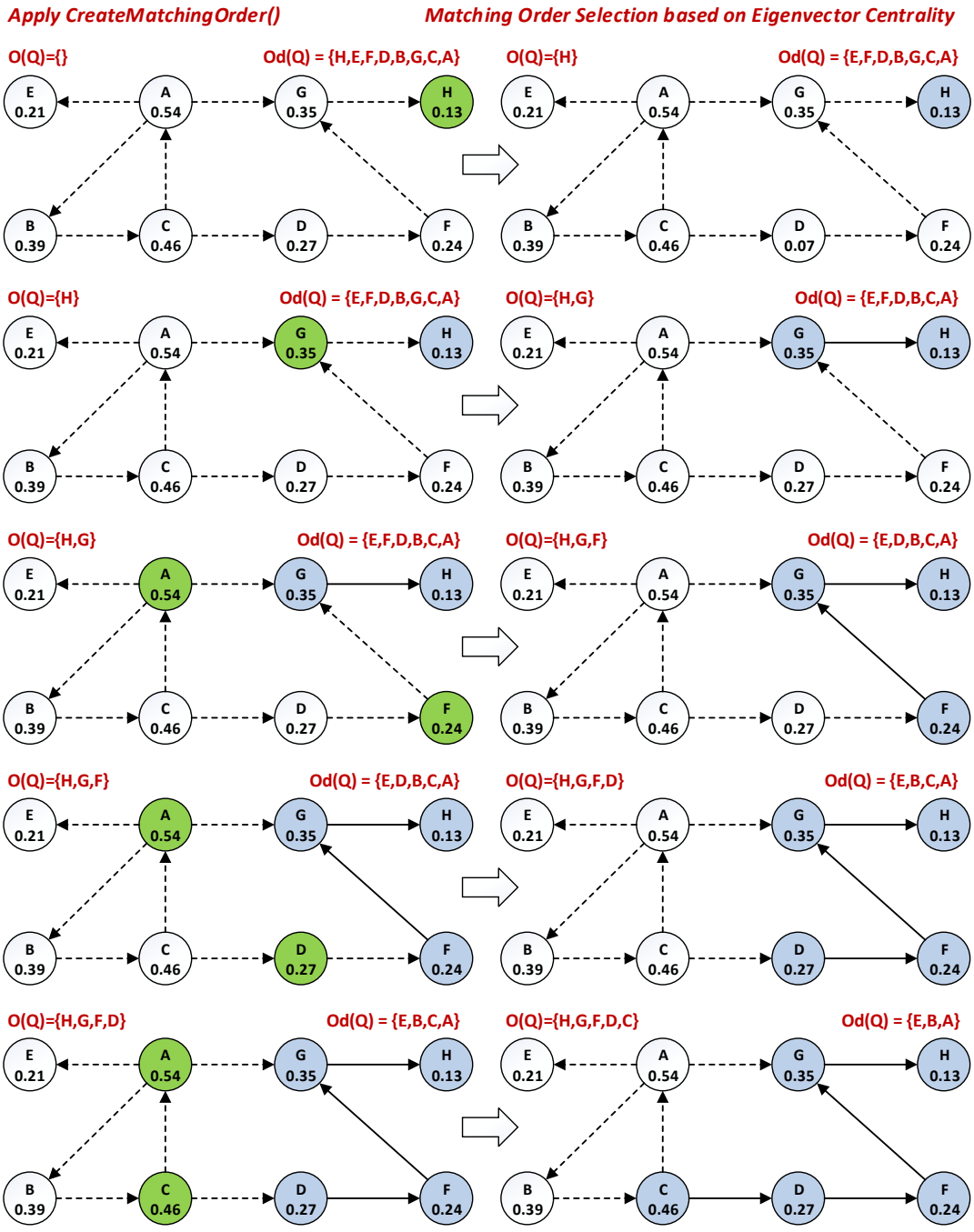

Figure 3.18: An Example of Creating Matching Order with Eigenvector Centrality Method (Part-1)

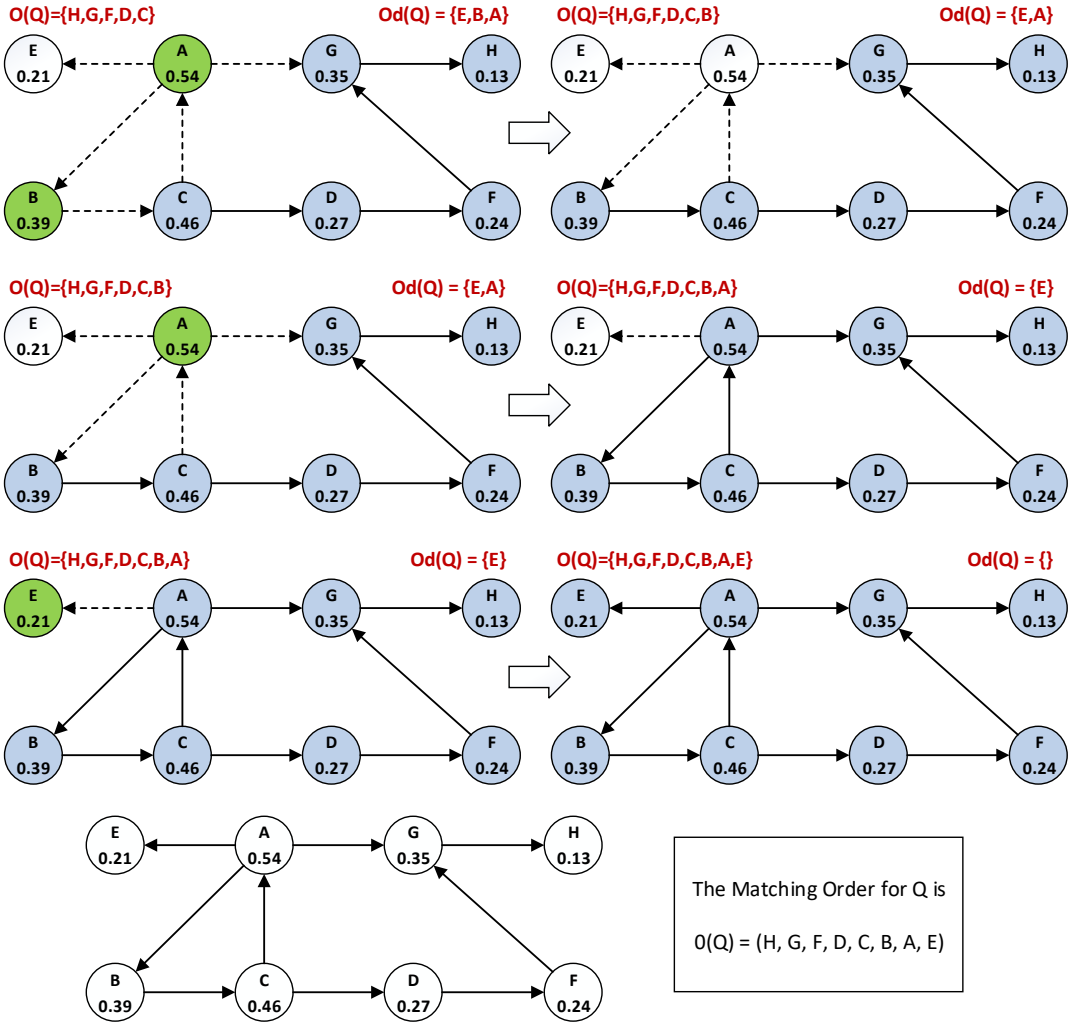

Figure 3.19: An Example of Creating Matching Order with Eigenvector Centrality Method (Part-1)

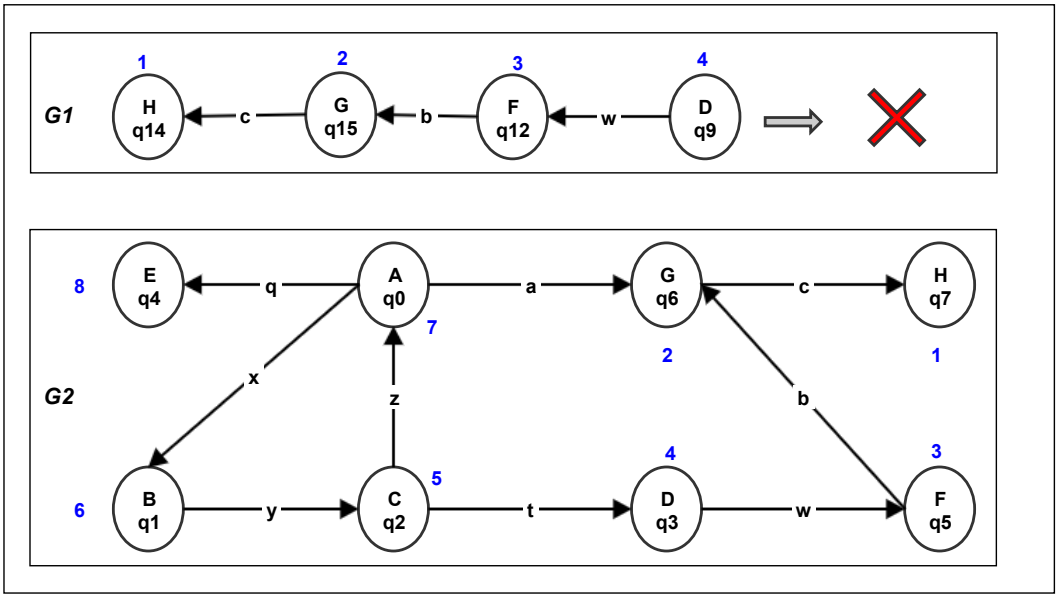

Figure 3.20: An Example of Finding All Exact Matches with the BB-Graph

3.5 Matching Order Selection Based On Hybrid Centrality

Based on the success of graph centrality measures on matching order selection, we decided to create a new matching order selection method that combines all graph centrality measures to improve the performance. The pseudocode of calculating a *Matching Order Selection based on Hybrid Centrality* is given in Algorithm 25.

Algorithm 25 Matching Order Selection Based On Hybrid Centrality

Input: Q : All query nodes, G : All data nodes

Output: $O(Q)$: Query nodes in matching order

1: **procedure** MOSBASEDONHC(Q, G)

2: $O(Q) \leftarrow \emptyset$ ▷ $O(Q)$: Query nodes in matching order

3: $O_h(Q) \leftarrow HybridCentrality(Q)$ ▷ $O_h(Q)$: List of query nodes that assigned hybrid value

4: $O_h(Q) \leftarrow Sort_{asc}(O_h(Q))$ ▷ Sort $O_h(Q)$ with ascending order

5: $O(Q) \leftarrow CreateMatchingOrder(O_h(Q))$

6: **return** $O(Q)$

Algorithm 26 Hybrid Centrality

Input: Q : Query graph

Output: $C_c(Q)$: Closeness Centrality values for nodes of graph Q

1: **procedure** HYBRIDCENTRALITY(Q)
2: $C_h(Q) \leftarrow \emptyset$
3: $O_d(Q) \leftarrow DegreeCentrality(Q)$
4: $O_c(Q) \leftarrow ClosenessCentrality(Q)$
5: $O_b(Q) \leftarrow BetweennessCentrality(Q)$
6: $O_e(Q) \leftarrow EigenvectorCentrality(Q)$
7: **for** i `from 1 to` $|Q|$> **do**
8: $order_d$ = order of i^{th} element of $O_d(Q)$
9: $order_c$ = order of i^{th} element of $O_c(Q)$
10: $order_b$ = order of i^{th} element of $O_b(Q)$
11: $order_e$ = order of i^{th} element of $O_e(Q)$
12: $C_h(v_i) = (order_d + order_c + order_b + order_e)/4$
13: Push $C_c(h_i)$ to $C_h(Q)$
14: **return** $C_h(Q)$

The algorithm firstly applies the *HybridCentrality* method that is given in Algorithm 26 to collect all graph centrality values. The method calculates *degree*, *closeness*, *betweenness* and *eigenvector* centrality values and find orders of query nodes for each centrality value. After finding the centrality values, it get averages of them. After calling *HybridCentrality*, the algorithm sorts each query nodes in ascending order by their values to obtain $O_h(Q)$ and then calls *CreateMatchingOrder* method. The method looks at $O_h(Q)$ and it adds a start query node with the minimum value into matching order list $O(Q)$ and continue to add nodes that it is connected to the already added nodes. If there are nodes with same value, then it selects the one that has neighbors with the minimum value.

The *BB-Plus* approach does not use this centrality because of its performance problem. When we are experimenting on different dataset, we see that each graph centrality measure fits with a spesific size and type of query. Combining them, makes them weaker on these queries. Therefore, we do not prefer to use this centrality measure.

3.6 Matching Order Selection Based On Candidate Node Selection

We use *Matching Order Selection Based On Candidate Node Selection* to create a matching order in order to find exact matches of Q in G efficiently by combining this order with a branch-and-bound algorithm. The pseudocode of the *Matching Order Selection based on Candidate Node Selection* is given in Algorithm 27.

Algorithm 27 Matching Order Selection Based On Candidate Node Selection

Input: Q : All query nodes, G : All data nodes,

Output: M : All embeddings of Q in G

1: **procedure** BBGRAPHWITHCNS(Q, G)
2: $O(Q) \leftarrow \emptyset$ ▷ $O(Q)$: Query nodes in matching order
3: **for all** <$u \in Q$> **do**
4: $C_u \leftarrow \emptyset$ ▷ C_u = Candidate nodes of u
5: $List(C_{eu}) \leftarrow \emptyset$
6: $C_u \leftarrow$ FilterByLabel(u, G)
7: $C_u \leftarrow$ FilterByRelationships(u, C_u, G)
8: **if** u has property **then**
9: $C_u \leftarrow$ FilterByProperty(u, C_u, G)
10: **for all** <$e \in edgesOf(u)$> **do**
11: C_{eu} = createCandidateEntity(label(e), label(u))
12: **for all** <$v \in edgesOf(u)$> **do**
13: **if** $e = v$ **then**
14: $C_{eu}.getSize() + +$
15: Push C_{eu} to $List(C_{eu})$
16: $O_{cns}(Q) \leftarrow SortByCandidateEntity_{asc}(O(Q), List(C_{eu}))$ ▷ Sort $O(Q)$ with ascending order based on C_eu
17: $O(Q) \leftarrow CreateMatchingOrder(O(Q))$
18: $M \leftarrow BBGraph(O(Q), G)$
19: **return** M

In the beginning, the algorithm uses filtering methods of the *BB-Graph* algorithm to obtain candidate node size for each query node. Firslty, it applies *filterByLabel* to narrow the candidate node list down by finding matching labels of both query and data nodes. The pseudo code for the *filterByLabel* can be found in Algorithm 28. Then, the algorithm applies *filterByRelationships* to the candidate node list to eliminate the data nodes which have not the same incoming and outgoing nodes with the query nodes. The pseudo code for the *filterByRelationships* can be found in Algorithm 29. The last step for finding candidate nodes, the algorithm uses *filterByProperty* to eliminate the data nodes that have not same property value with query node's if there is any. The pseudo code for the *filterByProperty* can be found in Algorithm 30.

Algorithm 28 Filter By Label

Input:: u : Query node,
$l(u)$: Label of query node,
C_u : Set of candidate nodes for u

1: **procedure** FILTERBYLABEL(u)
2: **for all** <$v \in G$> **do**
3: **if** $l(u)$ = $l(v)$ **then**
4: Add v into C_u
5: **return** C_u

Algorithm 29 Filter By Relationship

Input: u : Query node, C_u : Candidate set for u constructed by label
Output: C_u^* : Set of candidate nodes for u

1: **procedure** FILTERBYRELATIONSHIP(u, C_u)
2: $C_u^* \leftarrow \emptyset$
3: $L_G \leftarrow$ List of groups G of the adjacent relationships of u based on <$type, direction$>
4: **for all** <$v \in C_u$> **do**
5: **if** foreach $G < type, direction >$ in L_G, v has at least $G < type, direction >$ many number of adjacent relationships of type $type$ and direction $direction$ **then**
6: Add v into C_u^*
7: **return** C_u^*

Algorithm 30 Filter By Property

Input: u : Query node,

C_u : Candidate set for u constructed by label and relationships

Output: C_u^* : Set of candidate nodes for u

1: **procedure** FILTERBYPROPERTY(u, C_u)

2: $C_u^* \leftarrow \emptyset$

3: **for all** <$v \in C_u$> **do**

4: **if** foreach different property p of u, v satisfies the same value conditions as u for p **then**

5: Add v into C_u^*

6: **return** C_u^*

Algorithm 31 Create Candidate Entity

Input: l_e : Label of edge, l_u : Label of node

Output: C_{ue} : Candidate Entity

1: **procedure** CREATECANDIDATEENTITY(l_e, l_u)

2: $C_{eu}.setLabelOfEdge(l_e)$

3: $C_{eu}.setLabelOfNode(l_u)$

4: $C_{eu}.setSize(0)$

5: **return** C_{eu}

Algorithm 32 Sort By Candidate Entity

Input: $O(Q)$: Ordered Query Nodes, $List(C_{eu})$: List of Candidate Entities

Output: $O(Q)$: Ordered Query Nodes

1: **procedure** SORTBYCANDIDATEENTITY($O(Q)$, $List(C_{eu})$)

2: $List(C_{eu}) \leftarrow Sort_{asc}(List(C_{eu}))$

3: **for all** <$C_{eu} \in List(C_{eu})$> **do**

4: **if** !$C_{eu}.getNodeLabel() \in O(Q)$ **then**

5: Push $C_{eu}.getNodeLabel()$ to $O(Q)$

6: **return** $O(Q)$

After the algorithm finding the candidate node list size for query nodes, it creates *Candidate Entities* for each edges of query nodes with the *CreateCandidateEntity* method which can be found in Algorithm 31. It basically gives the total number of edges (that matches query node's labels) of candidates nodes. Each candidate entity consists of an edge label, node label and size. After calculating candidate entities, sort them candidate entity's sizes with the *SortByCandidateEntity* method which can be found in Algorithm 32 to obtain $O_{cns}(Q)$. If a node's candidate entity size is smaller than the others, than it finds itself a better place at the order.

Then, *CreateMatchingOrder* method is called to find a matching order for the query node. The method puts query node with minimum candidate node size first. However, if the candidate node list size is same with another query node, then it looks at their candidate node size of their neighbor. The method puts first the query node that has neighbor with minimum value. Because when it is branching from the data graph, big portion of the data nodes are eliminated at the first steps and it requires less candidate nodes to check in order to obtain all the matchings. Therefore, the algorithm can eliminate redundant candidate nodes and reach to the goal state in less time by reducing the search space.

The *BB-Plus* approach uses an extended version of *Matching Order Selection based on Candidate Node Selection* when the database is not real-time which is called *Matching Order Selection based on Candidate Node Selection Without Candidates* and the pseudocode for the algorithm given in Algorithm 33. The approach calculates candidate node size and store it to database before querying. It calculates the candidate node size based on *filterByLabel*, *filterByRelationships* and *filterByProperty*. While querying it uses that candidate node size information and calls *Matching Order Selection based on Candidate Node Selection Without Candidates* to create a matching order. This algorithm calculates matching order in less time in compare to *Matching Order Selection based on Candidate Node Selection*. Because, it uses already calculated candidates nodes and does not lost time for calculating them. After finding the matching order, the approach calls *BB-Graph* algorithm to find the exact matches efficiently. The *BB-Graph* algorithm starts with the node that has the first place in the order and continues to branch from it based on the order until finding all the exact matches for the query graph in the data graph.

Algorithm 33 Matching Order Selection based on Candidate Node Selection Without Candidates

Input: Q : All query nodes

$C(Q)$: Candidate node size list of Q

Output: $O(Q)$: Query nodes in matching order

1: **procedure** MOSBASEDONCNSWITHOUTCANDIDATES(Q, $C(Q)$)
2: $O(Q) \leftarrow \emptyset$ ▷ $O(Q)$: Query nodes in matching order
3: **for all** $<u \in Q>$ **do**
4: C_u = Candidate node size of u from $C(Q)$
5: Push C_u and u to $O(Q)$
6: $O_{cns}(Q) \leftarrow Sort_{asc}(O(Q))$ ▷ Sort $O_{cns}(Q)$ with ascending order based on candidate node size
7: $O(Q) \leftarrow CreateMatchingOrder(O(Q))$
8: **return** $O(Q)$

In the example, the *Matching Order Selection based on Candidate Node Selection* firstly applies *FilterByLabel* method as shown in Table 3.5 to get a candidate node list. According to the table, query node A is matching with $q0$ and $q17$, B is matching with $q1$ and $q16$, C is matching with $q2$, $q10$ and $q18$, D is matching with $q3$, $q9$ and $q19$, E is matching with $q4$, F is matching with $q5$, $q12$ and $q20$, G is matching with $q6$, $q8$, $q11$, $q13$, $q15$ and H is matching with $q7$, $q14$ in the data graph.

After applying the *FilterByLabel* method, it calls the *FilterByRelationships* method to apply on the candidate node list as shown in Table 3.6. According to the table, for the query node A, $q17$ is eliminated because it does not have a relationship with a node that have label E. For the query node B, the candidate node list remains the same because all of its relationships exits in data nodes $q1$ and $q16$. For the query node C, $q18$ is eliminated because it does not have a relationship with a node that have label B. $q10$ is also eliminated because it does not have a relationship with nodes that have label A and B. For the query node D, the candidate node list remains the same because all of its relationships exits in data nodes $q3$, $q9$ and $q19$. For the query node D, the candidate node list also remains the same because all of its relationships exits in data node $q4$. For the query node F, $q20$ is eliminated because it does not have a

Table 3.5: FilterByLabel method applies to Q

Nodes	**Candidate Node List (Nodes with same label)**
A	q0, q17
B	q1, q16
C	q2, q10, q18
D	q3, q9, q19
E	q4
F	q5, q12, q20
G	q6, q8, q11, q13, q15
H	q7, q14

relationship with a node that have label G. For the query node G, $q8$ is eliminated because it does not have a relationship with a node that have label A. $q11$ is also eliminated because it does not have a relationship with a node that have label F. $q13$ is also eliminated because it does not have a relationship with a node that have label A. Lastly for the query node H, the candidate node list also remains the same because all of its relationships exits for both data node $q7$ and $q14$. Finally, the algorithm looks that if is there any property on any query node. Because of lack of properties, the algorithm does not apply *FilterByProperty* method.

Table 3.6: FilterByRelationship method applies to Q

Nodes	**Incoming Relationships**	**Outgoing Relationships**	**Candidate Node List**
A	C->A	A->B, A->E, A->G	q0
B	A->B	B->C	q1, q16
C	B->C	C->D	q2
D	C->D	D->F	q3, q9, q19
E	A->E	-	q4
F	D->F	F->G	q5, q12
G	A->G, F->G	G->H	q6, q15
H	G->H	-	q7, q14

After all the filtering methods, the candidates nodes size for each query node is defined as $(2, 2, 1, 3, 1, 1, 2, 2)$ for (A, B, C, D, E, F, G, H). The algorithm sorts them in ascending order to obtain $O_{cns}(Q)$ = (C, E, F, A, B, G, H, D) and calls ***CreateMatchingOrder***. Therefore, it obtains $O(Q)$ = (C, A, E, D, G, F, D, H). All the steps for finding a matching order with the *Matching Order Selection based on Candidate Node Selection* is given in Figure 3.21, 3.22.

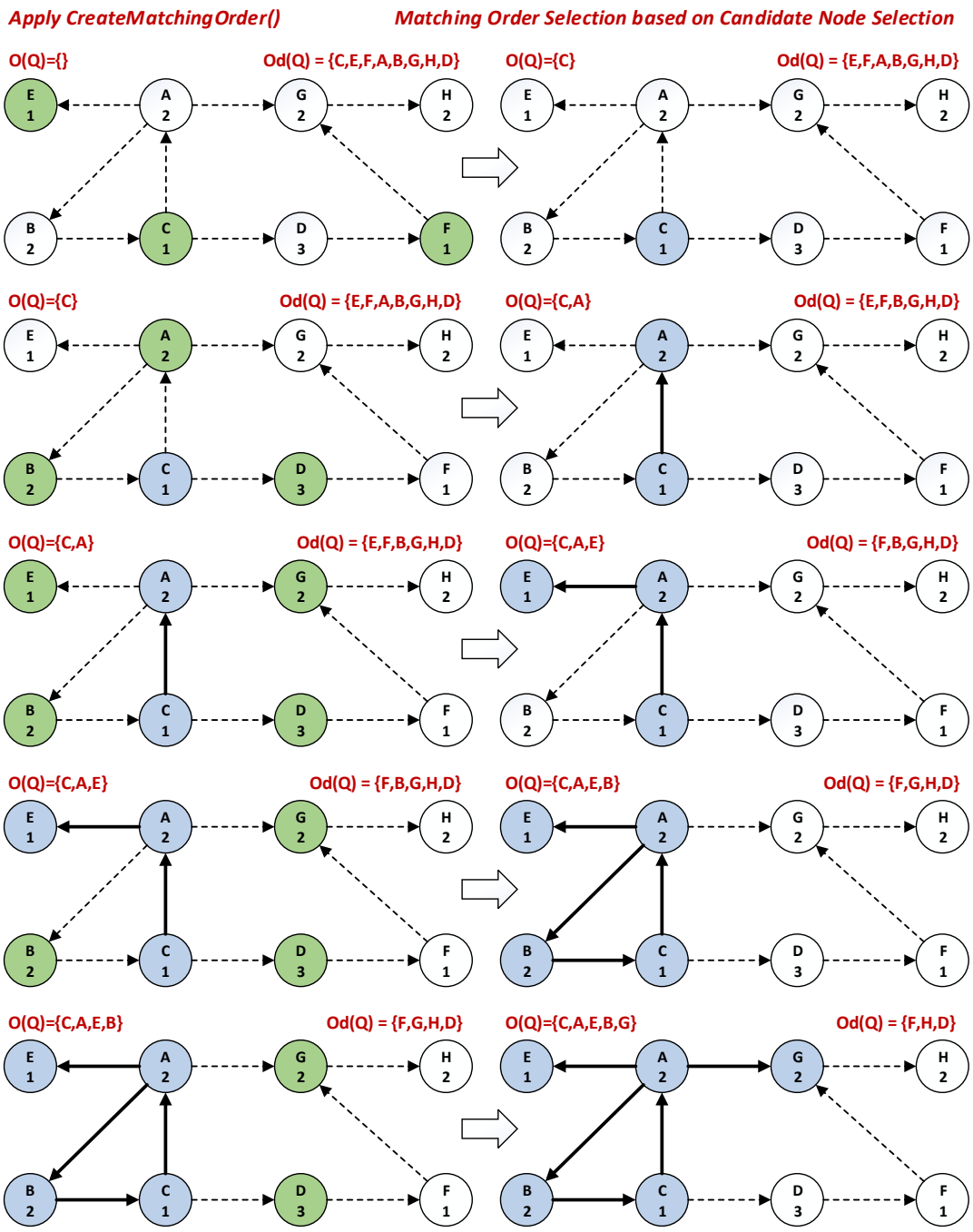

Figure 3.21: An Example of with the MosBasedOnCNS (Part-1)

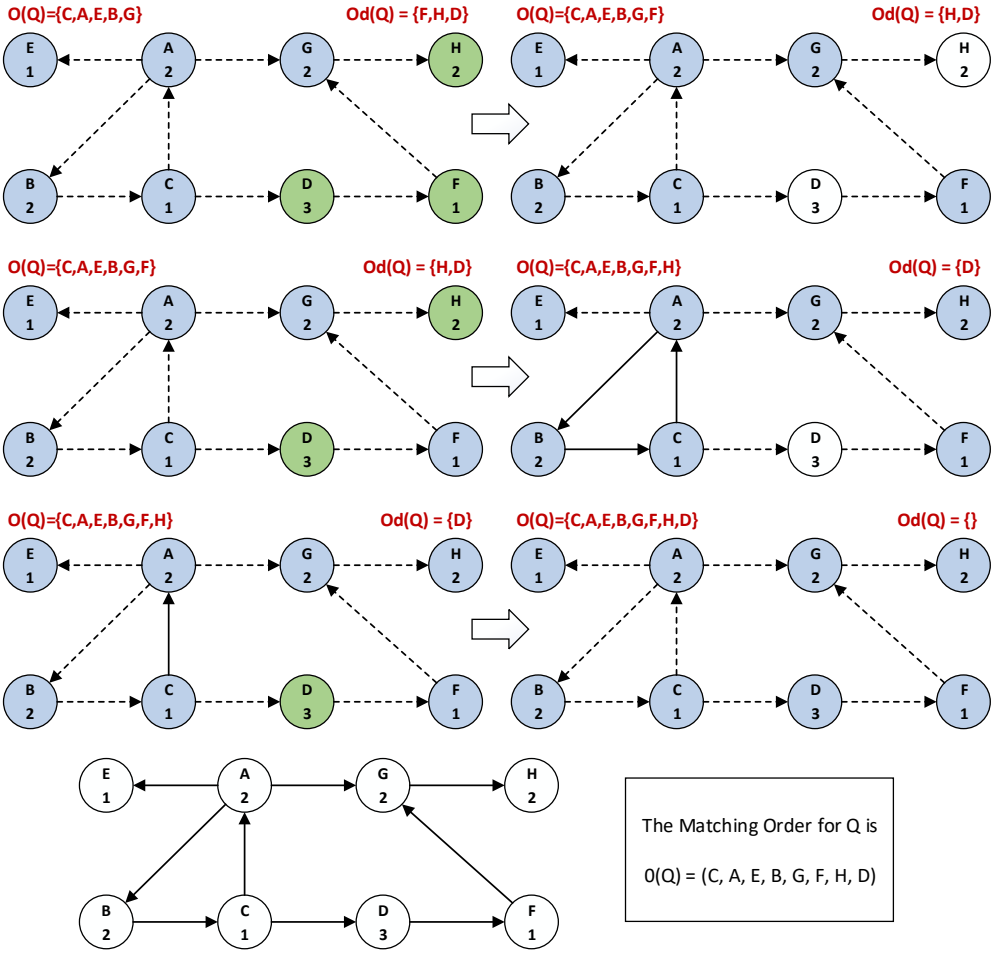

Figure 3.22: An Example of with the MosBasedOnCNS (Part-2)

After finding the matching order, the *BB-Plus* approach calls the *BB-Graph* algorithm with the ordered query graph $O(Q)$ and the data graph G. As shown in Figure 3.23, the algorithm branches for only graph $G1$ that contains all the exact matches for Q.

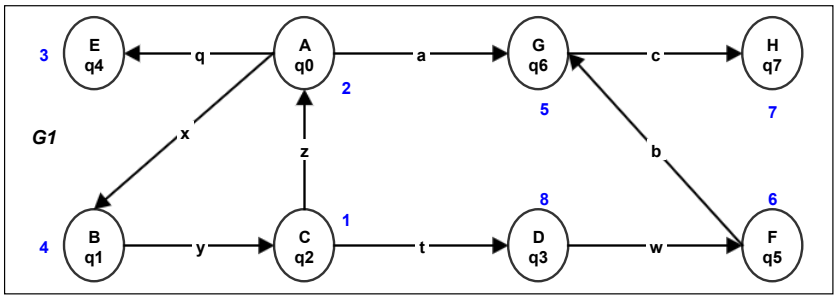

Figure 3.23: An Example of Finding All Exact Matches with the MosBasedOnCNS

3.7 Comparison of Matching Order Selection Methods

In this chapter, all matching order selection methods are compared to each other based on how they are created and how they act in different types of queries in different types of databases.

3.7.1 Based on Their Creation Methods

All the matching order selection methods are distinguished from each other with their way of creating their order. However, they can be divided into two groups according to whether they use fundamental graph centrality measures or not.

The first group consists of the *Matching Order Selection Based On Degree Centrality*, *Matching Order Selection Based On Closeness Centrality*, *Matching Order Selection Based On Betweenness Centrality*, *Matching Order Selection Based On Eigenvector Centrality*. All of them uses fundamental graph centrality measures on query graph to find a matching order and does not consider the data graph. Their only concern is detecting the most important nodes in query graph and then finding candidate nodes in data graph based on the order that determined with importance of query nodes. Their time of calculating matching order is shorter than the other group. Therefore if they create a great matching order, they will beat the other groups. Because they do not spend too much time create matching order and spends all time to finding exact matches. However if they create a bad matching order because of not considering the data graph, finding the exact matches in the data graph can take too much time and the algorithms will lose their effectiveness.

The second group consists of *Matching Order Selection Based On Candidate Node Selection* that does not use graph centrality measures. However, it work on both the query and data graph by finding candidate nodeon the data graph for the query graph. Because it touches the data graph, it can find a better matching order and get more efficient results. However, the time of calculating matching order takes much time in compare to the first group. Therefore, they can not compete with the ones in the first group because they spend extra time for gathering candidate node to create a matching order. On the other hand, if these candidate sizes are predefined in the database or we

do not consider the time for creating a matching order, it can be seen that they give quicker results than others.

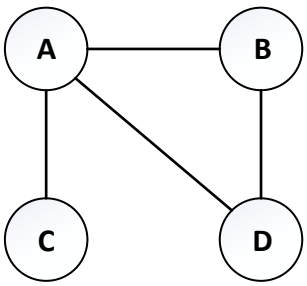

Figure 3.24: Example Query Graph for Comparison of the Improved BB-Graph Algorithms based on Matching Order

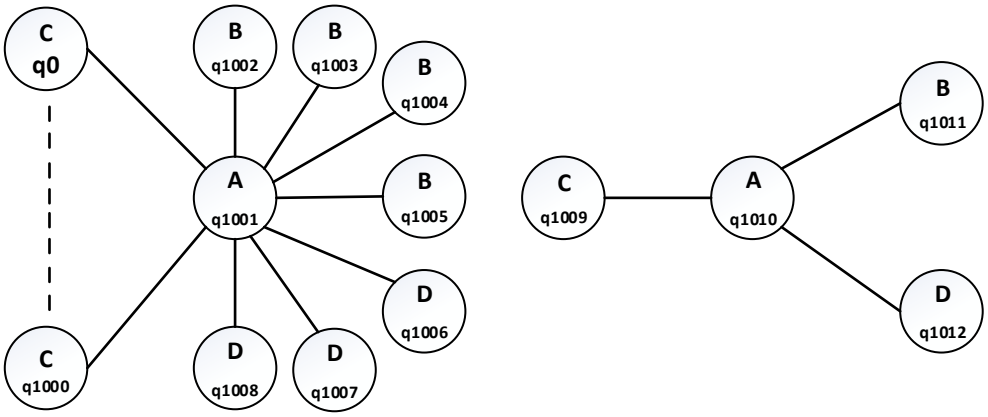

Figure 3.25: Example Data Graph 1 for Comparison of the Improved BB-Graph Algorithms based on Matching Order

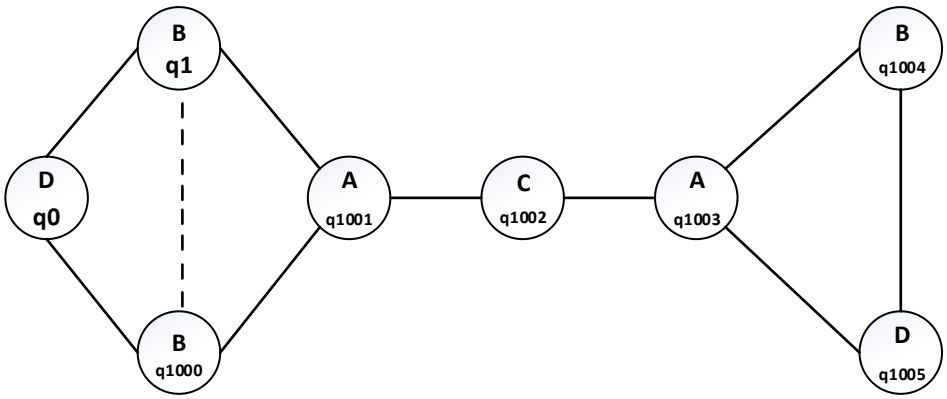

Figure 3.26: Example Data Graph 2 for Comparison of the Improved BB-Graph Algorithms based on Matching Order

Let's explain the effect of creating matching order with an example using example query graph as shown in Figure 3.24 and data graphs as shown in Figure 3.25, 3.26 and take the *Matching Order Selection Based On Degree Centrality* from the first group and the *Matching Order Selection Based On Candidate Node Selection* from the second group. There is only one exact matches at the both data graphs.

The *Matching Order Selection Based On Degree Centrality* creates a matching order $C - A - B - D$ using query graph. This matching order is good for the example second data graph, because it only takes 2 attemps to find the exact match. However, for the example first data graph, 1002 attempts are required to find it.

On the other hand, the *Matching Order Selection Based On Candidate Node Selection* adapts the matching order based on the data graph. It looks at the candidate node size for the query graph and finds the matching order $A-D-B-C$ for the first data graph and $C - A - D - B$ for the second graph. Only 4 and 2 attemps are required for first data graph and second graph respectively. Therefore the methods of second group are good at finding a good matching order to find exact matches efficiently. However, they can get behind the algorithms of first group like at the second graph. Because both algorithms find the matching order but it takes more time find a matching order for the algorithms of second group.

3.7.2 Based on the Type of Queries

The algorithms of first group can affected by the query types. They can give greater performance at specific query types. For example, degree centrality gives greater results than the others at path queries. Degree centrality also give greater results at cyclic queries as betweenness centrality. On the other hand, closeness and eigenvector centrality give greater results at other types of queries.

The algorithms of second group also does not affected by the query types. Because, they adapt themselves to changing data graphs and they can act with the best performance for the each query types as we can see from Figure 3.24, 3.25 and 3.26.

3.7.3 Based on the Volatility of Databases

Matching order selection methods of first group does not affected by the volatility of database because they do not consider data graph while they are calculating. Therefore, they are great when working on real-time databases.

On the other hand, matching order selection methods of second group consider both query and data graph while they are calculating. Therefore, they spend a lot of time for creating a matching order as we mentioned earlier. The time for creating matching order needs to be eliminated by calculating candidate nodes or edges before querying. If the database is real-time and changes very fast, then we cannot calculate candidate nodes or edges before querying and the total process time querying can be increased.

3.8 Determining Matching Order Selection Methods

As we mentioned earlier, the approach uses rules to determine the best matching order for queries that executed on any real-time databases. Graph centrality based matching order selection methods are used in real-time databases. However, there is no exact rules for which matching order selection method performs best at which query. Therefore, we used machine learning to create rules for determining best matching order selection method.

At first, we execute queries on three different databases which are the WorldCup, Pokec and Population dataset. We execute 100 queries on the WorldCup, 50 queries on the Pokec and 50 queries on the Population dataset for the training data set. Also we execute 20 queries on the WorldCup, 10 queries on the Pokec and 10 queries on the Population dataset for the test data set.

The information about training and test dataset for determining matching order selection can be found in Figure 3.7 and the dataset can be found Appendix A. In both training and test dataset, we generate different queries with different inputs that has executed on these three dataset. As shown in Figure 2.9, eight attributes used in the dataset which are *data graph node size*, *data graph edge size*, *query graph node size*, *query graph edge size*, *number of distinct query node label*, *number of distinct query edge label*, *number of query node with properties* and *query type*. We recored the best matching order selection method for each query.

Table 3.7: Information about Determining Matching Order Dataset

Training Data Size	**Test Data Size**	**♯ of Attributes**	**Output Values**
190	39	8	Degree Centrality Betweenness Centrality Closeness Centrality Eigenvector Centrality BB-Graph

We used Weka Software for creating decision tree based on the training and test dataset. We applied J48 to create the decision tree as shown in Figure 3.1 and get results for criticize its performance.

Based on the decision tree, if the query type is "path query", *degree centrality* gives the best results. On the other hand, if the query type is "cyclic query", the tree branches and look for edge sizes. If the edge size is bigger than four, then it selects *degree centrality* again. However, if the edge size is equals or smaller than four, then it selects *betweenness centrality*. Finally, if the query type is "others", then it branches based on node size. If the node size is bigger than five, then it selects *eigenvector centrality*. If not, then it selects *closeness centrality*.

Table 3.8: Attributes of Determining Matching Order Dataset

Attributes	**Type**
Data Graph Node Size	Small, Medium, Big
Data Graph Edge Size	Small, Medium, Big
Query Graph Node Size	Numeric
Query Graph Edge Size	Numeric
Number of Distinct Node Label	Numeric
Number of Distinct Edge Label	Numeric
Number of Nodes with Properties	Numeric
Query Type	Path, Cyclic, Others

According to the accuracy values as shown in Table 3.9, the decision tree correctly classifies the matching order selection methods with 79.48% percentage. In addition to the accuracy values, the confusion matrix for each matching order selection methods are given in Table 3.10.

Table 3.9: Detailed Accuracy for each Matching Order Selection Method

	Precision	**Recall**	**F-Measure**
Degree Centrality	0.81	0.94	0.87
Betweenness Centrality	0.83	0.71	0.77
Closeness Centrality	0.75	0.67	0.70
Eigenvector Centrality	0.75	0.60	0.67
Average	0.79	0.80	0.79

Table 3.10: Confusion Matrix for each Matching Order Selection Method

	Degree Centrality	**Betweenness Centrality**	**Closeness Centrality**	**Eigenvector Centrality**
Degree Centrality	17	1	0	0
Betweenness Centrality	2	5	0	0
Closeness Centrality	2	0	6	1
Eigenvector Centrality	0	0	2	3

CHAPTER 4

EXPERIMENTS AND RESULTS

In this section, the BB-Graph, Neo4j's Cypher, DualIso, GraphQl, TurboIso, VF3 and our approach BB-Plus are compared based on their time complexity. The algorithms were executed with different type of queries in different graph databases that created with importing WorldCup, Pokec and Population dataset via Neo4j GDMBS. Each dataset were considered as both real-time and historical when running experiments.

4.1 The Dataset

The WorldCup is a publicly available dataset that contains information about matches, players, squads, countries etc. and their relationships for all World Cup Tournaments from 1930 to present day [37]. The Data Model for WorldCup dataset can be seen in Figure 4.1.

The Pokec dataset is publicly available and it consists of data from Pokec which is one of the biggest social network platform in Slovakia. The dataset contains anonymized data from Pokec network, which is about the user's profile like gender, age, hobbies, interests, education, likes. Additionaly, it contains friendships relation between all the users [38]. The Data Model for Pokec dataset can be seen in Figure 4.2.

Our last dataset, the Population dataset, is not publicly available but it has very large and complex data to show the performance of all algorithms. The population database is owned by Kale Yazılım. It contains anonymized population data of Turkey. It includes personal information of people like gender, address etc. and the relationships like mother, father, spouse between these people.

Graph databases are good for all the datasets because they are both large, complex and highly connected. Especially, Population dataset needs expensive join operations to execute queries. Therefore, all the dataset is imported to Neo4j database with using Java API. Statistics of the WorldCup, Pokec and Population graph databases can be seen in Table 4.1.

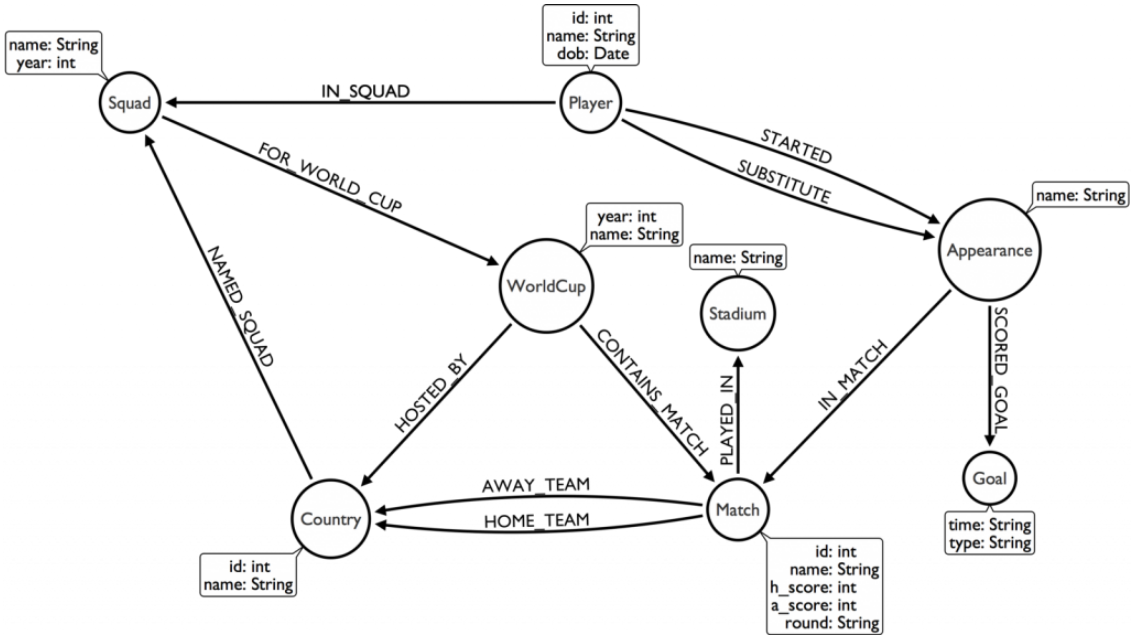

Figure 4.1: The Data Model for WorldCup dataset

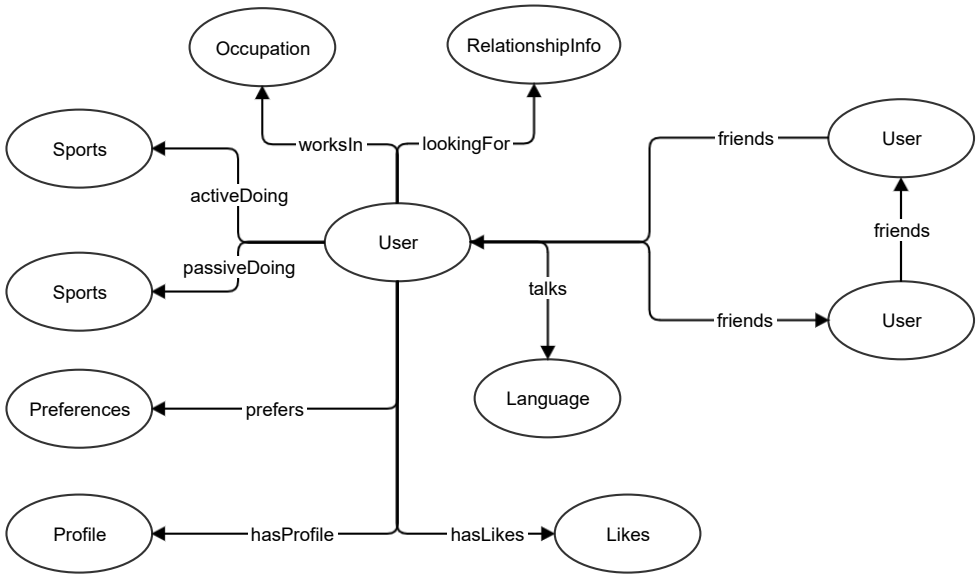

Figure 4.2: The Data Model for Pokec dataset

Table 4.1: Statistics of the WorldCup, Pokec and Population Graph Databases

	WorldCupDB	PokecDB	PopulationDB
Size	112 MB	4.93GB	19.6 GB
# of nodes	45348	1632803	70422787
# of relationships	86577	30622564	77163109
# of distinct node labels	12	8	14
# of distinct relationship types	17	10	18
Average of # of labels per node	1	1	1

4.2 The System Configuration

All experiments were performed on a machine with Intel Core Quad Core 2,70 GHz i7-6820HQ CPU, 32 GB DDR4 RAM and running Windows 10 operation system. On the other hand, all the algorithms were implemented in Java programming language on IntelliJ IDEA and using graph data structures of Neo4j GDBMS v3.4.9 via embedded Java API. The system architecture can be seen in Figure 4.3.

4.3 Queries

In the experiments, different types of queries were executed on the databases. We tried to use different nodes and relationships in each query to show the performance of algorithms under different cases. The results were collected for 5 real-world queries for the Population database, 4 real-world queries for the Pokec database and 5 real-world queries for the WorldCup.

The queries for all the algorithms were given in BFS format like in the Asiler's paper which was explained in Section 2.4.10, except Cypher's queries. The average execution time was used in the experiments to algorithm's performance, which was calculated with repeating each queries 10 times, except the slow queries which was repeated only one time.

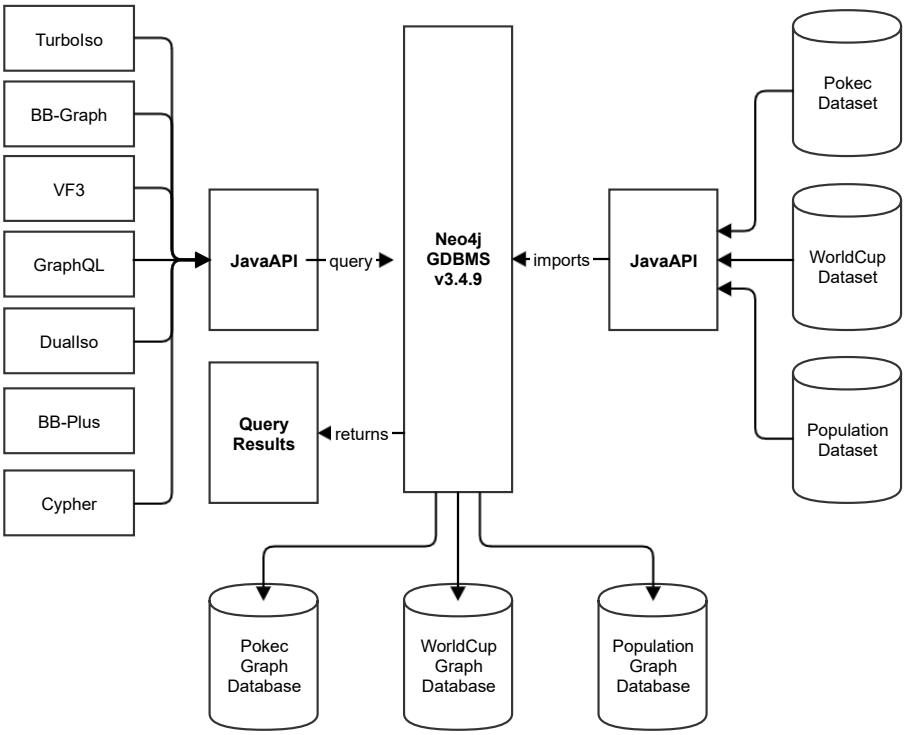

Figure 4.3: The System Architecture

4.4 Experiments on the Databases

In this section, queries are executed on the WorldCup, Pokec and Population databases to evaluate the performance of all the algorithms. Especially, we repeated the same queries in the Asiler's paper [1] in WorldCup and Population databases to show the improvement in time complexity. We show the performance of BB-Plus based on historical and real-time databases. In addition, we show the performance of historical BB-Plus with or without the consideration of degree factor in CNS to show the difference in time efficiency.

4.4.1 Experiments on the WorldCup Database

In this section, we compare the algorithms using the WorldCup database. We execute 5 different queries to evaluate the performance of each algorithm. All queries and their BB-Graph representation can be seen in Table 4.2 and all the results are going to discussed in this chapter.

Table 4.2: The queries and their BB-Graph representation on WorldCup database [1]

Queries	BB-Graph Represantation
Players who join squad of different countries	(0,1,Country,NAMED_SQUAD,OUTGOING,Squad)(1,2,Squad,IN_SQUAD,INCOMING,Player) (2,3,Player,IN_SQUAD,OUTGOING,Squad)(3,4,Squad,NAMED_SQUAD,INCOMING,Country)
Players who take role as both substitute and active (STARTED) in the same match	(0,1,Player,STARTED,OUTGOING,Performance)(0,2,Player,SUBSTITUTE,OUTGOING,Performance) (1,3,Performance,IN_MATCH,OUTGOING,Match)(2,3,Performance,IN_MATCH,OUTGOING,Match)
Matches between the same countries occurred in different world cups	(0,1,Time,PLAYED_AT_TIME,INCOMING,Match)(0,2,Time,PLAYED_AT_TIME,INCOMING,Match) (1,3,Match,PLAYED_IN,INCOMING,Country)(1,4,Match,PLAYED_IN,INCOMING,Country) (1,5,Match,CONTAINS_MATCH,INCOMING,WorldCup)(2,3,Match,PLAYED_IN,INCOMING,Country) (2,4,Match,PLAYED_IN,INCOMING,Country)(2,6,Match,CONTAINS_MATCH,INCOMING,WorldCup)
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	(0,1,WorldCup,CONTAINS_MATCH,OUTGOING,Match)(0,2,WorldCup,CONTAINS_MATCH,OUTGOING,Match) (1,3,Match,IN_MATCH,INCOMING,Performance)(1,4,Match,HOME_TEAM,OUTGOING,Country) (1,5,Match,AWAY_TEAM,OUTGOING,Country)(2,4,Match,AWAY_TEAM,OUTGOING,Country) (2,5,Match,HOME_TEAM,OUTGOING,Country)(2,6,Match,IN_MATCH,INCOMING,Performance) (3,7,Performance,SCORED_GOAL,OUTGOING,Goal)(3,8,Performance,STARTED,INCOMING,Player) (6,8,Performance,STARTED,INCOMING,Player)(6,9,Performance,SCORED_GOAL,OUTGOING,Goal)
Players who take role in any match at least 3 world cups	(0,1,Player,STARTED,OUTGOING,Performance)(0,2,Player,STARTED,OUTGOING,Performance) (0,3,Player,STARTED,OUTGOING,Performance)(1,4,Performance,IN_MATCH,OUTGOING,Match) (2,5,Performance,IN_MATCH,OUTGOING,Match)(3,6,Performance,IN_MATCH,OUTGOING,Match) (4,7,Match,CONTAINS_MATCH,INCOMING,WorldCup)(5,8,Match,CONTAINS_MATCH,INCOMING,WorldCup) (6,9,Match,CONTAINS_MATCH,INCOMING,WorldCup)

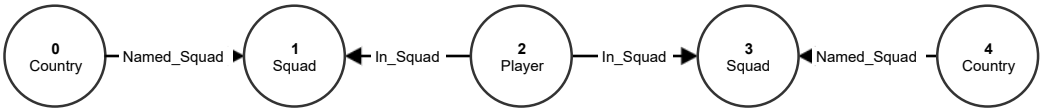

Figure 4.4: The Query1 for WorldCup dataset

In the first query, we are looking for *"Player who join squad of different countries"* in the WorldCup database. The query is an example to path queries and it consists of 5 nodes and 4 edges like shown in Figure 4.4. The results for the query is shown in Table 4.3, 4.4. According to the results, *the BB-Plus* approach gives the best result among all the algorithms on both historical and real-time WorldCup database.

The BB-Plus applies *Matching Order Selection Based On Candidate Node Selection Without Candidates* by using the candidate node size that is already calculated and stored in the database. It finds the best result with $(0-4-1-3-2)$ order on historical WorldCup database as shown in Table 4.4.

On the other hand, the approach applies *Matching Order Selection Based On Degree Centrality*, because the query can be described as path query. It finds best results with $(0-4-1-2-3)$ order on real-time WorldCup database as shown in Table 4.4.

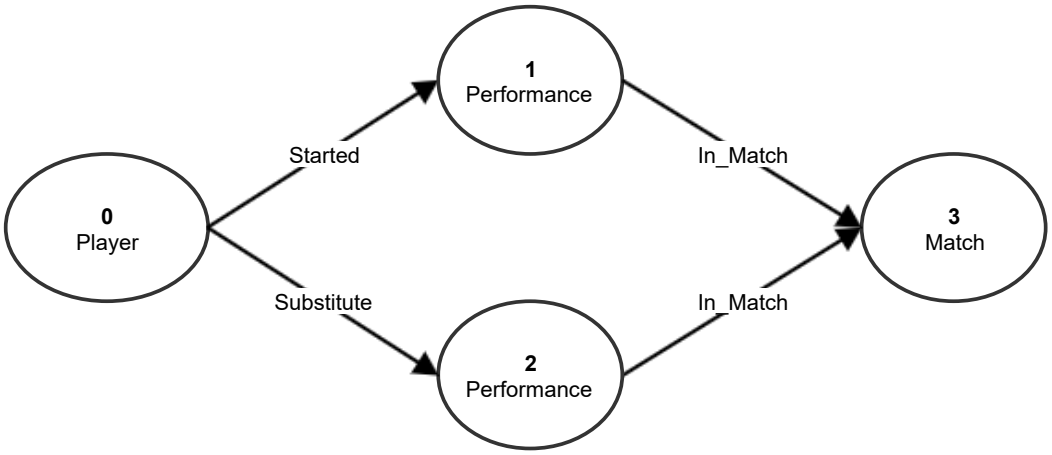

Figure 4.5: The Query2 for WorldCup dataset

In the second query, we are looking for *"Players who take role as both substitute and active (STARTED) in the same match"*. The query is an example to cyclic queries and it consists of 4 nodes and 4 edges like shown in Figure 4.5. The results for the query is shown in Table 4.3, 4.4. According to the results, *the BB-Plus* approach gives the best result among all the algorithms on both historical and real-time WorldCup database.

The BB-Plus applies *Matching Order Selection Based On Candidate Node Selection Without Candidates* by using the candidate node size that is already calculated and stored in the database. It finds the best result with $(3-0-1-2)$ order on historical WorldCup database as shown in Table 4.4.

On the other hand, the approach applies *Matching Order Selection Based On Betweenness Centrality*, because the query can be described as cyclic query and the query edge size is smaller than four. It finds best results with $(3-0-1-2)$ order on real-time WorldCup database as shown in Table 4.4.

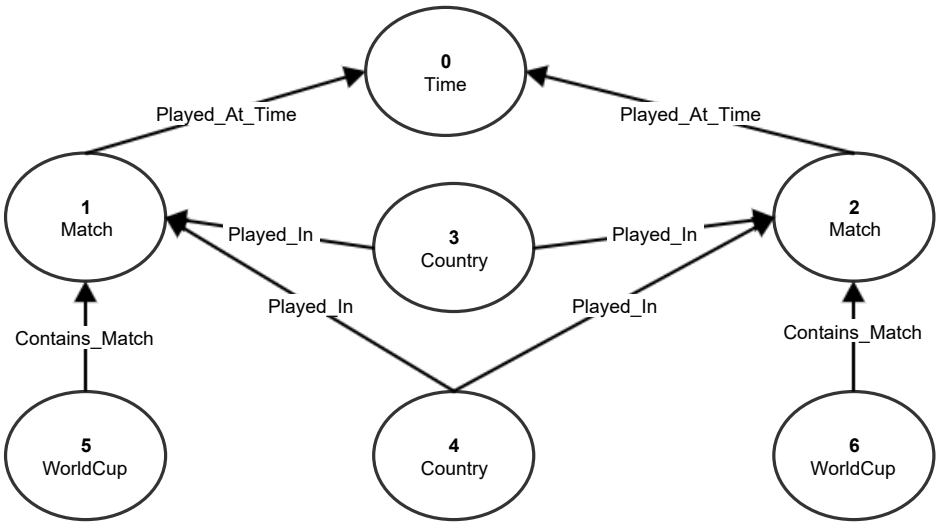

Figure 4.6: The Query3 for WorldCup dataset

In the third query, we are looking for *"Matches between the same countries occured in different worldcups"*. The query consists of 7 nodes, 8 edges and 2 cycles like shown in Figure 4.6. The results for the query is shown in Table 4.3, 4.4. According to the results, *the BB-Plus* approach gives the best result among all the algorithms on both historical and real-time WorldCup database again.

The BB-Plus applies *Matching Order Selection Based On Candidate Node Selection Without Candidates* on the historical WorldCup database and it finds the best result with $(5-6-0-3-4-1-2)$ order as shown in Table 4.4.

On the other hand, the approach applies *Matching Order Selection Based On Degree Centrality* on real-time WorldCup database. Because the query can be described as cyclic query and the query edge size is bigger than four. It beats other algorithms with $(5-6-0-3-4-1-2)$ order as shown in Table 4.4.

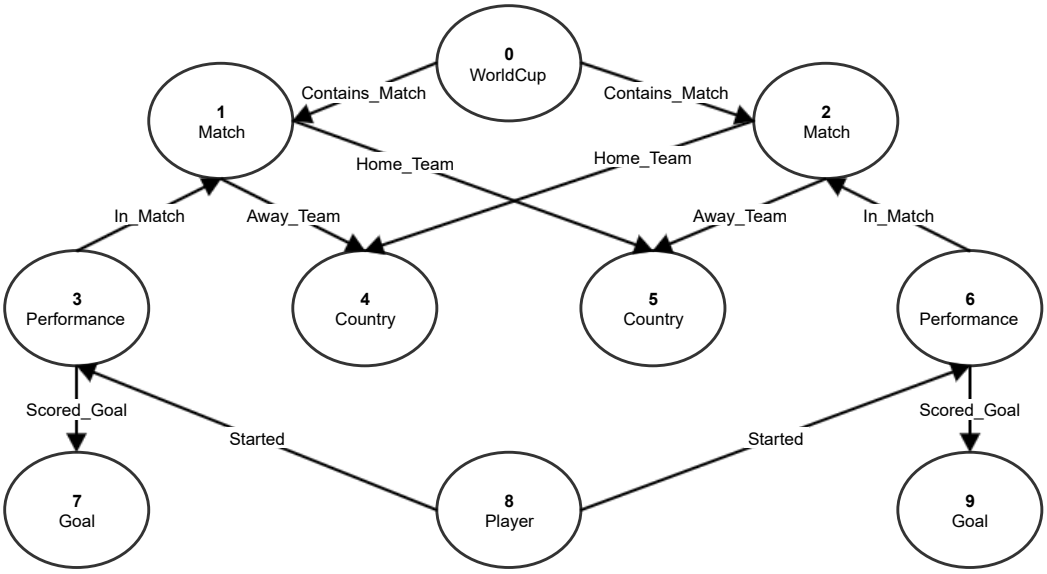

Figure 4.7: The Query4 for WorldCup dataset

In the forth query, we are looking for *"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"* in the WorldCup database. The query consists of 10 nodes, 12 edges and 4 cycles like shown in Figure 4.7. The results for the query is shown in Table 4.3, 4.4. According to the results, *the BB-Plus* approach again beats all the other algorithms on both historical and real-time WorldCup database again.

In the historical database, *The BB-Plus* applies *Matching Order Selection Based On Candidate Node Selection Without Candidates* and finds the best result with $(0-4-5-1-2-7-9-8-3-6)$ order as shown in Table 4.4.

However in the real-time database, it applies *Matching Order Selection Based On Degree Centrality* based on cyclic query type and with twelve edges and it gets the best results with $(7-9-0-4-5--3-6-1-2)$ order.

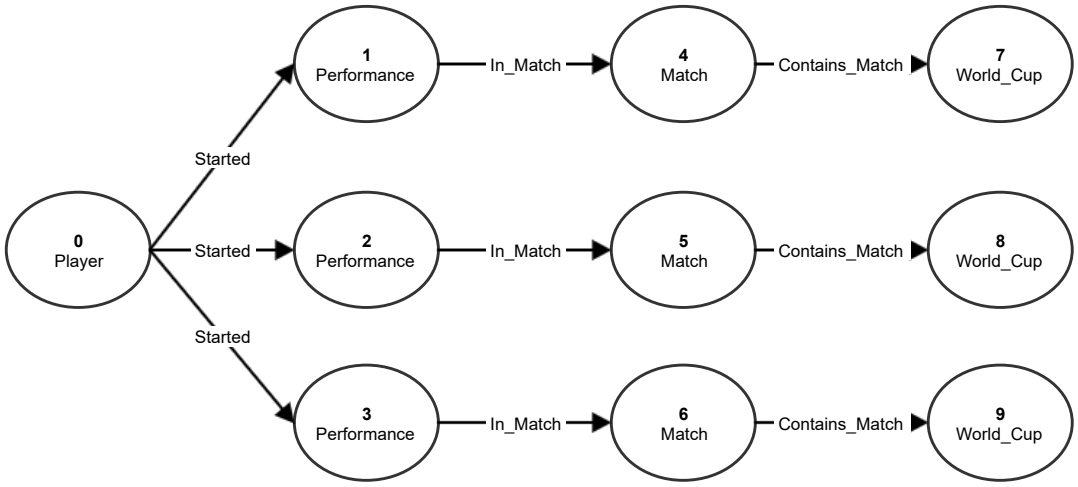

Figure 4.8: The Query5 for WorldCup dataset

In the final query, we are looking for *"Players who take role in any match at least 3 worldcups"*. The query consists of 10 nodes and 9 edges like shown in Figure 4.8. The results for the query is shown in Table 4.3, 4.4. *the BB-Plus* approach is the best among all the algorithm on both real-time and historical WorldCup database again when we look at the results.

The *BB-Plus* approach uses $7-8-9-4-5-6-0-1-2-3$ order based on *Matching Order Selection Based On Candidate Node Selection Without Candidates* and beats all algorithms on historical WorldCup database as shown in Table 4.4.

On the other hand, it applies *Matching Order Selection Based On Eigenvector Centrality* on real-time database. Because, the query type can be described as *"others"* and the query node size is bigger than five. Therefore, it gets the best result with $(5-6-0-3-4-1-2)$ order in real-time WorldCup database as shown in Table 4.4.

Table 4.3: The query results for the WorldCup Database

Algorithm / Query (ms)	Query-1	Query-2	Query-3	Query-4	Query-5
Cypher	1052	10123	7851	26711	2M >
DualIso	10453	34675	35763	26311	2M >
GraphQL	24747	2M >	16082	10654	2M >
TurboIso	5116	6361	-	22072	113714
VF3	6337	2M >	-	78543	2M >
BB-Graph	1050	3353	6500	11193	23754
BB-Plus (Historical) (CNS without Degree Cons.)	602	3231	3092	10850	18121
BB-Plus (Historical)	**574**	**3120**	**2990**	**10672**	**18033**
BB-Plus (Real-Time)	**874**	**3124**	**3180**	**3939**	**18126**

Table 4.4: Matching Order of Different Methods for the WorldCup Database

Algorithm / Query (ms)	Query-1	Query-2	Query-3	Query-4	Query-5
Degree Centrality	0-4-1-2-3	0-1-2-3	5-6-0-3-4-1-2	7-9-0-4-5-8-3-6-1-2	7-8-9-1-2-3-4-5-6-0
Closeness Centrality	1-3-2-0-4	3-0-1-2	0-3-4-5-6-1-2	4-5-7-9-8-0-3-6-1-2	4-5-6-0-1-2-3-7-8-9
Betweenness Centrality	0-1-2-3-4	0-3-1-2	0-3-4-5-6-1-2	0-4-5-7-8-9-3-6-1-2	0-4-5-6-7-8-9-1-2-3
Eigenvector Centrality	0-4-1-3-2	0-1-2-3	5-6-1-2-0-3-4	9-7-8-6-3-0-4-5-1-2	7-8-9-4-5-6-1-2-3-0
Candidate Node Selection	0-4-1-3-2	3-0-1-2	5-6-0-3-4-1-2	0-4-5-1-2-7-9-8-3-6	7-8-9-4-5-6-0-1-2-3

Table 4.5: The process time for calculating matching order selection on WorldCupDB

Algorithm / Query (ms)	Query-1	Query-2	Query-3	Query-4	Query-5
Degree Centrality	**69**	**68**	**69**	**64**	**64**
Closeness Centrality	88	89	90	92	89
Betweenness Centrality	81	79	82	77	78
Eigenvector Centrality	79	78	75	87	81
Candidate Node Selection	356	855	279	1195	972

Table 4.6: The query results with different matching orders on WorldCupDB

Algorithm / Query (ms)	Query-1	Query-2	Query-3	Query-4	Query-5
Degree Centrality	874	3342	**3260**	**3939**	18334
Closeness Centrality	1022	9491	6392	6735	18764
Betweenness Centrality	1097	**3124**	6640	12586	26021
Eigenvector Centrality	1148	3339	3673	7030	**18224**
Candidate Node Selection	**848**	3314	3270	11973	19045

Table 4.7: Total process time without calculating matching order on WorldCupDB

Algorithm / Query (ms)	Query-1	Query-2	Query-3	Query-4	Query-5
Degree Centrality	805	3274	3191	**3875**	18270
Closeness Centrality	934	9402	6302	6643	18675
Betweenness Centrality	1016	3045	6558	12499	25943
Eigenvector Centrality	1069	3261	3598	6943	18743
Candidate Node Selection	**492**	**2168**	**2715**	10778	**17985**

We also apply *ANOVA* on query results of subgraph isomorphism algorithms on WorldCup database to show the effects of state-of-the-art subgraph isomorphism algorithms on subgraph isomorpishm problem. We use query results as given in Appendix A. The results are given in Table 4.8 and 4.9.

Table 4.8: ANOVA results of the effects of state-of-the-art subgraph isomorphism algorithms on subgraph isomorpishm problem

Algorithms	**Count**	**Sum**	**Average**	**Variance**
Cypher	48	627154	13065,70833	87392902,38
DualIso	48	1812112	37752,33333	2257631713
GraphQL	35	616660	17618,85714	131615013,2
TurboIso	30	1156157	38538,56667	880991739,8
VF3	22	923757	41988,95455	445518926,9
BB-Graph	50	427307	8546,14	45323765,8
BB-Plus (Historical)	50	398677	7973,54	40869012,99
BB-Plus (Real-Time)	50	373225	7464,5	37541892,09

Table 4.9: ANOVA results of the effects of state-of-the-art subgraph isomorphism algorithms on subgraph isomorpishm problem

Source of Variation	**SS**	**df**	**MS**	**F**	**P-value**	**F-value**
Between Groups	59912310167	7	8558901452	17,87	4,61376E-20	2,037793878
Within Groups	1,55659E+11	325	478949920,6			
Total	2,15571E+11	332				

According to the results BB-Plus (Historical) and BB-Plus (Real-Time) gives the best results with the queries that executed on WorldCup databases. On the other hand, TurboIso and VF3 gives the worst results.

4.4.2 Experiments on the Pokec Database

In this section, we compare the algorithms using the Pokec database. We execute 4 different queries to evaluate the performance of each algorithm. All queries and their BB-Graph representation can be seen in Table 4.10 and all the results are going to discussed in this chapter.

Table 4.10: The queries and their BB-Graph representation on Pokec Database

Queries	**BB-Graph Represantation**
The user with id 22 who has a friendship with the user with id 2	(0,1,User&UserId=2,friends,OUTGOING,User&UserId=22)
The user list which consists of people who is 17 years old and 'stredoskolske' at education level	(0,1,User,worksIn,OUTGOING,Occupation&EducationLevel=stredoskolske) (0,2,User,hasProfile,OUTGOING,profile&Age=17)
The friends of users who used to skateboarding but now playing basketball	(0,1,User,activeDoing,OUTGOING,Sports&name=basketbal) (0,2,User,passiveDoing,OUTGOING,Sports&name=skateboarding) (0,3,User,friends,OUTGOING,User)
The people who can talk English and 'vysokoskolske' education level that friends with users who is also at 'vysokoskolske' education level and can talk English	(0,1,User,worksIn,OUTGOING,Occupation&EducationLevel=vysokoskolske) (0,2,User,talks,OUTGOING,Language&name=english) (0,3,User,friends,OUTGOING,User) (3,2,User,talks,OUTGOING,Language&name=english)

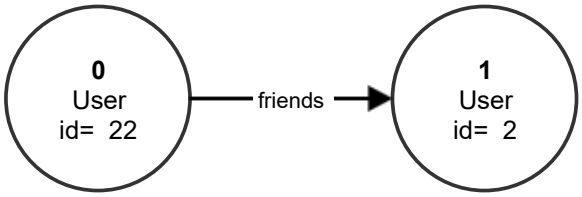

Figure 4.9: The Query1 for Pokec dataset

In the first query, we are looking for *"A User with id 22 who has a friendship with the user with id 2"* in the Pokec database. The query is an example to the path queries and it consists of 2 nodes and 1 edges like shown in Figure 4.9. The results for the query is shown in Table 4.11 and 4.13. According to the results, *the BB-Plus* approach gives the best results only on the historical Pokec database in compare to the other algorithms.

The *BB-Plus* approach uses $0-1$ order based on *Matching Order Selection Based On Candidate Node Selection Without Candidates* and beats all algorithms on historical Pokec database as shown in Table 4.4.

On the other hand, it applies *Matching Order Selection Based On Degree Centrality* with $0-1$ on real-time dPokec database. Because, the query type can be described as *"path query"*. However, the *Cypher* beats the *BB-Plus* approach and all the other algorithms in this query. Cypher beats all the algorithms since the query is too short and does not effected by matching order as shown in 4.13. Table 4.14 is the proof of the *BB-Plus* approach is greater than *Cypher* algorithm, however the time that was spending on calculating of the matching order causes to get behind of it.

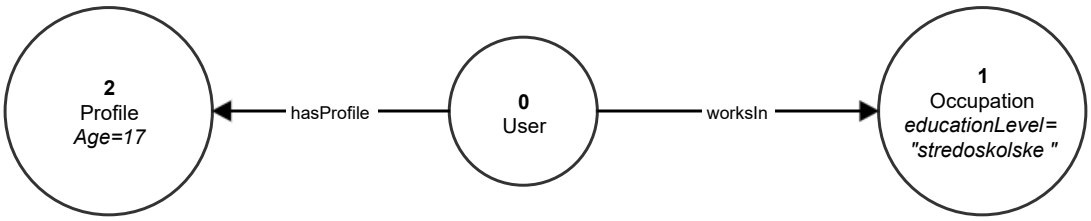

Figure 4.10: The Query2 for Pokec dataset

In the second query, we are looking for *"The user list which consists of people who is 17 years old and 'stredoskolske' at education level"* in the Pokec database. The query is also an example to path queries and it consists of 3 nodes and 2 edges like shown in Figure 4.10. The results for the query is shown in Table 4.11, 4.13. According to the results, *the BB-Plus* approach gives the best results on both historical and real-time Pokec database.

The *BB-Plus* approach uses $(2-1-0)$ order based on *Matching Order Selection Based On Candidate Node Selection Without Candidates* and beats all algorithms on historical Pokec database as shown in Table 4.4.

On the other hand, it applies *Matching Order Selection Based On Closeness Centrality* with $(2-1-0)$ order in real-time Pokec database based on the type of "others" and the edge size is smaller than five and it beats all the other algorithms as shown in Table 4.4.

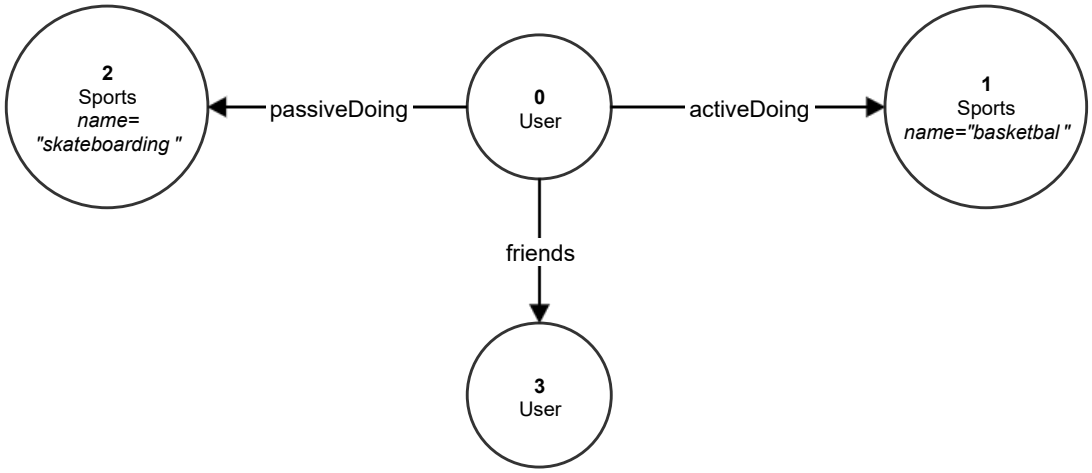

Figure 4.11: The Query3 for Pokec dataset

In the third query, we are looking for *"The friends of users who used to skateboarding but now playing basketball"*. The query consists of 4 nodes and 3 edges as shown in Figure 4.11. The results for the query is shown in Table 4.11, 4.13. According to the results, *the BB-Plus* approach gives the best results on both historical and real-time Pokec database.

The *BB-Plus* approach uses $(1-2-3-0)$ order based on *Matching Order Selection Based On Candidate Node Selection Without Candidates* and beats all algorithms on historical Pokec database as shown in Table 4.4.

On the other hand, it applies *Matching Order Selection Based On Closeness Centrality* with same order in real-time Pokec database based on the type of "others" and the edge size is smaller than five and it beats all the other algorithms again.

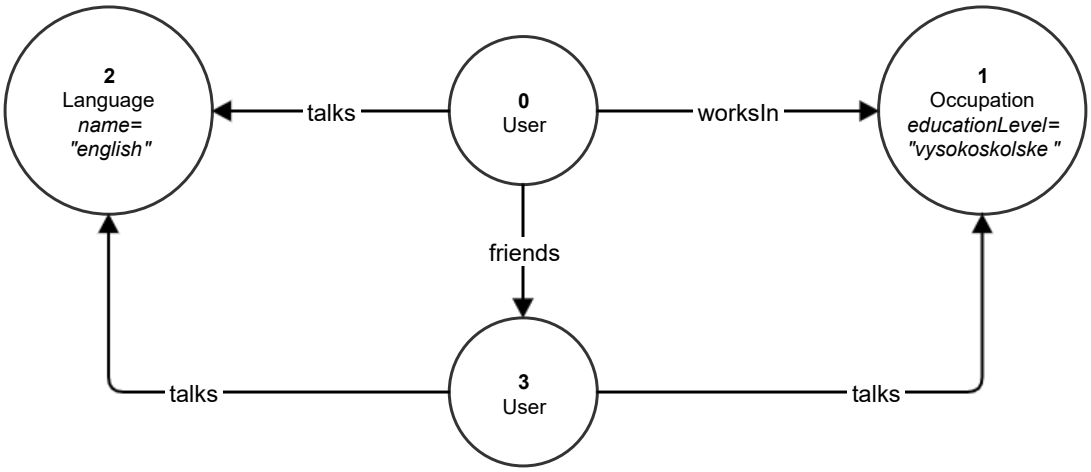

Figure 4.12: The Query4 for Pokec dataset

The last one is about *"The people who can talk English and 'vysokoskolske' education level that friends with users who is also at 'vysokoskolske' education level and can talk English"*. It is a cyclic query with 4 nodes, 5 edges and 2 cycles as shown in Figure 4.12. According to the results as shown in Table 4.11, 4.13, *the BB-Plus* approach gives the best results on both historical and real-time Pokec database.

The *BB-Plus* approach uses $(1-2--0-3)$ order based on *Matching Order Selection Based On Candidate Node Selection Without Candidates* and beats all algorithms on historical Pokec database as shown in Table 4.4.

On the other hand, it applies *Matching Order Selection Based On Degree Centrality* with same order in real-time Pokec database based on *"cyclic"* type of query and edge size is bigger than four and it beats all the other algorithms again.

Table 4.11: The query results for the Pokec Database

Algorithm / Query (ms)	Query-1	Query-2	Query-3	Query-4
Cypher	**94**	2761	4873	4832
DualIso	109	30915	-	30506
GraphQL	108	-	2851	12069
TurboIso	2465	9436	7527	8124
VF3	2998	9532	8126	8576
BB-Graph	126	5940	4734	4700
BB-Plus (Historical) (CNS without Degree Cons.)	67	2021	**1758**	2193
BB-Plus (Historical)	**65**	**2006**	1787	**2115**
BB-Plus (Real-Time)	125	**2565**	**1752**	**1372**

Table 4.12: The query results with different matching orders on Pokec Database

Algorithm / Query (ms)	Query-1	Query-2	Query-3	Query-4
Degree Centrality	**125**	2593	1814	**1372**
Closeness Centrality	127	**2565**	**1752**	1376
Betweenness Centrality	126	5465	5663	5313
Eigenvector Centrality	128	5474	5659	5633
Candidate Node Selection	126	2660	4098	3587

Table 4.13: Total process time for calculating matching order in the Pokec Database

Algorithm / Query (ms)	Query-1	Query-2	Query-3	Query-4
Degree Centrality	**67**	**69**	**61**	**58**
Closeness Centrality	81	85	88	82
Betweenness Centrality	75	78	79	76
Eigenvector Centrality	76	73	76	72
Candidate Node Selection	91	753	1765	1653

Table 4.14: The total process time without calculating matching order selection

Algorithm / Query (ms)	Query-1	Query-2	Query-3	Query-4
Degree Centrality	58	2524	1753	1314
Closeness Centrality	46	2480	**1664**	**1294**
Betweenness Centrality	51	5387	5585	5237
Eigenvector Centrality	52	5401	5583	5561
Candidate Node Selection	**35**	**1907**	2333	1834

Table 4.15: Matching Order of Different Methods for the Pokec Database

Algorithm / Query (ms)	Query-1	Query-2	Query-3	Query-4
Degree Centrality	0-1	1-2-0	1-2-3-0	1-2-0-3
Closeness Centrality	0-1	2-10	1-2-3-0	1-2-0-3
Betweenness Centrality	0-1	0-1-2	0-1-2-3	0-3-1-2
Eigenvector Centrality	0-1	0-1-2	0-1-2-3	0-3-1-2
Candidate Node Selection	0-1	2-10	1-2-3-0	1-2-0-3

Table 4.16: The queries and their BB-Graph representation on Population Database [1]

Queries	BB-Graph Represantation
Extended families consisting of mother, father, son and son's wife and all living in the same address	(0,1,PERSON,SPOUSE,INCOMING,PERSON)(0,2,PERSON,MOTHER,INCOMING,PERSON) (0,3,PERSON,LIVES_IN,OUTGOING,FLAT)(1,2,PERSON,FATHER,INCOMING,PERSON) (1,3,PERSON,LIVES_IN,OUTGOING,FLAT) (2,3,PERSON,LIVES_IN,OUTGOING,FLAT)(2,4,PERSON,SPOUSE,OUTGOING,PERSON) (3,4,FLAT,LIVES_IN,INCOMING,PERSON)
Married couples whose mothers are sisters	(0,1,PERSON,FATHER,INCOMING,PERSON)(0,2,PERSON,FATHER,INCOMING,PERSON) (1,3,PERSON,MOTHER,INCOMING,PERSON)(2,4,PERSON,MOTHER,INCOMING,PERSON) (3,4,PERSON,EŞİ,OUTGOING,PERSON)
Couples whose state registers are different	(0,1,PERSON,SPOUSE,OUTGOING,PERSON)(0,2,PERSON,MEMBER_OF,OUTGOING,STATE_REGISTER) (1,3,PERSON,MEMBER_OF,OUTGOING,STATE_REGISTER)
Fathers and their sons along 8-degree-generation	(0,1,PERSON,FATHER,OUTGOING,PERSON)(1,2,PERSON,FATHER,OUTGOING,PERSON) (2,3,PERSON,FATHER,OUTGOING,PERSON)(3,4,PERSON,FATHER,OUTGOING,PERSON) (4,5,PERSON,FATHER,OUTGOING,PERSON)(5,6,PERSON,FATHER,OUTGOING,PERSON) (6,7,PERSON,FATHER,OUTGOING,PERSON)
Twins who live in different flats of the same apartment which is different from the apartment where their parents live	(0,1,PERSON,SPOUSE,OUTGOING,PERSON)(0,2,PERSON,FATHER,INCOMING,PERSON) (0,3,PERSON,FATHER,INCOMING,PERSON)(0,4,PERSON,LIVES_IN,OUTGOING,FLAT) (1,2,PERSON,MOTHER,INCOMING,PERSON)(1,3,PERSON,MOTHER,INCOMING,PERSON) (1,4,PERSON,LIVES_IN,OUTGOING,FLAT)(2,5,PERSON,BORN,OUTGOING,BIRTH_DAY) (2,6,PERSON,LIVES_IN,OUTGOING,FLAT)(3,5,PERSON,BORN,OUTGOING,BIRTH_DAY) (3,7,PERSON,LIVES_IN,OUTGOING,FLAT)(4,8,FLAT,FLAT_OF_APARTMENT,OUTGOING,APARTMENT) (6,9,FLAT,FLAT_OF_APARTMENT,OUTGOING,APARTMENT)(7,9,FLAT,FLAT_OF_APARTMENT,OUTGOING,APARTMENT)

4.4.3 Experiments on the Population Database

In this section, we compare the algorithms using the Population database. We execute 5 different queries to evaluate the performance of each algorithm. All queries and their BB-Graph representation can be seen in Table 4.16 and all the results are going to discussed in this chapter.

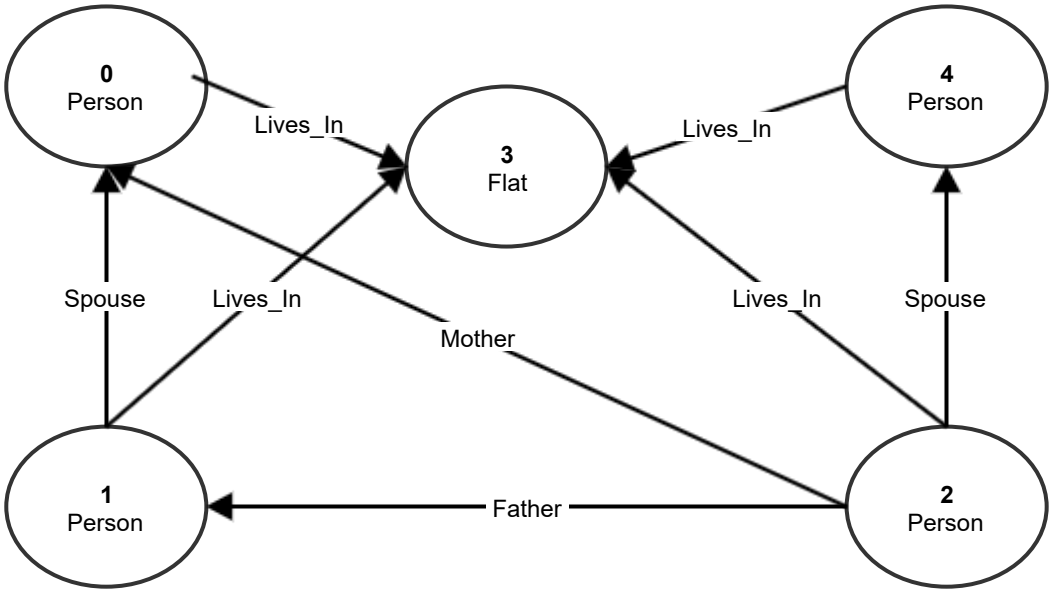

Figure 4.13: The Query1 for Population dataset

First query for the population database is about finding *"Extended families consisting of mother, father, son and son's wife and all living in the same address"*. It consists of 5 nodes and 7 edges as shown in Figure 4.13. According to the results that is given in 4.17, 4.21, the *BB-Plus* approach beats all the algorithms only on real-time Population database and give close results to the winner on the historical Population database.

The *BB-Plus* approach uses $(0{-}1{-}2{-}4{-}3)$ order based on *Matching Order Selection Based On Candidate Node Selection Without Candidates* on historical Pokec database as shown in Table 4.21. However, *BB-Graph* algorithm gives greater result than *BB-Plus* in this query with no matching order.

The *BB-Plus* approach beats all the algorithms with $4-0-1-2-3$ order using *Matching Order Selection Based On Degree Centrality* on real-time Population database. It chooses *Degree Centrality*, because the query is *"cyclic"* and has edges more than four.

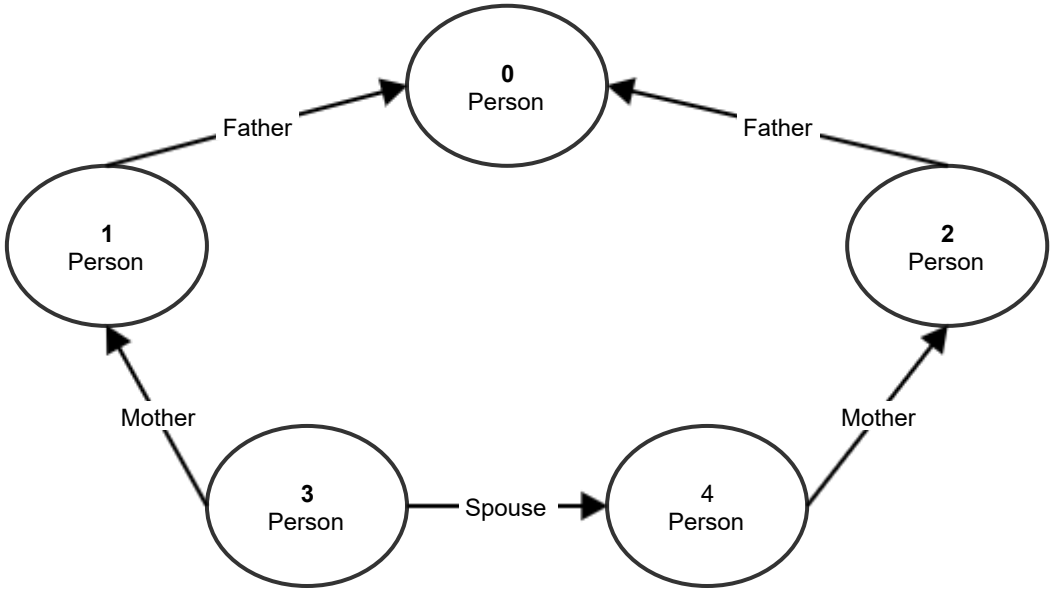

Figure 4.14: The Query2 for Population dataset

In the second query, we are trying to find *"Married couples whose mothers are sisters"*. This cyclic query consists of 5 nodes and 5 edges as shown in Figure 4.14. Query results that is given in 4.17, 4.21 show us that the *BB-Plus* approach gives the best results on both historical and real-time Population database.

The *BB-Plus* approach uses no matching order on both historical and real-time Population database. Because all the nodes and edges in the query graph has the same label and nodes has no properties. Therefore, matching order does not affect the results. The *BB-Plus* approach directly calls the *BB-Graph* algorithm for this query and beats all the other algorithms.

The third query is path query which is about finding *"Couples whose state registers are different"* which is consists of 4 nodes and 3 edges as shown in 4.15. According to the the query results that is given in 4.17, 4.21, the *BB-Plus* approach gives the best results on both historical and real-time Population database.

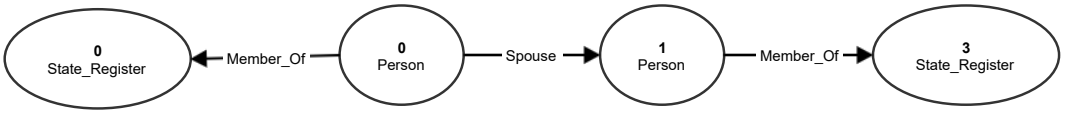

Figure 4.15: The Query3 for Population dataset

The *BB-Plus* approach uses $(2-3-0-1)$ order based on *Matching Order Selection Based On Candidate Node Selection Without Candidates* on historical Pokec database and give greater results than the other algorithms.

The *BB-Plus* approach chooses *Matching Order Selection Based On Degree Centrality*, because the query is type of *"path"* and beats all the algorithms with $4-0-1-2-3$ order on real-time Population database.

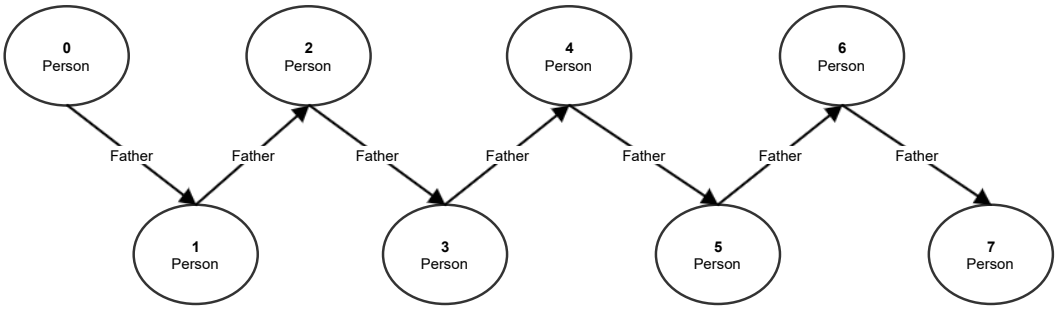

Figure 4.16: The Query4 for Population dataset

The forth query is also a path query that is searching *"Fathers and their sons along 8-degree-generation"*. The query is structured around 8 nodes and 7 edges as shown in 4.16. *BB-Plus* approach gives the best results on both historical and real-time Population database.

The *BB-Plus* approach uses no matching order on both historical and real-time Population database also in this query. Because all the nodes and edges in the query graph has the same label and nodes has no properties again. The *BB-Plus* approach directly calls the *BB-Graph* algorithm for this query and beats all the other algorithms as shown in 4.17, 4.21.

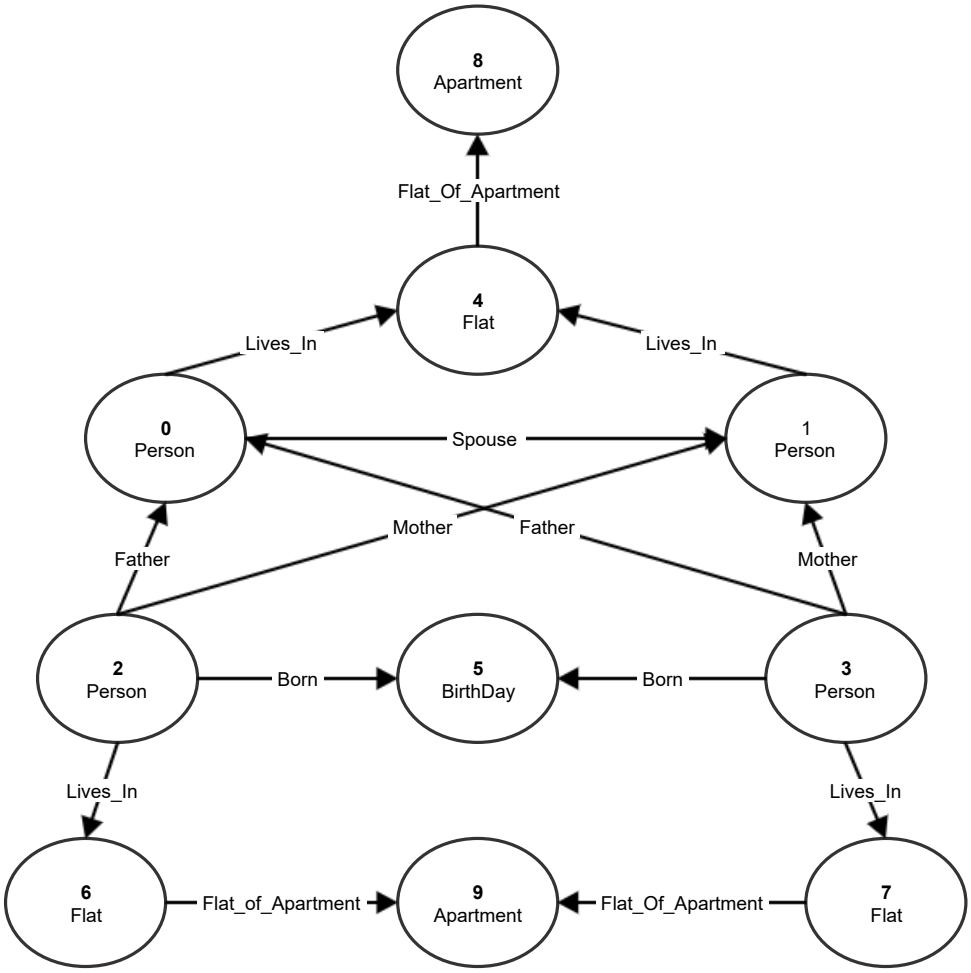

Figure 4.17: The Query5 for Population dataset

The last query is a complex cyclic query for finding *"Twins who live in different flats of the same apartment which is different from the apartment where their parents live"*. It consists of 10 nodes and 14 edges as shown in 4.17. The *BB-Plus* approach is failed to find the results in an acceptable and the *BB-Graph* algorithm sits the first place for this query without taking consideration of matching order as shown in 4.17.

Table 4.17: The query results for the Population Database

Algorithm / Query (ms)	Query-1	Query-2	Query-3	Query-4	Query-5
Cypher	33376	40234	24215	38756	2M >
DualIso	2M >	2M >	2M >	2M >	2M >
GraphQL	2M >	2M >	2M >	2M >	2M >
TurboIso	2M >	2M >	2M >	2M >	2M >
VF3	2M >	2M >	2M >	2M >	2M >
BB-Graph	**31376**	34405	24215	37159	**86452**
BB-Plus (Historical) (CNS without Degree Cons.)	35296	34481	18873	**37140**	2M >
BB-Plus (Historical)	35518	**34223**	**18873**	37063	2M >
BB-Plus (Real-Time)	**28436**	**34401**	**19031**	**37136**	116863 >

Table 4.18: The query results with different matching orders on Population Database

Algorithm / Query (ms)	Query-1	Query-2	Query-3	Query-4	Query-5
Degree Centrality	**28436**	**34402**	**19031**	38521	2M >
Closeness Centrality	1M >	36325	19112	**27512**	2M >
Betweenness Centrality	31526	39853	26847	38984	**116863**
Eigenvector Centrality	28524	34458	18916	39102	2M >
Candidate Node Selection	35542	35095	19101	37172	2M >

Table 4.19: The Total Process Time for Calculating Matching Order in the Population Database

Algorithm / Query (ms)	Query-1	Query-2	Query-3	Query-4	Query-5
Degree Centrality	**64**	**57**	**51**	**51**	**65**
Closeness Centrality	74	72	79	82	99
Betweenness Centrality	71	69	71	67	76
Eigenvector Centrality	**64**	72	81	70	68
Candidate Node Selection	2904	380	595	1037	10375

Table 4.20: The total process time without calculating matching order selection in the Population Database

Algorithm / Query (ms)	Query-1	Query-2	Query-3	Query-4	Query-5
Degree Centrality	**28372**	**34345**	18980	38470	-
Closeness Centrality	-	36253	19033	**27430**	-
Betweenness Centrality	31455	39784	26776	38917	**116787**
Eigenvector Centrality	28460	34386	19053	39032	-
Candidate Node Selection	32860	34493	**17838**	3623	-

Table 4.21: Matching Order of Different Methods for the Population Database

Algorithm / Query (ms)	Query-1	Query-2	Query-3	Query-4	Query-5
Degree Centrality	4-0-1-2-3	0-1-2-3-4	2-3-0-1	0-7-1-2-3-4-5-6	8-5-6-7-9-4-0-1-2-3
Closeness Centrality	3-2-1-0-4	0-3-4-1-2	2-3-0-1	7-0-1-2-3-4-5-6	5-8-9-2-3-0-1-4-6-7
Betweenness Centrality	0-1-2-3-4	0-3-1-2-4	0-2-3-1	0-7-1-6-2-5-3-4	2-3-5-8-9-6-7-0-1-4
Eigenvector Centrality	4-0-1-2-3	0-1-2-3-4	2-3-0-1	0-7-1-6-2-5-3-4	9-8-7-6-5-4-2-3-0-1
Candidate Node Selection	0-1-2-4-3	0-1-2-3-4	2-3-0-1	0-1-2-3-4-5-6-7	5-0-1-2-3-8-9-4-6-7

4.5 Discussions on the Experimental Results

Concept

We propose *BB-Plus* approach which chooses six different matching order selection methods automatically to improve the performance of subgraph isomorphism queries. The new matching orders are *degree centrality*, *closeness centrality*, *betweenness centrality*, *eigenvector centrality*, and *selection based on candidate nodes*.

Discussion

1. *Performance Increase:* With a great matching order, algorithms can easily prune out the data that is not matching with query graph. Therefore, they can reduce the search space and find matching nodes and edges efficiently using less memory and time.

2. *Performance on Different Query Types:* Centrality based matching order selection methods such as *Matching Order Selection Based On Degree Centrality*, *Matching Order Selection Based On Closeness Centrality*, *the BB-Plus with Betweenness Centrality*, *Matching Order Selection Based On Eigenvector Centrality* does not consider data graph. Although their matching order can be easily calculated, they may found a bad matching order because they do not consider data graph.

   However, *Matching Order Selection Based On Candidate Node Selection* considers the query and data graph. Therefore, it does not affected by the query types and find a great matching order that fits with both the query and data graph.

   On the other hand, centrality based matching order selection methods are good for real-time databases. Because they can be easily calculated on execution time and they can adapt themselves to different types of queries.

3. *Drawbacks of Matching Order Selection:*

   (a) *Cost of Calculating Process:* Calculation of matching order reduce the performance when the the query nodes is already in best order. Because the approach loses time for calculating the matching order to find the same order. Therefore, *BB-Graph* or the other subgraph isomorphism algorithms gives better results than *BB-Plus* approach in some experiment.

   (b) *Working with Poor Matching Order:* Matching order increase the performance if the order is selected based on considering both query and data graph. When the matching order is selected poorly, the search space is can be become bigger than the one without matching order and the time for finding matchings can be increase.

Concept

All the algorithms in the comparison are developed using Neo4j's graph data structures such as nodes, edges and indexes. Normally GraphQl, TurboIso and VF3 use their own data structures to store and retrieve nodes and edges.

Discussion

1. *Easy Implemantation:* With Neo4j's graph data structures, the implementation of algorithms become easier. We do not need to develop new data structures for each algorithm. We only need to store nodes and edges of dataset in Neo4j's graph database and use its methods to retrieve those when we need them in any step of any algorithm.

2. *Performance Issues:* GraphQl, TurboIso and VF3, Matching Order Selection Based On Candidate Edge Selection have performance issues based on the usage of Neo4j's graph data structures. Especially, TurboIso and VF3 are affected directly and need their own data structures in most of the queries. For example, retrieving nodes based on an edge can be a big problem in Neo4j. There is no efficient method defined to retrieve them. However, the data structures of TurboIso and VF3 can handled this efficiently. Therefore, in most of the queries TurboIso and VF3 does not give great results as we expected.

3. *Fair Comparison:* Eventhough TurboIso and VF3 affected by the usage Neo4j's graph data structures, the comparison gives fair results to us because all the algorithms are run in the same environment using same data structures.

4. *Memory Usage:* All the algortihms use memory more effectively with Neo4j's graph data structures. They do not need extra memory for large indexing or data storing.

5. *Query Language:* We can query using same query represantation for all the algorithms. We can a query and run for all the algorithms without changing anything on the query represantation.

Concept

Matching Order Selection Based On Candidate Node Selection faster than the other algorithms if the candidate node size given before querying.

Discussion

Eventhough the *Matching Order Selection Based On Candidate Node Selection* is good at finding good matching order and effective querying, calculating candidate node size while querying can be take too much time and it affects the performance of the algorithm. Therefore, the *BB-Plus* approach takes it into the consideration and use the *Matching Order Selection Based On Candidate Node Selection Without Candidates* algorithm which is an extension of the *Matching Order Selection Based On Candidate Node Selection* algorithm that calculate candidate node size before querying in order to do not lose time in matching order calculation and query efficiently.

Table 4.22: Comparison of the BB-Plus approach with other subgraph isomorphism algorithms

	GraphQL	TurboIso	DualIso	BB-Graph	VF3	BB-Plus
Year	2008	2013	2014	2014	2017	2019
Indexing	Label-based Indexing	Pattern-based Indexing	-	-	-	-
Prunning Methods	Neighborhood Signature-based Pruning Pseudo Subgraph Isomorphism Test	Neighborhood Equivalence Class	Simple Simulation Dual Simulation	Matching Node Principal Matching Relationship Principal	Core rule 2 level of look-ahead rule	Matching Node Principal Matching Relationship Principal
Matching Order Selection Method	Candidate Size for Query Node	Candidate Region Exploration	-	-	Node Exploration Sequence	Degree Centrality Closeness Centrality Betweenness Centrality Eigenvector Centrality Candidate Node Selection
Performance on Large Volume of Graph Data	Poor	Poor	Medium	Medium	Poor	Good
Calculation Time of Matching Order (Historical Database)	-	Slow	-	-	Slow	Slow
Calculation Time of Matching Order (Real-Time Database)	-	Slow	-	-	Slow	Fast
Total Process Time	Medium	Slow	Medium	Medium	Slow	Fast
Adaptation to Different Type of Queries	Poor	Good	Poor	Medium	Good	Good
Performance on Path Queries	Medium	Poor	Medium	Medium	Poor	Good
Performance on Cyclic Queries	Medium	Poor	Medium	Medium	Poor	Good
Performance on Other Queries	Medium	Poor	Medium	Medium	Poor	Good

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

In this thesis, we introduce new approach called the *BB-Plus*, which improves the performance of the branch-and-bounnd algorithms by using different matching order selection methods for subgraph isomorpishm queries to find the best matching order and reduce the search space. The approach uses rules to find matching order selection methods automatically, which are degree centrality, closeness centrality, betweenness centrality, eigenvector centrality and candidate node selection.

We compare our *BB-Plus* approach with the *BB-Graph*, *Cypher*, *DualIso*, *GraphQl*, *TurboIso* and *VF3* algorithms with the publicly available WorldCup and Pokec datasets and with a much larger data set of the entire population of a country, the Population dataset. In most of the queries, our *BB-Plus* approach outperforms the rest of all the algorithms for most queries in these three dataset.

On the other hand, we realize that if we do not take into account the computation time of the matching order, the *Matching Order Selection Based On Candidate Node Selection* gives better results than BB-Graph, Cypher, DualIso, GraphQl, TurboIso, VF3 and other matching order selection methods with branch-bound algorithms. Most of the time, the algorithm beats or gives very close results to the other improved BB-Graph algorithms. Therefore, we divide queries based on database that they are executed.

In historical databases, we use *Matching Order Selection with Candidate Node Selection Without Candidates* that uses already calculated candidate node size list. However, on real-time databases, we uses graph centrality measures as matching order. Because they can be easily calculated and they adapt themselves easily to the any type of query. Therefore, the BB-Plus approach becomes the best algorithm among all the algorithms on both historical and real-time databases.

In addition, we see that in some queries for some matching order selection method, the *BB-Graph* algorithm (or the other algorithms) still gives better results. The reason *BB-Plus* approach is worse than the *BB-Graph* algorithm is that the *BB-Graph* query is already given in the best selection order to the algorithm. The *BB-Graph* does not consider the time for calculating the order unlike ours and the others. Therefore, it can give better results than others with the difference in processing time of computing the matching order.

Although GraphQL does not give the worst result for a small database such as WorldCup, it cannot handle large data sets such as Pokec and Population and gives no results in a reasonable amount of time. On the other hand, VF3, DualIso and TurboIso consider matching order selection, they are both worse than the BB-Graph and the BB-Plus algorithms. The reason of low performance of VF3, DualIso and TurboIso is determined by the use of Neo4j's graph data structures. Normally, these algorithms use their own data structures. However, with the usage of Neo4j's data structures, it has become very difficult to find nodes with a corresponding relationship and we detect that it affects their performance.

5.2 Future Work

As we pointed out in the conclusion, all the matching order selection methods used in the BB-Plus approach are not affected by query types. Their performance is determined by the distribution of the nodes in the data graph. For future works, different machine learning algorithms or different inputs could be used in the dataset or the size of the dataset could be increased to determine which matching order selection methods performs best with which query types in order to improve the performance.

REFERENCES

[1] M. Asiler and A. Yazıcı, "Bb-graph: A new subgraph isomorphism algorithm for efficiently querying big graph databases," *arXiv preprint arXiv:1706.06654*, 2017.

[2] H. He and A. K. Singh, "Closure-tree: An index structure for graph queries," in *22nd International Conference on Data Engineering (ICDE'06)*, pp. 38–38, IEEE, 2006.

[3] W.-S. Han, J. Lee, and J.-H. Lee, "Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 337–348, ACM, 2013.

[4] P. Bhatia and B. Mallick, "Critique of wordcount blueprint by virtue of mapreduce postulate," *International Journal of Advanced Research in Computer and Communication Engineer (IJARCCE)*, vol. 5, no. 2, 2016.

[5] R. Angles, "A comparison of current graph database models," in *IEEE 28th International Conference on Data Engineering Workshops (ICDEW)*, pp. 171–177, IEEE, 2012.

[6] S. Jouili and V. Vansteenberghe, "An empirical comparison of graph databases," in *2013 International Conference on Social Computing (SocialCom)*, pp. 708–715, IEEE, 2013.

[7] A. Nayak, A. Poriya, and D. Poojary, "Type of nosql databases and its comparison with relational databases," *International Journal of Applied Information Systems*, vol. 5, no. 4, pp. 16–19, 2013.

[8] A. Moniruzzaman and S. A. Hossain, "Nosql database: New era of databases for big data analytics-classification, characteristics and comparison," *arXiv preprint arXiv:1307.0191*, 2013.

[9] R. Giugno and D. Shasha, "Graphgrep: A fast and universal method for querying graphs," in *Object recognition supported by user interaction for service robots*, vol. 2, pp. 112–115, IEEE, 2002.

[10] 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*, pp. 335–346, ACM, 2004.

[11] 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*, pp. 138–147, 2005.

[12] D. W. Williams, J. Huan, and W. Wang, "Graph database indexing using structured graph decomposition," in *2007 IEEE 23rd International Conference on Data Engineering*, pp. 976–985, IEEE, 2007.

[13] S. Zhang, M. Hu, and J. Yang, "Treepi: A novel graph indexing method," in *2007 IEEE 23rd International Conference on Data Engineering*, pp. 966–975, IEEE, 2007.

[14] 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*, pp. 938–949, VLDB Endowment, 2007.

[15] 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*, vol. 1, no. 1, pp. 364–375, 2008.

[16] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub) graph isomorphism algorithm for matching large graphs," *IEEE transactions on pattern analysis and machine intelligence*, vol. 26, no. 10, pp. 1367–1372, 2004.

[17] 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*, pp. 192–203, ACM, 2009.

[18] 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*, pp. 405–418, ACM, 2008.

[19] P. Zhao and J. Han, "On graph query optimization in large networks," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 340–351, 2010.

[20] M. Saltz, A. Jain, A. Kothari, A. Fard, J. A. Miller, and L. Ramaswamy, "Dualiso: An algorithm for subgraph pattern matching on very large labeled graphs," in *Big Data (BigData Congress), 2014 IEEE International Congress on*, pp. 498–505, IEEE, 2014.

[21] V. Carletti, P. Foggia, A. Saggese, and M. Vento, "Introducing vf3: A new algorithm for subgraph isomorphism," in *International Workshop on Graph-Based Representations in Pattern Recognition*, pp. 128–139, Springer, 2017.

[22] V. Carletti, P. Foggia, and M. Vento, "Vf2 plus: An improved version of vf2 for biological graphs," in *International Workshop on Graph-Based Representations in Pattern Recognition*, pp. 168–177, Springer, 2015.

[23] M. Ciglan, A. Averbuch, and L. Hluchy, "Benchmarking traversal operations over graph databases," in *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, pp. 186–189, IEEE, 2012.

[24] R. De Virgilio, A. Maccioni, and R. Torlone, "Converting relational to graph databases," in *First International Workshop on Graph Data Management Experiences and Systems*, p. 1, ACM, 2013.

[25] 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*, p. 42, ACM, 2010.

[26] R. Angles and C. Gutierrez, "Survey of graph database models," *ACM Computing Surveys (CSUR)*, vol. 40, no. 1, p. 1, 2008.

[27] J. J. Miller, "Graph database applications and concepts with neo4j," in *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, vol. 2324, p. 36, 2013.

[28] P. Pareja-Tobes, E. Pareja-Tobes, M. Manrique, E. Pareja, and R. Tobes, "Bio4j: An open source biological data integration platform.," in *IWBBIO*, p. 281, 2013.

[29] B. Iordanov, "Hypergraphdb: a generalized graph database," in *International conference on web-age information management*, pp. 25–36, Springer, 2010.

[30] H. Huang and Z. Dong, "Research on architecture and query performance based on distributed graph database neo4j," in *Consumer Electronics, Communications and Networks (CECNet), 2013 3rd International Conference on*, pp. 533–536, IEEE, 2013.

[31] O. Manual, "Distributed architecture." `http://orientdb.com/docs/2.1/Distributed-Architecture.html`. Accessed: 2016-04-04.

[32] C. Phillips, "Centrality measures." `http://web.eecs.utk.edu/~cphillip/cs594_spring2015_projects/CentralityProject.pdf`. Accessed: 2019-07-21.

[33] N. Matas, "Comparing network centrality measures as tools for identifying key concepts in complex networks: A case of wikipedia.," *Journal of Digital Information Management*, vol. 15, no. 4, 2017.

[34] 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*, vol. 6, pp. 133–144, VLDB Endowment, 2012.

[35] J. R. Ullmann, "An algorithm for subgraph isomorphism," in *Journal of the ACM (JACM)*, pp. 31–42, ACM, 1976.

[36] S. Zhang, S. Li, and J. Yang, "Summa: subgraph matching in massive graphs," in *Proceedings of the 19th ACM international conference on Information and knowledge management*, pp. 1285–1288, ACM, 2010.

[37] N. Staffl, "World cup fun with neo4j." `http://worldcup.neo4j.org`. Accessed: 2018-12-08.

[38] J. Leskovec, "Pokec social network." `https://snap.stanford.edu/data/soc-Pokec.html`. Accessed: 2018-12-08.

APPENDIX A

APPENDIX 1

A.1 Matching Order Training Dataset

Table A.1: Training Data for Determining Matching Order (1)

	Query	Dataset	Data Graph Node Size	Data Graph Edge Size	Query Graph Node Size	Query Graph Edge Size	# of Distinct Node Label	# of Distinct Edge Label	# of Nodes with Properties	Query Type	Best Results
1	Players who join squad of different countries	WorldCup	Small (45348)	Small (86577)	5	4	3	2	0	Path	DC
2	Players who take role as both substitute and starter in the same match	WorldCup	Small (45348)	Small (86577)	4	4	3	3	0	Cyclic	BC
3	Matches between the same countries occurred in different world cups	WorldCup	Small (45348)	Small (86577)	7	8	4	3	0	Cyclic	DC
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	WorldCup	Small (45348)	Small (86577)	10	12	6	6	0	Cyclic	DC
5	Players who take role in any match at least 3 world cups	WorldCup	Small (45348)	Small (86577)	10	9	4	3	0	Others	EC
6	Players who take role in any match at least 2 world cups	WorldCup	Small (45348)	Small (86577)	7	6	4	3	0	Others	EC
7	Players scores a hat-trick as starters	WorldCup	Small (45348)	Small (86577)	5	4	3	2	0	Others	CC
8	Players scores at least 5 goals as starters	WorldCup	Small (45348)	Small (86577)	7	6	3	2	0	Others	EC
9	Countries which hosted World Cup more than one time	WorldCup	Small (45348)	Small (86577)	3	2	2	1	0	Path	DC
10	Players who take role as substitute and scores goal	WorldCup	Small (45348)	Small (86577)	3	2	3	2	0	Path	DC
11	Two countries played at least 2 matches (as away team in one and home team in the other) in the same world cup	WorldCup	Small (45348)	Small (86577)	5	6	3	3	0	Cyclic	BC
12	Players in the squad of England Team in 1990 World Cup	WorldCup	Small (45348)	Small (86577)	4	3	4	3	2	Others	CC
13	Players in the squad of Italy Team in 1990 World Cup	WorldCup	Small (45348)	Small (86577)	4	3	4	3	2	Others	CC
14	Players in the squad of France Team in 1998 World Cup	WorldCup	Small (45348)	Small (86577)	4	3	4	3	2	Others	CC

Table A.2: Training Data for Determining Matching Order (2)

15	Players in the squad of Brazil Team in 2002 World Cup	WorldCup	Small (45348)	Small (86577)	4	3	4	3	2	Others	CC
16	Players in the squad of England Team in 1986 World Cup	WorldCup	Small (45348)	Small (86577)	4	3	4	3	2	Others	EC
17	Players in the squad of Italy Team in 1982 World Cup	WorldCup	Small (45348)	Small (86577)	4	3	4	3	2	Others	CC
18	Players in the squad of Argentina Team in 1978 World Cup	WorldCup	Small (45348)	Small (86577)	4	3	4	3	2	Others	CC
19	Players in the squad of Germany Team in 1974 World Cup	WorldCup	Small (45348)	Small (86577)	4	3	4	3	2	Others	CC
20	Players in the squad of Brazil Team 1970 in World Cup	WorldCup	Small (45348)	Small (86577)	4	3	4	3	2	Others	EC
21	All World Cups hosted by France	WorldCup	Small (45348)	Small (86577)	2	1	2	1	1	Path	DC
22	All World Cups hosted by England	WorldCup	Small (45348)	Small (86577)	2	1	2	1	1	Path	DC
23	All World Cups hosted by Italy	WorldCup	Small (45348)	Small (86577)	2	1	2	1	1	Path	DC
24	All World Cups hosted by Brazil	WorldCup	Small (45348)	Small (86577)	2	1	2	1	1	Path	DC
25	All matches played in San Siro Stadium	WorldCup	Small (45348)	Small (86577)	2	1	2	1	1	Path	DC
26	All matches played in Olympiastadion Stadium	WorldCup	Small (45348)	Small (86577)	2	1	2	1	1	Path	DC
27	All matches played in San Siro Stadium	WorldCup	Small (45348)	Small (86577)	2	1	2	1	1	Path	DC
28	All matches played in Stade de France Stadium	WorldCup	Small (45348)	Small (86577)	2	1	2	1	1	Path	DC
29	All matches played in Rose Bowl Stadium	WorldCup	Small (45348)	Small (86577)	2	1	2	1	1	Path	DC
30	Matches between France and Brazil in 1998 World Cup	WorldCup	Small (45348)	Small (86577)	4	3	3	3	3	Others	CC
31	Matches between Germany and Brazil in 2002 World Cup	WorldCup	Small (45348)	Small (86577)	4	3	3	3	3	Others	CC
32	Matches between Uruguay and Brazil in 1994 World Cup	WorldCup	Small (45348)	Small (86577)	4	3	3	3	3	Others	CC
33	Matches between Italy and Brazil in 1994 World Cup	WorldCup	Small (45348)	Small (86577)	4	3	3	3	3	Others	EC
34	Matches between Italy and France in 1994 World Cup	WorldCup	Small (45348)	Small (86577)	4	3	3	3	3	Others	CC
35	All players takes role as substitute in any match that played in Olympiastadion Stadium in any World Cup	WorldCup	Small (45348)	Small (86577)	5	4	5	4	1	Others	CC
36	All players takes role as substitute in any match that played in Daegu World Cup Stadium in any World Cup	WorldCup	Small (45348)	Small (86577)	5	4	5	4	1	Others	CC
37	All players takes role as substitute in any match that played in Stade de France Stadium in any World Cup	WorldCup	Small (45348)	Small (86577)	5	4	5	4	1	Others	CC
38	All players takes role as substitute in any match that played in Rose Bowl in any World Cup	WorldCup	Small (45348)	Small (86577)	5	4	5	4	1	Others	CC
39	All players takes role as substitute in any match that played in San Siro Stadium in any World Cup	WorldCup	Small (45348)	Small (86577)	5	4	5	4	1	Others	CC
40	All matches of Brazil in a World Cup that hosted by them	WorldCup	Small (45348)	Small (86577)	3	4	3	4	1	Cyclic	BC

Table A.3: Training Data for Determining Matching Order (3)

41	All matches of France in a World Cup that hosted by them	WorldCup	Small (45348)	Small (86577)	3	4	3	4	1	Cyclic	BC
42	All matches of Argentina in a World Cup that hosted by them	WorldCup	Small (45348)	Small (86577)	3	4	3	4	1	Cyclic	BC
43	All matches of Germany in a World Cup that hosted by them	WorldCup	Small (45348)	Small (86577)	3	4	3	4	1	Cyclic	BC
44	All players takes role as starters and scores a hat trick in any match that played in San Siro Stadium	WorldCup	Small (45348)	Small (86577)	7	6	5	4	1	Others	EC
45	All players takes role as starters and scores a hat trick in any match that played in Olympiastadion Stadium	WorldCup	Small (45348)	Small (86577)	7	6	5	4	1	Others	EC
46	All players takes role as starters and scores a hat trick in any match that played in Daegu World Cup Stadium	WorldCup	Small (45348)	Small (86577)	7	6	5	4	1	Others	EC
47	All players takes role as starters and scores a hat trick in any match that played in Stade de France Stadium	WorldCup	Small (45348)	Small (86577)	7	6	5	4	1	Others	CC
48	All players takes role as starters and scores a hat trick in any match that played in Rose Bowl Stadium	WorldCup	Small (45348)	Small (86577)	7	6	5	4	1	Others	EC
49	Countries scores at least 5 goals as home team in any match at World Cup which is hosted by the same country	WorldCup	Small (45348)	Small (86577)	3	3	3	3	1	Cyclic	BC
50	Countries scores at least 4 goals as home team in any match at World Cup which is hosted by the same country	WorldCup	Small (45348)	Small (86577)	3	3	3	3	1	Cyclic	BC
51	Countries scores at least 3 goals as away team in any match at World Cup which is hosted by the home teams' country	WorldCup	Small (45348)	Small (86577)	4	4	3	3	1	Cyclic	BC
52	World Cups hosted by at least two countries	WorldCup	Small (45348)	Small (86577)	3	2	2	1	0	Path	DC
53	World Cups hosted by at least three countries	WorldCup	Small (45348)	Small (86577)	4	3	2	1	0	Others	CC
54	Players who is in the squad of same team that played at least two different World Cup	WorldCup	Small (45348)	Small (86577)	6	6	4	3	0	Cyclic	DC
55	Players who is in the squad of same team that played at least three different World Cup	WorldCup	Small (45348)	Small (86577)	6	6	4	3	0	Cyclic	DC
56	Players who is in the squad of Brazil team that played at least two different World Cup	WorldCup	Small (45348)	Small (86577)	6	6	4	3	0	Cyclic	DC
57	Players who is in the squad of England team that played at least two different World Cup	WorldCup	Small (45348)	Small (86577)	6	6	4	3	0	Cyclic	DC
58	Players who is in the squad of France team that played at least two different World Cup	WorldCup	Small (45348)	Small (86577)	6	6	4	3	0	Cyclic	DC
59	Players who is in the squad of Italy team that played at least two different World Cup	WorldCup	Small (45348)	Small (86577)	6	6	4	3	1	Cyclic	DC

Table A.4: Training Data for Determining Matching Order (4)

60	Players who is in the squad of Argentina team that played at least two different World Cup	WorldCup	Small (45348)	Small (86577)	6	6	4	3	1	Cyclic	DC
61	Stadiums used for at least two World Cups	WorldCup	Small (45348)	Small (86577)	5	4	3	2	0	Path	DC
62	Goals that Ronaldo scores in 1998 World Cup	WorldCup	Small (45348)	Small (86577)	5	5	5	5	2	Others	CC
63	Goals that Raul scores in 1998 and 2002 World Cup	WorldCup	Small (45348)	Small (86577)	5	5	5	5	3	Others	CC
64	Goals that Michel Platini scores in 1978, 1982 and 1986 World Cup	WorldCup	Small (45348)	Small (86577)	5	5	5	5	4	Others	EC
65	Goals that Thierry Henry scores in 1998 and 2006 World Cup	WorldCup	Small (45348)	Small (86577)	5	5	5	5	3	Others	CC
66	Goals that Dennis Bergkamp scores in 1994 and 1998 World Cup	WorldCup	Small (45348)	Small (86577)	5	5	5	5	3	Others	CC
67	Goals that Rivaldo scores in 1998 and 2002 World Cup	WorldCup	Small (45348)	Small (86577)	5	5	5	5	3	Others	CC
68	All the goals that Pelé scores in any World Cup	WorldCup	Small (45348)	Small (86577)	5	5	5	5	1	Others	CC
69	All the goals that Ronaldo scores in any World Cup	WorldCup	Small (45348)	Small (86577)	5	5	5	5	1	Others	CC
70	All the goals that Thierry Henry scores in any World Cup	WorldCup	Small (45348)	Small (86577)	5	5	5	5	1	Others	CC
71	All the goals that Dennis Bergkamp scores in any World Cup	WorldCup	Small (45348)	Small (86577)	5	5	5	5	1	Others	EC
72	All the goals that Raul scores in any World Cup	WorldCup	Small (45348)	Small (86577)	5	5	5	5	1	Others	CC
73	Players bigger than 30 years old that both take role as both substitute and starter in the same match	WorldCup	Small (45348)	Small (86577)	3	3	3	3	1	Others	CC
74	Players bigger than 30 years old that both take role as both substitute and starter in the same match and scores a goal	WorldCup	Small (45348)	Small (86577)	4	4	4	4	1	Others	CC
75	Players take role as starter in at least two match in 1998 World Cup	WorldCup	Small (45348)	Small (86577)	6	5	4	3	1	Others	EC
76	Players take role as starter in at least two match in 1994 World Cup	WorldCup	Small (45348)	Small (86577)	6	5	4	3	1	Others	CC
77	Players take role as starter in at least two match in 1990 World Cup	WorldCup	Small (45348)	Small (86577)	6	5	4	3	1	Others	EC
78	Players take role as starter in at least two match in 2002 World Cup	WorldCup	Small (45348)	Small (86577)	6	5	4	3	1	Others	EC
79	Players take role as starter in at least two match in 1986 World Cup	WorldCup	Small (45348)	Small (86577)	6	5	4	3	1	Others	EC
80	All matches of Brazil as an away team in 1998 World Cup	WorldCup	Small (45348)	Small (86577)	3	2	3	2	2	Path	DC
81	All matches of France as an away team in 1998 World Cup	WorldCup	Small (45348)	Small (86577)	3	2	3	2	2	Path	DC
82	All matches of England as an away team in 1998 World Cup	WorldCup	Small (45348)	Small (86577)	3	2	3	2	2	Path	DC
83	All matches of Argentina as an away team in 1998 World Cup	WorldCup	Small (45348)	Small (86577)	3	2	3	2	2	Path	DC
84	All matches of Germany as an away team in 1998 World Cup	WorldCup	Small (45348)	Small (86577)	3	2	3	2	2	Path	DC

Table A.5: Training Data for Determining Matching Order (5)

85	All World Cups that any player scores a hat trick in any match	WorldCup	Small (45348)	Small (86577)	5	5	3	2	1	Others	CC
86	All World Cups that any player scores at least 4 goals in any match	WorldCup	Small (45348)	Small (86577)	8	8	5	5	1	Others	EC
87	All World Cups that any player scores at least 5 goals in any match	WorldCup	Small (45348)	Small (86577)	4	4	5	5	1	Others	CC
88	All stadiums that Final Matches are played	WorldCup	Small (45348)	Small (86577)	2	1	2	1	1	Path	DC
89	Home and away teams of final matches of all World Cups	WorldCup	Small (45348)	Small (86577)	4	3	3	3	1	Others	CC
90	Countries played at finals of a World Cup that is hosted by the same country	WorldCup	Small (45348)	Small (86577)	3	4	3	4	1	Cyclic	BC
91	Countries played at semi-finals of a World Cup that is hosted by the same country	WorldCup	Small (45348)	Small (86577)	3	4	3	4	1	Cyclic	BC
92	Players scores at finals games	WorldCup	Small (45348)	Small (86577)	4	4	4	4	1	Others	CC
93	Players scores at semi-finals games	WorldCup	Small (45348)	Small (86577)	4	4	4	4	1	Others	CC
94	Players takes role as starters at 1998 World Cup Finals and score at least one goal	WorldCup	Small (45348)	Small (86577)	5	4	5	4	2	Others	CC
95	Players takes role as starters at 1998 World Cup Semi-Finals and score at least one goal	WorldCup	Small (45348)	Small (86577)	5	4	5	4	2	Others	CC
96	Players takes role as starters at 1990 World Cup Finals and score at least one goal	WorldCup	Small (45348)	Small (86577)	5	4	5	4	2	Others	CC
97	Players takes role as starters at 1990 World Cup Semi-Finals and score at least one goal	WorldCup	Small (45348)	Small (86577)	5	4	5	4	2	Others	CC
98	Players takes role as starters of home team at 1994 World Cup Finals	WorldCup	Small (45348)	Small (86577)	5	4	5	4	2	Others	CC
99	Players takes role as starters of away team at 1994 World Cup Finals	WorldCup	Small (45348)	Small (86577)	5	4	5	4	2	Others	EC
100	The user with id 22 who has a friendship with the user with id 2	Pokec	Medium (1632803)	Medium (30622564)	2	1	2	1	2	Path	DC
101	The user with id 1 who has a friendship with the user with id 131	Pokec	Medium (1632803)	Medium (30622564)	2	1	2	1	2	Path	DC
102	The user with id 8 who has a friendship with the user with id 495258	Pokec	Medium (1632803)	Medium (30622564)	2	1	2	1	2	Path	DC
103	The user with id 1822 who has a friendship with the user with id 2922	Pokec	Medium (1632803)	Medium (30622564)	2	1	2	1	2	Path	DC
104	The user with id 151 who has a friendship with the user with id 141	Pokec	Medium (1632803)	Medium (30622564)	2	1	2	1	2	Path	DC
105	The user list which consists of people who is 17 years old and 'stredoskolske' at education level	Pokec	Medium (1632803)	Medium (30622564)	3	2	3	2	2	Path	CC
106	The user list which consists of people who is 18 years old and 'stredoskolske' at education level	Pokec	Medium (1632803)	Medium (30622564)	3	2	3	2	2	Path	DC

Table A.6: Training Data for Determining Matching Order (6)

107	The user list which consists of people who is 19 years old and 'stredoskolske' at education level	Pokec	Medium (1632803)	Medium (30622564)	3	2	3	2	2	Path	CC
108	The user list which consists of people who is 20 years old and 'stredoskolske' at education level	Pokec	Medium (1632803)	Medium (30622564)	3	2	3	2	2	Path	DC
109	The user list which consists of people who is 21 years old and 'stredoskolske' at education level	Pokec	Medium (1632803)	Medium (30622564)	3	2	3	2	2	Path	CC
110	The friends of users who used to skateboarding but now playing basketball	Pokec	Medium (1632803)	Medium (30622564)	4	3	2	3	2	Others	CC
111	The friends of users who used to squash but now playing tennis	Pokec	Medium (1632803)	Medium (30622564)	4	3	2	3	2	Others	CC
112	The friends of users who used to tennis but now playing basketball	Pokec	Medium (1632803)	Medium (30622564)	4	3	2	3	2	Others	CC
113	The friends of users who used to football but now playing basketball	Pokec	Medium (1632803)	Medium (30622564)	4	3	2	3	2	Others	CC
114	The friends of users who used to basketball but now playing football	Pokec	Medium (1632803)	Medium (30622564)	4	3	2	3	2	Others	CC
115	The people who can talk 'English' and 'vysokoskolske' education level that	Pokec	Medium (1632803)	Medium (30622564)	4	5	3	3	2	Cyclic	DC
	friends with users who is also at 'vysokoskolske'										
	education level and can talk 'English'										
116	The people who can talk 'Italian' and 'vysokoskolske' education level that	Pokec	Medium (1632803)	Medium (30622564)	4	5	3	3	2	Cyclic	DC
	friends with users who is also at 'vysokoskolske'										
	education level and can talk 'Italian'										
117	The people who can talk 'Spanish' and 'vysokoskolske' education level that	Pokec	Medium (1632803)	Medium (30622564)	4	5	3	3	2	Cyclic	DC
	friends with users who is also at 'vysokoskolske'										
	education level and can talk 'Spanish'										
118	The people who can talk 'French' and 'vysokoskolske' education level that	Pokec	Medium (1632803)	Medium (30622564)	4	5	3	3	2	Cyclic	BC
	friends with users who is also at 'vysokoskolske'										
	education level and can talk 'French'										
119	The people who can talk 'German' and 'vysokoskolske' education level that	Pokec	Medium (1632803)	Medium (30622564)	4	5	3	3	2	Cyclic	BC
	friends with users who is also at 'vysokoskolske'										
	education level and can talk 'German'										
120	Users talk English and play hockey and basketball	Pokec	Medium (1632803)	Medium (30622564)	4	3	3	3	3	Others	CC

Table A.7: Training Data for Determining Matching Order (7)

121	Users talk French and play football and basketball	Pokec	Medium (1632803)	Medium (30622564)	4	3	3	3	3	Others	CC
122	Users talk Italian and play tennis and squash	Pokec	Medium (1632803)	Medium (30622564)	4	3	3	3	3	Others	CC
123	Users talk Dutch and play tennis and basketball	Pokec	Medium (1632803)	Medium (30622564)	4	3	3	3	3	Others	CC
124	Users talks Spanish and play hockey and tennis	Pokec	Medium (1632803)	Medium (30622564)	4	3	3	3	3	Others	EC
125	Friends play basketball	Pokec	Medium (1632803)	Medium (30622564)	3	3	2	3	1	Cyclic	BC
126	Friends play hockey	Pokec	Medium (1632803)	Medium (30622564)	3	3	2	3	1	Cyclic	BC
127	Friends play football	Pokec	Medium (1632803)	Medium (30622564)	3	3	2	3	1	Cyclic	BC
128	Friends play tennis	Pokec	Medium (1632803)	Medium (30622564)	3	3	2	3	1	Cyclic	BC
129	Friends play volleyball	Pokec	Medium (1632803)	Medium (30622564)	3	3	2	3	1	Cyclic	BC
130	Friends of friends of user with id 1	Pokec	Medium (1632803)	Medium (30622564)	3	2	1	1	1	Path	DC
131	Friends of friends of user with id 2	Pokec	Medium (1632803)	Medium (30622564)	3	2	1	1	1	Path	DC
132	Friends of friends of user with id 3	Pokec	Medium (1632803)	Medium (30622564)	3	2	1	1	1	Path	DC
133	Friends of friends of user with id 4	Pokec	Medium (1632803)	Medium (30622564)	3	2	1	1	1	Path	CC
134	Friends of friends of user with id 5	Pokec	Medium (1632803)	Medium (30622564)	3	2	1	1	1	Path	DC
135	Friends both talks English and French	Pokec	Medium (1632803)	Medium (30622564)	3	2	2	2	2	Cyclic	BC
136	Friends both talks English and German	Pokec	Medium (1632803)	Medium (30622564)	3	2	2	2	2	Cyclic	BC
137	Friends both talks English and Slovak	Pokec	Medium (1632803)	Medium (30622564)	3	2	2	2	2	Cyclic	BC
138	Friends both talks Slovak and French	Pokec	Medium (1632803)	Medium (30622564)	3	2	2	2	2	Cyclic	BC
139	Friends both talks Slovak and German	Pokec	Medium (1632803)	Medium (30622564)	3	2	2	2	2	Cyclic	BC
140	Female users play football and friend with 1 female users that play football	Pokec	Medium (1632803)	Medium (30622564)	4	5	2	2	2	Cyclic	BC
141	Female users play football and friend with 2 female users that play football	Pokec	Medium (1632803)	Medium (30622564)	5	8	2	2	2	Cyclic	BC
142	Female users play football and friend with 3 female users that play football	Pokec	Medium (1632803)	Medium (30622564)	6	11	2	2	2	Cyclic	DC
143	Female users play football and friend with 4 female users that play football	Pokec	Medium (1632803)	Medium (30622564)	7	14	2	2	2	Cyclic	DC
144	Female users play football and friend with 5 female users that play football	Pokec	Medium (1632803)	Medium (30622564)	8	17	2	2	2	Cyclic	DC
145	Users with leo zodiac and 17 years old friends with libra zodiac and 17 years old	Pokec	Medium (1632803)	Medium (30622564)	6	5	2	2	3	Others	EC

Table A.8: Training Data for Determining Matching Order (8)

146	Users with virgo zodiac and 18 years old friends with libra zodiac and 18 years old	Pokec	Medium (1632803)	Medium (30622564)	6	5	2	2	3	Others	EC
147	Users with gemini zodiac and 19 years old friends with virgo zodiac and 19 years old	Pokec	Medium (1632803)	Medium (30622564)	6	5	2	2	3	Others	EC
148	Users with gemini zodiac and 20 years old friends with capricorn zodiac and 20 years old	Pokec	Medium (1632803)	Medium (30622564)	6	5	2	2	3	Others	EC
149	Users with leo zodiac and 21 years old friends with scorpio zodiac and 21 years old	Pokec	Medium (1632803)	Medium (30622564)	6	5	2	2	3	Others	EC
150	Extended families consisting of mother, father, son and son's wife and all living in the same address	Population	Big (70422787)	Big (77163109)	5	8	2	4	0	Cyclic	DC
151	Extended families consisting of mother, father, daughter and daughter's husband and all living in the same address	Population	Big (70422787)	Big (77163109)	5	8	2	4	0	Cyclic	DC
152	Married couples whose mothers are sisters	Population	Big (70422787)	Big (77163109)	5	5	1	3	0	Cyclic	BB-Graph
153	Married couples whose fathers are brothers	Population	Big (70422787)	Big (77163109)	5	5	1	3	0	Cyclic	BB-Graph
154	Married couples whose mothers of mothers are sisters	Population	Big (70422787)	Big (77163109)	7	7	1	3	0	Cyclic	BB-Graph
155	Married couples whose fathers of fathers are brothers	Population	Big (70422787)	Big (77163109)	7	7	1	3	0	Cyclic	BB-Graph
156	Married couples whose state registers are different	Population	Big (70422787)	Big (77163109)	4	3	2	2	0	Path	DC
157	Sisters whose state registers are different	Population	Big (70422787)	Big (77163109)	5	4	2	3	0	Path	DC
158	Father and their sons along 8-degree-generation	Population	Big (70422787)	Big (77163109)	8	7	1	1	0	Path	BB-Graph
159	Mothers and their daughters along 8-degree-generation	Population	Big (70422787)	Big (77163109)	8	7	1	1	0	Path	BB-Graph
160	Twins who live in different flats of the same apartment which is different from the apartment	Population	Big (70422787)	Big (77163109)	9	14	4	6	0	Cyclic	BC
	where their parents live										
161	Persons live in same flat but their mothers are different	Population	Big (70422787)	Big (77163109)	5	4	2	2	0	Path	DC
162	Persons live in same flat but their fathers and mothers are different	Population	Big (70422787)	Big (77163109)	7	6	2	2	0	Others	EC
163	Sisters live in same flat but their father and mother lives in different flat	Population	Big (70422787)	Big (77163109)	6	8	2	2	0	Cyclic	DC
164	Twins live in same flat but their father and mother lives in different flat	Population	Big (70422787)	Big (77163109)	7	9	2	3	0	Cyclic	DC
165	Divorced couples who lives in same flat	Population	Big (70422787)	Big (77163109)	3	3	2	2	0	Cyclic	BC
166	Married couples born in same day	Population	Big (70422787)	Big (77163109)	3	3	2	2	0	Cyclic	BC
167	Daughters born in same day with her mother	Population	Big (70422787)	Big (77163109)	3	3	2	2	0	Cyclic	BC
168	Sons born in same day with her mother	Population	Big (70422787)	Big (77163109)	3	3	2	2	0	Cyclic	BC
169	Daughters born in same day with her father	Population	Big (70422787)	Big (77163109)	3	3	2	2	0	Cyclic	BC

Table A.9: Training Data for Determining Matching Order (9)

170	Mothers lives with her daughter from ex-husband in the same apartment	Population	Big (70422787)	Big (77163109)	6	7	3	4	0	Cyclic	DC
171	Mothers lives with her son from ex-husband in the same apartment	Population	Big (70422787)	Big (77163109)	6	7	3	4	0	Cyclic	DC
172	Fathers lives with her son from ex-wife in the same apartment	Population	Big (70422787)	Big (77163109)	6	7	3	4	0	Cyclic	DC
173	People born after 1995 and having at least one daughter as a mother	Population	Big (70422787)	Big (77163109)	3	2	2	2	1	Path	DC
174	People born after 1995 and having at least one son as a mother	Population	Big (70422787)	Big (77163109)	3	2	2	2	1	Path	DC
175	People born after 1995 and having at least one daughter as a father	Population	Big (70422787)	Big (77163109)	3	2	2	2	1	Path	DC
176	People born after 1995 and having at least one son and one daughter as a father	Population	Big (70422787)	Big (77163109)	4	3	2	2	1	Others	EC
177	People born after 1995 and having at least one son and one daughter as a mother	Population	Big (70422787)	Big (77163109)	4	3	2	2	1	Others	EC
178	Families with at least 3 children and all (mother + father + 3 children) living in the same address	Population	Big (70422787)	Big (77163109)	6	11	3	4	0	Cyclic	DC
179	Families with at least 4 children and all (mother + father + 4 children) living in the same address	Population	Big (70422787)	Big (77163109)	7	14	3	4	0	Cyclic	DC
180	People who has 2 ex-wife	Population	Big (70422787)	Big (77163109)	3	2	1	1	0	Path	BB-Graph
181	People who has 3 ex-wife	Population	Big (70422787)	Big (77163109)	4	3	1	1	0	Others	BB-Graph
182	People who has 2 ex-husband	Population	Big (70422787)	Big (77163109)	3	2	1	1	0	Path	BB-Graph
183	People who has 3 ex-husband	Population	Big (70422787)	Big (77163109)	4	3	1	1	0	Others	BB-Graph
184	Women born after 2000 and married	Population	Big (70422787)	Big (77163109)	3	2	2	2	1	Path	CC
185	Women born after 2000 and does not live in the same address with her father	Population	Big (70422787)	Big (77163109)	5	4	3	3	1	Others	CC
186	Men born after 2000 and does not live in the same address with his father	Population	Big (70422787)	Big (77163109)	5	4	3	3	1	Others	CC
187	Women born after 1995 and does not live in the same address with her father	Population	Big (70422787)	Big (77163109)	5	4	3	3	1	Others	EC
188	Men born after 1995 and does not live in the same address with his father	Population	Big (70422787)	Big (77163109)	5	4	3	3	1	Others	CC
189	Women born after 2000 and got divorced	Population	Big (70422787)	Big (77163109)	3	2	2	2	1	Path	DC

Table A.10: Training Data for Determining Matching Order (10)

190	Women born after 1995 and got divorced	Population	Big (70422787)	Big (77163109)	3	2	2	2	1	Path	DC
191	Men born after 2000 and got divorced	Population	Big (70422787)	Big (77163109)	3	2	2	2	1	Path	DC
192	Men born after 1995 and got divorced	Population	Big (70422787)	Big (77163109)	3	2	2	2	1	Path	DC
193	Women who lives with her mother and daughter	Population	Big (70422787)	Big (77163109)	7	8	3	3	0	Cyclic	DC
194	Women who lives with her father and son	Population	Big (70422787)	Big (77163109)	7	8	3	3	0	Cyclic	BC
195	Men who lives with his father and son	Population	Big (70422787)	Big (77163109)	7	8	3	3	0	Cyclic	DC
196	Men who lives with his father and daughter	Population	Big (70422787)	Big (77163109)	7	8	3	3	0	Cyclic	DC
197	Married couples with more than 70 years old	Population	Big (70422787)	Big (77163109)	3	3	2	2	1	Cyclic	BC
198	Women who is more than 40 years old and live her mother and father	Population	Big (70422787)	Big (77163109)	8	7	4	5	1	Cyclic	BC
199	Men who is more than 50 years old and live her mother and father	Population	Big (70422787)	Big (77163109)	8	7	4	5	1	Cyclic	DC
200	Women who is more than 50 years old and live her mother and father	Population	Big (70422787)	Big (77163109)	8	7	4	5	1	Cyclic	DC

A.2 Matching Order Test Dataset

Table A.11: Test Data for Determining Matching Order (1)

	Query	Dataset	Data Graph Node Size	Data Graph Edge Size	Query Graph Node Size	Query Graph Edge Size	# of Distinct Node Label	# of Distinct Edge Label	# of Nodes with Properties	Query Type	Best Results
1	Players who take role in any match at least 4 world cups	WorldCup	Small (45348)	Small (86577)	13	12	4	3	0	Others	EC
2	Players scores at least 4 goals as starters	WorldCup	Small (45348)	Small (86577)	6	5	3	2	0	Others	EC
3	Players in the squad of Brazil Team in 1998 World Cup	WorldCup	Small (45348)	Small (86577)	4	3	4	3	2	Others	CC
4	Players in the squad of England Team in 1966 World Cup	WorldCup	Small (45348)	Small (86577)	4	3	4	3	2	Others	CC
5	All World Cups hosted by Germany	WorldCup	Small (45348)	Small (86577)	2	1	2	1	1	Path	DC
6	All matches played in Daegu World Cup Stadium	WorldCup	Small (45348)	Small (86577)	2	1	2	1	1	Path	DC
7	Matches between Sweden and Brazil in 1994 World Cup	WorldCup	Small (45348)	Small (86577)	4	3	3	3	3	Others	CC
8	All matches of England in a World Cup that hosted by them	WorldCup	Small (45348)	Small (86577)	3	4	3	4	2	Cyclic	BC
9	Countries scores at least 3 goals as home team in any match at World Cup which is hosted by the same country	WorldCup	Small (45348)	Small (86577)	3	3	3	3	1	Cyclic	BC
10	All stadiums that Semi-Final Matches are played	WorldCup	Small (45348)	Small (86577)	2	1	2	1	1	Path	DC
11	All players takes role as substitute in any match that played in Estadio Azteca Stadium in any World Cup	WorldCup	Small (45348)	Small (86577)	5	4	5	4	1	Others	CC
12	All players takes role as starters and scores a hat trick in any match that played in Estadio Azteca Stadium	WorldCup	Small (45348)	Small (86577)	7	6	5	4	1	Others	EC
13	Players who is in the squad of Uruguay team that played at least two different World Cup	WorldCup	Small (45348)	Small (86577)	6	6	4	3	1	Cyclic	DC
14	Goals that Roberto Carlos scores in 2002 World Cup	WorldCup	Small (45348)	Small (86577)	5	5	5	5	2	Others	CC
15	All matches of Uruguay as an away team in 1998 World Cup	WorldCup	Small (45348)	Small (86577)	3	2	3	2	2	Path	DC
16	The user with id 11 who has a friendship with the user with id 61	Pokec	Medium (1632803)	Medium (30622564)	2	1	2	1	2	Path	DC
17	Friends play baseball	Pokec	Medium (1632803)	Medium (30622564)	3	3	2	3	2	Cyclic	BC
18	The people who can talk 'Russian' and 'vysokoskolske' education level that	Pokec	Medium (1632803)	Medium (30622564)	4	5	3	3	3	Cyclic	DC

Table A.12: Test Data for Determining Matching Order (2)

	Query	Dataset	Data Graph Node Size	Data Graph Edge Size	Query Graph Node Size	Query Graph Edge Size	# of Distinct Node Label	# of Distinct Edge Label	# of Nodes with Properties	Query Type	Best Results
19	The friends of users who used to basketball but now playing tennis	Pokec	Medium (1632803)	Medium (30622564)	4	3	2	3	2	Others	CC
20	The user list which consists of people who is 20 years old and 'stredoskolske' at education level	Pokec	Medium (1632803)	Medium (30622564)	3	2	3	2	2	Path	DC
21	Users talk English and play tennis and basketball	Pokec	Medium (1632803)	Medium (30622564)	4	3	3	3	3	Others	CC
22	Friends of friends of user with id 6	Pokec	Medium (1632803)	Medium (30622564)	3	2	1	1	1	Path	DC
23	Friends both talks Slovak and Italian	Pokec	Medium (1632803)	Medium (30622564)	3	2	2	2	2	Cyclic	BC
24	Female users play football and friend with 3 male users that play football	Pokec	Medium (1632803)	Medium (30622564)	7	14	2	2	3	Cyclic	DC
25	Users with cancer zodiac and 21 years old friends with scorpio zodiac and 21 years old	Pokec	Medium (1632803)	Medium (30622564)	6	5	2	2	2	Others	EC
26	Extended families consisting of mother, father, son, son's wife and children and all living in the same address	Population	Big (70422787)	Big (77163109)	7	10	2	4	0	Cyclic	DC
27	Brothers whose state registers are different	Population	Big (70422787)	Big (77163109)	5	4	2	3	0	Path	DC
28	Persons live in same flat but their fathers are different	Population	Big (70422787)	Big (77163109)	5	4	2	2	0	Path	DC
29	Brothers live in same flat but their father and mother lives in different flat	Population	Big (70422787)	Big (77163109)	6	8	2	2	0	Cyclic	DC
30	Daughters born in same day with her father	Population	Big (70422787)	Big (77163109)	3	3	2	2	0	Cyclic	BC
31	Fathers lives with her daughter from ex-wife in the same apartment	Population	Big (70422787)	Big (77163109)	6	7	3	4	0	Cyclic	DC
32	People born after 1995 and having at least one son as a father	Population	Big (70422787)	Big (77163109)	3	2	2	2	1	Path	DC
33	Families with at least 5 children and all (mother + father + 5 children) living in the same address	Population	Big (70422787)	Big (77163109)	8	17	3	4	0	Cyclic	DC
34	People who has 3 ex-husband	Population	Big (70422787)	Big (77163109)	4	3	1	1	0	Others	BB-Graph
35	Men born after 2000 and married	Population	Big (70422787)	Big (77163109)	3	2	2	2	1	Path	DC
36	Men born after 1997 and got divorced	Population	Big (70422787)	Big (77163109)	3	2	2	2	1	Path	DC
37	Men born after 1997 and does not live in the same address with his father	Population	Big (70422787)	Big (77163109)	5	4	3	3	1	Others	CC
38	Men who lives with his mother and daughter	Population	Big (70422787)	Big (77163109)	7	8	3	3	0	Cyclic	DC
39	Married couples with more than 65 years old	Population	Big (70422787)	Big (77163109)	3	3	2	2	1	Cyclic	BC
40	Men who is more than 40 years old and live her mother and father	Population	Big (70422787)	Big (77163109)	8	7	4	5	1	Cyclic	DC

A.3 Query Results of Subgraph Isomorpishm Algorithms on WorldCup Database

Table A.13: Query Results of Subgraph Isomorpishm Algorithms on WorldCup Database (Part-1)

	Cypher	**DualIso**	**GraphQL**	**TurboIso**	**VF3**	**BB-Graph**	**BB-Plus (Historical)**	**BB-Plus (Real-Time)**
Query 1	1052	10453	24747	5116	6337	1050	602	874
Query 2	10123	34675		6361		3353	3231	3124
Query 3							3092	3260
Query 4	26711	26311		22072	78543	11193	10850	3939
Query 5			10654	113714		23754	18121	18224
Query 6			15243	132842		25743	21245	21382
Query 7	15234	30324		8746		4324	3874	3764
Query 8	12245	28465		7543		4121	3645	3521
Query 9	12345	27364	16237	7923	12432	3421	3127	3317
Query 10	12963	37852		8754		3682	3495	3374
Query 11	34950	36475		29845	87421	12875	12123	10874
Query 12	6745	29745	13425			3421	2983	2869
Query 13	7213	30818	14759			3874	3145	3111
Query 14	7126	30485	14743			3842	3214	3222
Query 15	6323	29845	13543			3485	2964	3097
Query 16	6983	28453	13845			3742	3125	3098
Query 17	6853	28745	13954			3632	3245	4532
Query 18	7342	31938	14756			3975	3423	3398
Query 19	7263	31634	14367			3795	3214	3211
Query 20	7163	31641	13226			3887	3874	3812
Query 21	8932	33421	17432	45632	42386	5436	4932	4873
Query 22	9321	32873	17293	39563	45684	5983	5821	6873
Query 23	9123	38742	18734	41937	43857	5743	5674	5678
Query 24	8674	35949	16384	46327	48932	5274	4984	4976
Query 25	8372	34632	18353	51345	48235	5129	4564	4485
Query 26	8234	38938	20983	48732	45973	5243	4673	4569
Query 27	8268	37945	21874	65321	55328	5283	4988	4925
Query 28	7921	29834	18274	45684	51423	5198	4873	4912
Query 29	9834	31736	17365	58732	68234	5291	5189	5211

Table A.14: Query Results of Subgraph Isomorpishm Algorithms on WorldCup Database (Part-2)

	Cypher	**DualIso**	**GraphQL**	**TurboIso**	**VF3**	**BB-Graph**	**BB-Plus (Historical)**	**BB-Plus (Real-Time)**
Query 30	6431	21467	11235	18365	21567	4539	4128	3875
Query 31	6648	22747	12398	20943	22774	4673	4576	4597
Query 32	5983	19834	11723	19823	20983	4555	4432	4245
Query 33	6436	21745	11986	18373	19834	4984	4698	4356
Query 34	7121	20872	11764	12764	13873	4763	4768	4759
Query 35	9746	34867	10883			7531	6537	6973
Query 36	10223	38745	12456			7432	7425	7399
Query 37	9654	37659	11345			7683	7643	7598
Query 38	9879	32746	10763			7573	7523	7511
Query 39	9435	40875	12743			7442	7228	7246
Query 40	21712	19374		38734	52495	15367	14850	14249
Query 41	20974	25748		37645	51828	14324	14214	14356
Query 42	22875	27648		41983	39857	16437	16246	16298
Query 43	21857	18736		40872	45761	15384	15224	15325
Query 44	32185	356793				22395	22390	22384
Query 45	30987	29845				21874	21876	2143
Query 46	35784	41983				22888	22654	22756
Query 47	35873	42985				23567	23123	23432
Query 48	32865	37646				19284	18723	18992
Query 49	7348	34867	63259	69383		6246	6128	6221
Query 50	7974	29874	59832	51083		6117	6001	6005