An Exploration of Graph Algorithms and Graph Databases

Phininder Balaghan BSc (Hons)

A thesis presented for the degree of

Doctor of Philosophy

University of Hull

May 2019

A best friend is the only one
that walks into your life when
the world has walked out.

Shannon L. Alder

Dedication

I dedicate this thesis to my Mum, Dad and my Sister. I know how much you have sacrificed to make me the person I am today. Thank you. Your continued guidance and love drives me forwards.

To my main supervisor, Ken Hawick. I couldn't have asked for a better supervisor. Your guidance and friendship throughout my journey has been invaluable. Thank you for going above and beyond, and educating me to the world. I would like to extend my gratitude to my co-supervisor, Neil Gordon. You've been on my side since my first day of undergraduate studies, and have helped me every step of the way. And finally thank you to Mike Brayshaw for supporting me as well as chairing my panel meetings.

In addition, I would like to say thank you to the many administration and technical staff at the University of Hull for their help over the years, in particular

Helen El-Sharkawy, Lynn Morrell, Sally Byford, Jo Clappison, Mike Bielby, Mark Bell, Adam Hird and Andrew Hancock. Thank you to all of the various academics that I've spoken to that have also given me guidance in this PhD, particularly David Chalupa, James Walker, Kevin Elner and Martin Walker. I would like to extend my thanks to my internal examiner, Alexander Turner and my external examiner, Omer Rana for their additional guidance for improvements to the thesis.

To all members of the Computational Science Research Group, past and present, and in particular; Anth Quinn, Craig Maddra, Liam Stockwell, Daniel Fleming, Russell Billingsley, Mariusz Kosmatka, George Lacey, Andrew Johnson and Lee Odiam. Thank you all for pushing me and also for keeping me sane.

Thank you also to my friends outside of university who have also supported me, especially; Abdullah-isa Amole, Danny Alan, Bhavin Mistry, Seven Sawmynaden, Sarah Nevins, Raksha Aggarwal, Sam Cooke, Hina Patel and all of my other many great friends. I'm very lucky to have you all by my side and supporting me through this long journey. In addition I would also like to thank Nick Fletcher and John Murray.

Finally, Bronya Holliday. I never would have made it this far if it wasn't for your constant support, love, encouragement and kindness that you've given me throughout my PhD. You've supported me throughout this PhD and especially in the tough times you have been my rock, thank you.

For Beji, Priya Didi and Babaji, I hope you are resting wherever you are.

Phini

Abstract

With data becoming larger in quantity, the need for complex, efficient algorithms to solve computationally complex problems has become greater. In this thesis we evaluate a selection of graph algorithms; we provide a novel algorithm for solving and approximating the Longest Simple Cycle problem, as well as providing novel implementations of other graph algorithms in graph database systems.

The first area of exploration is finding the Longest Simple Cycle in a graph problem. We propose two methods of finding the longest simple cycle. The first method is an exact approach based on a flow-based Integer Linear Program. The second is a multi-start local search heuristic which uses a simple depth-first search as a basis for a cycle, and improves this with four perturbation operators.

Secondly, we focus on implementing the Minimum Dominating Set problem into graph database systems. An unoptimised greedy heuristic solution to the Minimum Dominating Set problem is implemented into a client-server system using a declarative query language, an embedded database system using an imperative query language and a high level language as a direct comparison. The performance of the graph back-end on the database systems is evaluated. The language expressiveness of the query languages is also explored. We identify limitations of the query methods of the database system, and propose a function that increases the functionality of the queries.

We further evaluate graph database systems by implementing computationally complex problems such as the Graph Diameter, Betweenness Centrality and the Component Labelling problems. These all have algorithms, or heuristics that provide consistent results, which is in contrast to the Minimum Dominating Set and the Longest Simple Cycle. We provide novel implementations that are applied into client-server, embedded and high level languages in the same manner as the dominating set. We evaluate query methods, their expressiveness and propose functions that could be used to improve these.

Contents

List of Figures

List of Tables

Rancho: That day I understood
that this heart scares easily.
You have to trick it, however
big the problem is. Tell your
heart, 'aal izz well, aal izz well.'
Raju: Does that solve the
problem?
Rancho: No, but you gain
courage to face it.

3 idiots

CHAPTER 1

Introduction

The definition of big data has altered in recent times. The sheer amount of data gathered has increased exponentially. So much so, that previous storage mechanisms have struggled to cope with the size and the complexity of the data. Thus finding efficient storage mechanisms of *graph-like data* has become a pressing issue. Once the data has been stored, there must be an ability to efficiently analyse and explore the data with algorithms.

Data is being collected at a greater rate than ever before. It is estimated that by 2020 1.7 megabytes of new information will be created every second for every human on the planet (Marr, 2015). Much of this data will be unstructured and complex. Storage mechanisms that rely on strictly schema data, such as relational databases,

have found issues arising with both the size and the complexity of this type of data.

A new type of storage mechanism was created, the birth of Not Only SQL (NOSQL) databases. These new types of databases have fundamentally different engines which allow them to efficiently store and query data in ways that relational databases struggled with. A database family that has recently gained traction uses a graph engine as its core. Graph Databases has been made to specifically target weaknesses found in relational databases.

1.1 Graph databases

In the database families within NOSQL, graph databases stand out. Rather than using documents or tabular like data structures to store data, the underlying engine is a graph. This graph structure allows for a focus on relationships between data nodes. Graph databases have adopted a different kind of graph model; the property graph model. This can be simply defined as:

$$G = (N, E, P) \tag{1.1}$$

Where G = graph, N is a non-empty set of nodes that exist in graph G, and E is a non-empty set of edges limited to NxN, and P is a key-value list of properties which are attached to every node and edge in the graph. Every node and edge in the graph will have a unique property list assigned to it. An example of a graph formatted with the property model can be found in fig. 1.1.

As shown in fig. 1.1, each of the nodes and edges have unique property lists.

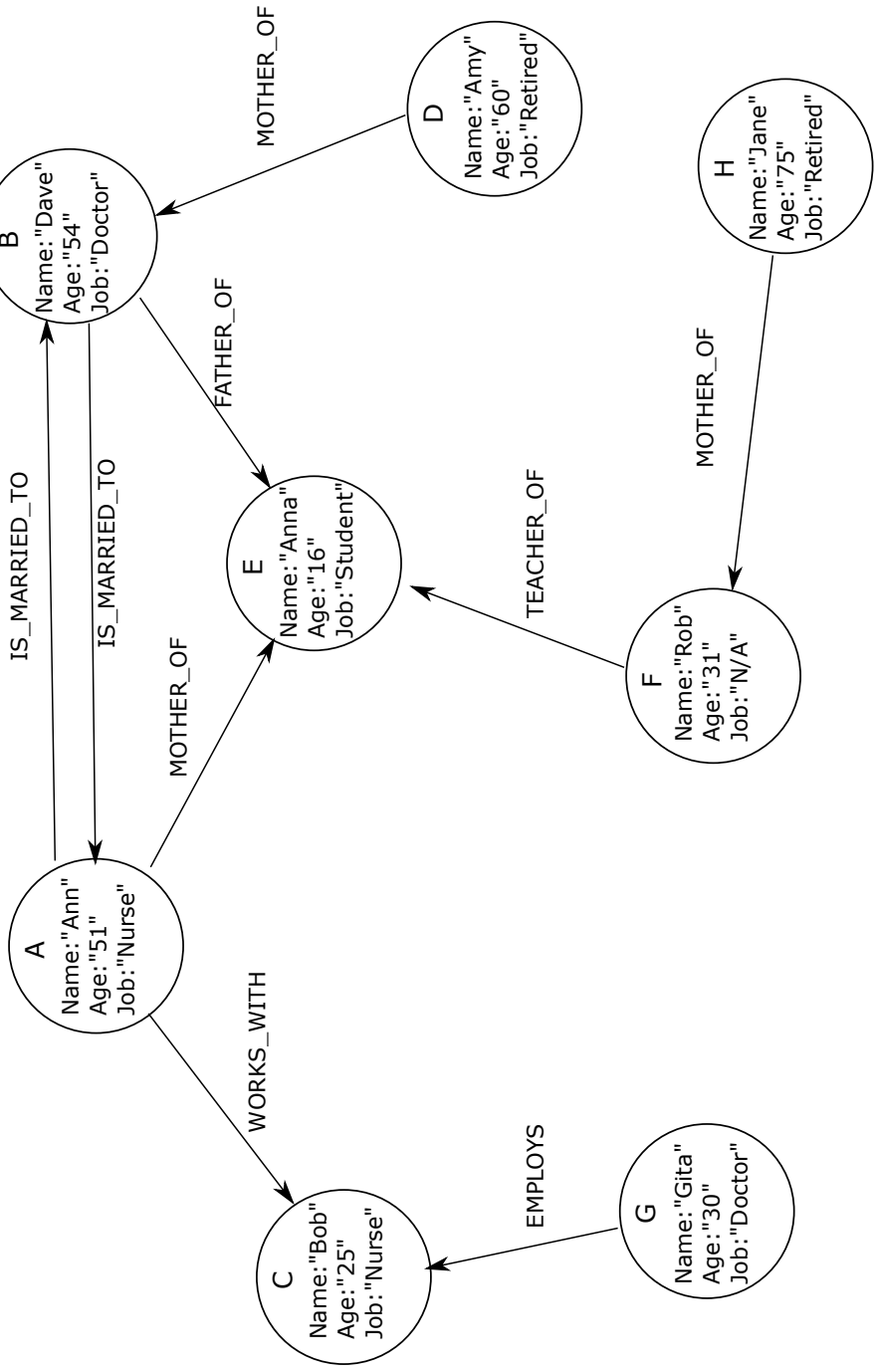

Figure 1.1: A basic example of a graph database. Each of the nodes and edges in the graph have unique property lists. These property lists can contain values which are unique to the node or edge, allowing additional information to be stored.

Graph database queries can take advantage of the uniqueness of these lists. Further exploration into graph databases and their multiple uses can be found in section 2.1. An advantage to the graph based underlying engine is that it has the ability to implement complex graph algorithms.

1.2 Graph algorithms and complexity

Graph theory famously began with the seven bridges of Königsberg problem (Euler, 1736). From there, graph theory as a field has created a vast amount of algorithms.

These algorithms tend to have a varying amount of complexity. They fall into a selection of complexity classes, with the three most well known as Polynomial (P), Non-deterministic Polynomial-time Hardness (NP-Hard) and Non-deterministic Polynomial-time Completeness (NP-Complete). In this thesis, we explore a variety of problems which fall into these classifications. It is worth noting that there are many other classifications of complexity that are not investigated here.

We firstly focus on the Longest Simple Cycle problem. This problem finds the longest simple cycle in an undirected graph. A simple cycle can be defined as a cycle of minimum length 2 without any repetition of nodes or edges apart from the initial node. The longest simple cycle problem is a generalisation of the Hamiltonian cycle problem. Because of this, it is a classical NP-Hard problem. The longest cycle problem is further explored in section 2.4.7 and in chapter 3.

We then explore the minimum dominating set problem. A dominating set in a graph is a subset of nodes such that every node in the graph is either in the

dominating set or is adjacent to a node in the dominating set. It is classified as a NP-Complete problem. The most well known heuristic is the greedy algorithm. The problem is further explored in section 2.4.5 and in chapter 4.

The next algorithm that is explored is the shortest path problem, its variations and some algorithms which use it as a base for further operations. The shortest path algorithm is a well known P problem. A variation of this is the all-shortest paths problem, which essentially finds all of the possible shortest paths in a graph. These are further analysed in section 2.4.2. The graph diameter is the longest shortest path, and can be found using the all shortest paths algorithm. This is explored in section 2.4.3. The betweenness centrality of a graph ranks each of the nodes by its importance within shortest paths. These are explored in section 2.4.4.

Finally, the Component Labelling problem finds all of the different components within a graph. Each of the components found are given unique labels. This falls in the P complexity area, and is explored in section 2.4.6. The shortest path, all shortest paths graph diameter, betweenness centrality and the component labelling problems are all also implemented in a variety of ways in chapter 5.

The variety of graph algorithms explored in this thesis give an insight into different graph problems. The Graph Diameter and Betweenness Centrality both use a shortest path algorithm as a base. Betweenness Centrality is a node ranking problem and the Graph Diameter is a distance calculation problem. The Minimum Dominating Set problem is a classical decision problem while the Longest Simple Cycle is a generalisation of the Hamiltonian Cycle problem. Finally, the component labelling problem is a region detecting problem. These problems provide a unique

graph challenge for the underlying engines explored in this thesis.

1.3 Applying graph theory to graph databases

Graph databases and graph theory go hand in hand. The uses of complex algorithms from the well studied theory area can be applied and implemented into databases. Graph databases can support graphs from a multitude of disciplines, so an algorithm that can be written for a generic graph in a graph database can be easily applied to different graph types.

We can evaluate how efficiently the graph database engines are by implementing the algorithms of varying complexity. The query languages or libraries provided by the graph database system will be used. This allows an evaluation of a number of items; the graph engines, the given method of extracting data and the combination of both.

After we implement the algorithms into graph databases using the query methods provided by them, we compare performance timings with a high level language implementing of the said algorithms.

The advantage of being able to apply these algorithms directly into a graph database allows for more dynamic-ism and better resource management. For example, a graph database could hold a a snapshot of an scenario due to the persistent storage mechanism that can only be uniquely provided by a graph database. The scenario could be dynamic and ever-changing, thus the graph held in the graph database may be constantly changing, with nodes and edges being removed and added. Taking

a copy of the graph out of the database into memory not only means that the algorithms could be running on an out-of-date snapshot of the scenario, the amount of extra resources required to be able to do this may not make it viable.

1.4 Problem statement

By their very nature, NP-Hard classified problems cannot be solved efficiently. These problems typically have an approximation or a heuristic algorithm in order to find a close to optimum solution. Heuristical algorithms can provide interesting information even if not giving an exact solution. Graph databases provide a new engine that can be used to run such algorithms on many different types of instances.

This thesis explores the Longest Simple Cycle problem, which is an NP-Hard problem, in depth. A heuristic for a different NP-Hard problem is then implemented into graph databases which allows the underlying engine(s) to be evaluated for performance and language expressibility. Finally, other algorithms which are known to be classified as P to NP-Complete are also implemented into graph databases to further explore the underlying engines.

Due to the nature of the thesis, several hypotheses can be found.

- Can the underlying engines of graph databases efficiently handle complicated graph algorithms?
- How can these algorithms be expressed in current graph database languages?
- How efficient are the built in functions of the systems?

- Can a more efficient exact solution be found for the Longest Simple Cycle problem?
  - If not, can a heuristic be produced?

1.5 Contributions found in this thesis

There are a number of contributions within this thesis, which directly correlate with the hypotheses given in 1.4. The contributions are:

- Computational methods of solving the longest simple cycle problem
  - A novel exact algorithm using an Integer Linear Programming (ILP) has been created to find the longest simple cycle in a graph
  - A Depth-First Search (DFS) based heuristical algorithm which uses perturbation operators to improve a long cycle has been created to find the longest simple cycle in a graph
- The Minimum Dominating Set problem has been explored in detail when implemented into graph databases.
  - A novel implementation of the Minimum Dominating Set problem in graph database systems
  - The efficiency of the underlying engines have been explored
  - The expressibility of the query languages when applied to this problem are also evaluated

- Graph databases are then further evaluated when other problems that are classified from P and NP-Complete are implemented with novel solutions, such as:
  - Betweenness Centrality
  - Graph Diameter
  - Component Labelling
  - The efficiency of in built functions, such as the shortest path, of the systems are evaluated.

A novel flow-based exact ILP has been created to solve the longest simple cycle in a graph. We compared this to an existing ILP formulation. In addition to this, we have created a novel DFS based heuristic algorithm. We compare the results found in all three problem implementations.

We then implement the Minimum Dominating Set algorithm into graph databases using query methods. We evaluated the expressiveness of the query languages, as well as the performance of said algorithms by comparing them with a high level language implementation.

We then further evaluate the graph database engines by implementing other highly computational algorithms. We compare these with a high level language implementation, as well as the general expressiveness of the query methods.

In the above implementations, we used a set of instances from a wide variety of sources, ranging from biological networks to social media networks. All instances were implemented as undirected graphs, with no self loops. Through these results,

we have identified limitations to the query methods provided by the graph databases, and propose improvements to the query methods.

1.6 Structure of this thesis

The thesis can be read in a linear manner. However each chapter has been written in a style such that they can be read independently. As such, some sections may include repetition. These sections have been clearly labelled, and referenced where repetition may have occurred.

Chapter 2 gives an in-depth review of the various graph algorithms and database literature required for the oncoming chapters.

In chapter 3 the Longest Simple Cycle problem is explored in depth. An exact algorithm for finding the longest simple cycle in a graph using an ILP formulation has been created and compared to existing methods. A heuristic to find the longest simple cycle based on an initial DFS cycle with improvements due to additional perturbation operators has also been created.

Chapter 4 explores the Minimum Dominating Set problem, and evaluates how efficiently the algorithm can be implemented into graph databases. The limitations and efficiency of the implementations are explored and discussed in detail.

Following from the Minimum Dominating Set problem, other computationally complex problems such as the Betweenness Centrality, Graph Diameter and Component Labelling are implemented into graph databases for further performance and efficient evaluation in chapter 5.

In the final chapter, we conclude this thesis. We also discuss long cycles in graph database systems. This can be found in chapter 6.

The greatest education in the world is watching the masters at work.

Michael Jackson

CHAPTER 2

Literature Review

The need for storing graph-like data has increased in recent times (Yannakakis, 1990). However, traditional, conventional database storage mechanisms have found this difficult. Relational databases are one such database type. At the time of writing, relational databases are the most popular databases (DBEngine, 2017b). The relational model consists of tables, with data stored in a row (Codd, 1970).

The figure in 2.1 shows an example of a relational database model. The relational model requires the different entities of the data to be split into separate tables. Within these tables are many columns that can represent attributes of these entities. This gives the benefit of having a strong schema on the data, allowing the database manager to constrain the data allowed into the database. In the figure, the data has

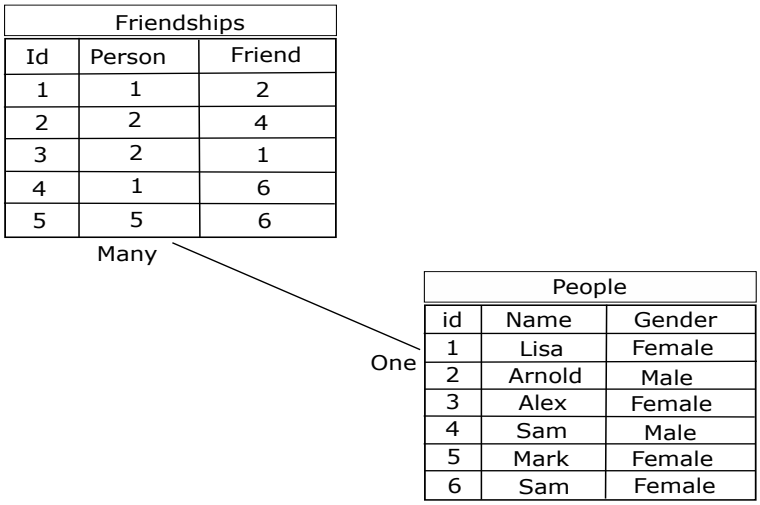

Figure 2.1: A social media example using the relational database model. The relational model requires the data to be structured into multiple tables that are different entities within the data. In this example, the people and the friendships of the people are separated into two tables. Within these tables, the columns may have a "Key" column. This is unique to that table. For the people and friendships table, the key component is the ID of the person. The data can be joined together by using one of three relations. The "One-to-One", "One-to-Many" and "Many-to-Many". The "One-to-Many" is shown above. The "One" represents a key value. There can only ever be "One" person, however that person could have "Many" relationships (Data, 1975).

been split into two separate entities to signify the people, and the friendships the people have in relation to each other. The "Key" column is a unique field. The key field for both the friendships and the people are the ID, as there can only be one unique person. The two entities can then be joined together by three different types of relationships. The "One-to-One", the "One-to-Many" and the "Many-to-Many". An example of a "One-to-Many" is given in the figure (Data, 1975; Howe et al., 2013).

However, there are many downfalls to using a relational database. Due to the strict schema that is required by relational databases, data which requires a less restrictive schema is very difficult to store within a relational database. Using the example in figure 2.1, if we wanted to know the age of the people, we would have to create a new column within the table. This opens up the opportunity for missing data as not all age data may be known.

This was not the only restriction. Scaling a relational database to be able to hold a significant amount of data is very difficult to do. Comparing the example to a real-life social media, we would hope to have more information than just the age of the person. The amount of data would increase, and the size of the persons tables would also dramatically increase, thus effecting the querying time.

Because of these restrictions, Not Only SQL (NOSQL) was introduced. NOSQL databases do not claim that relational databases are redundant, in fact there are times when it is recommended to use relational databases for certain types of data. Instead, NOSQL databases are an alternative to relational databases when limitations of relational databases are found.

Within NOSQL, there are a number of different database "families". These families include:

- Graph database
- Document database
- Key-Value Store database
- Object database

- Tabular database
- Triple/Quad Store database
- Multi-Modal database

Most of these different NOSQL families have been reviewed in (McColl et al., 2014; Corbellini et al., 2017). One of the families which are interesting is graph databases.

2.1 Review of graph databases

Storing graph data in relational databases heavily relies on computationally intensive operations. This limitation has been recognised, particularly in social media platforms. Some social media platforms such as Facebook created their own graph storage methods (Bronson et al., 2013). Tools for computation of graph-like data are also required (Hawick, 2007).

Definition 2.1.1. A **database** is a collection of related data. **Data** is facts that are recorded and have implicit meanings (Elmasri and Navathe, 2010). In this thesis, *data* is used as both a singular and plural definition, as opposed to datum as its singular, as it is common in database literature for it to be referred as this.

Definition 2.1.2. A **database management system (DBMS)** is a collection of programs which enable users to create and maintain a database. A **transaction** represents a piece of work occurring in a database. A transaction typically describes any change in a DBMS. Transactions can follow the four rules of Acidity.

Definition 2.1.3. ACID is a collection of rules that all transactions in a database must follow. Atomicity, Consistency, Isolation and Durability are all properites that must be followed in a all-or-nothing manner (Gray, 1981; Gray and Reuter, 1992). The four properties are explored in the definitions below.

Definition 2.1.4. Atomicity is a property of ACID. An atomic transaction is one where a series of database operations must occur, otherwise the database rolls back. An **operation** is a single process that can occur in a query. For example, an exchange of a single item between two people involves two operations. The removal of item from the first person is a single operations, and an addition of the item to the second person is the second operation. An atomic transaction either follows both the operation through, or none of it is carried out, thus preventing any loss of data.

Definition 2.1.5. Another property of ACID is **Consistency**. This gives the requirement that data can only be changed as according to any constraints or rules imposed to the data, such as a schema.

Definition 2.1.6. Isolation is a transaction processing manager that defines when one process has finished, and when the next begins. It essentially manages the concurrency effects of transactioned data.

Definition 2.1.7. The final property of ACID is **Durability**. This guarantees that all transactions that have occurred will survive permanently.

Definition 2.1.8. A **database schema** is a specific structure described in a formal language. This schema is a constraint on data stored.The schema will needto be supported by the DBMS.

In order to store and query graph data efficiently, it became clear that a new type of database was required. Thus, graph databases were born. They are a part of the NOSQL family.

Graph databases have been used in a wide variety of research, such as in analyzing voting and donation networks in Brazilian politics (Bursztyn et al., 2016), analysing bibliographic data (Zhu and Yan, 2016), systems engineering (Harrison, 2016), prevenance segmentation (Abreu et al., 2016) and much more.

Graph databases built upon some of the research into graph models in the 1990's. Many different unique graph models were created. These graph models have formed the basis of different modern day models. For example, HypergraphDB has based its model off the Hypergraph model created by Levene in the 1990s (Levene and Poulovassilis, 1990; Levene and Poulovanssilis, 1991). As well as this, other models such as GOOD (Gyssens et al., 1994), Gram (Amann and Scholl, 1992), GraphDB (Güting, 1994), PaMaL (Gemis and Paredaens, 1993), GOAL (Hidders and Paredaens, 1994) and LDM (Kuper and Vardi, 1993) were created.

In addition, an extensive survey of these and other models was completed by Angles and Gutierrez (2008).

Around the millennium, there was a dip in research into graph storage models. This was most likely due to the memory restrictions of hardware that was available at the time, as well as the prominent rise of relational databases. With the rise of processing power and general increase in memory available in machines as standard, graph storage and graph computation has become more prominent (Robinson et al., 2013).

Graph databases typically are fully transactional and maintain Atomicity, Consistency, Isolation and Durability (ACID). ACID is a set of properties of database transactions that guarantee the validity of a transaction, regardless of an error or power failure. If an error or power failure occurs, the database will revert to its previous form (Haerder and Reuter, 1983; Gray, 1981).

According to the database ranking site DBEngine (2017c), since 2013, graph databases have seen the highest rise in popularity .

This rise in popularity was due to a number of factors. One of which is that graph databases have focused on storing data by using the structure of a graph. It allows the user to focus on the relationships between the data. Graph databases tend to have a few defining characteristics. One such defining characteristic is the ability to store data free of schema. This is one of the main differences in comparison to Relational and, in some circumstances, other NOSQL databases as they require a strict schema. Another characteristic is the focus of the relationship between the data in the graphs. This can expand through different data types, as explained in section 2.1.1.

Graphs can be created in many different database types. In fig 2.2 it shows how to create a graph in different database types such as relational, XML and others. However it can be argued that one of the unique features that make graph databases different from other database types is the ability to create an explicit graph without relying on indexes (Rodriguez and Neubauer, 2010).

2.1.1 The Property Graph Model

For this section, a *graph* is referred to as a property graph. Fig. 2.2 shows an example of a simple property graph. The example shows a basic social media scenario, with the nodes representing different people, and the edges representing the relationships between the people. The properties are a list of "Key-Pairs" which are attached to the nodes and edges. In the example, the property list for the nodes add more information about the person, and for the edges they define the type of relationship between the people. The schema-free feature of the graph can be seen in the property list for both the edges and nodes. For example, the edges each have their own defining property such as Beth being a *mother to*, *a friend to* and *is married to* three different people.

Definition 2.1.9. A simple definition of the **Property Graph model** consist of Nodes (N), Edges (E) and Properties (λ) such that $G = (N, E, \lambda_N, \lambda_E)$. Properties are a list of key-value pairs which are attached to nodes ($\lambda_N \cup N$) and edges ($\lambda_E \cup E$). Every node and edge in the graph have their own unique property list. In general, the property graph model is a *directed graph* that allows self-loops.

It is worth noting that the property graph model can be tweaked in order to suit a different need for a graph database system. For example, HyperGraphDB uses a *directed hypergraph* at it's core (Iordanov, 2010) .

Another definition of the Property Graph Model

The property graph model has been defined in multiple ways. A defintion was provided in section 2.1.1, however this is quite simplistic. A much more detailed

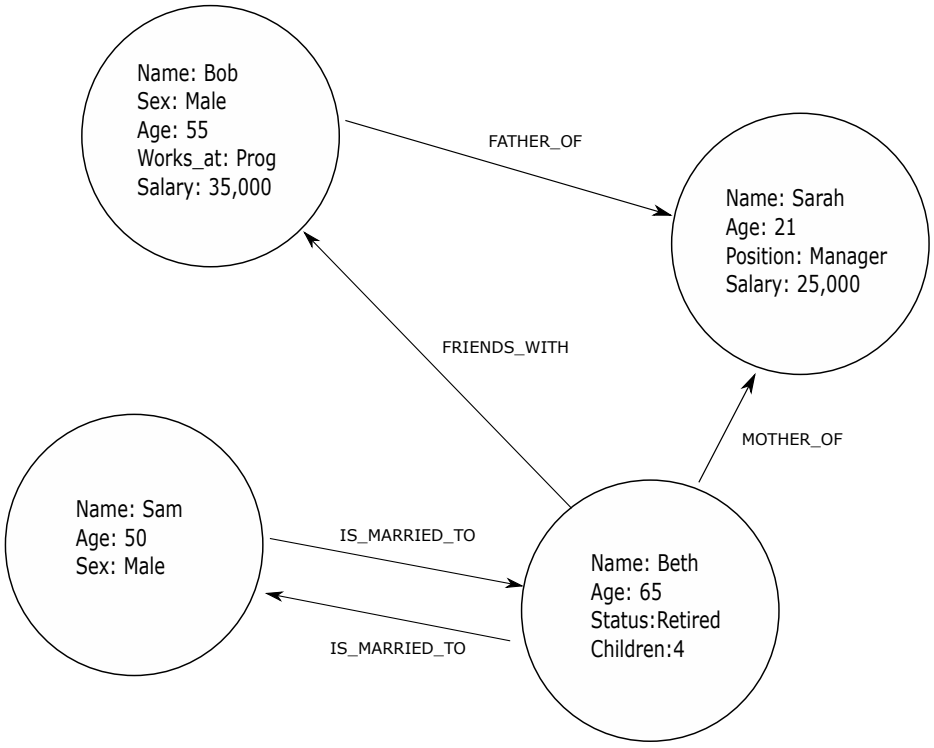

Figure 2.2: A property graph modelled in an social media scenario. As defined in section 2.1.1, the property graph model encapsulates nodes, edges and properties. The nodes represent different people and the edges represent the relations between the people. Both the nodes and edges contain properties. Properties are generally a list of "Key-Pairs" which are associated with each node and edge. In the example, the properties represent additional data about the people in the nodes and the type of relationships in the edges. All the nodes represent different people, and because of the property graph's schema free feature, we are able to store different information about the people.

definition was given by Tomaszuk (2016), who defines the property graph model as $PG = \langle V, E, S, P, h_e, t_e, l_v, l_e, p_v, p_e \rangle$ where:

1. V is a non-empty set of nodes
2. E is a set of edges
3. S is a set of strings
4. P contains each property that has a form $p = \langle k, v \rangle$ where $k \in S$ and $v \in S$
5. $h_e : E \to V$ is a function which yields the source of each edge (head)
6. $t_e : E \to V$ is a function which yields the target of each edge (tail)
7. $l_v : V \to S$ is a function mapping each node to a label
8. $l_e : E \to S$ is a function mapping each edge to a label
9. $p_v : V \to 2^p$ is a function used to assign nodes to their multiple properties
10. $p_e : E \to 2^p$ is a function used to assign edges to their multiple properties

It is quite clear to see how many overheads a graph database has from the definition in (Tomaszuk, 2016). There are multiple functions required which assign properties to their respective nodes and edges. For example, item 9 in the above list shows how a function is used to map properties to each individual node in a graph.

Why use the Property Graph model?

The schema free feature allows the property graph to model other graph models. For example, should the data require weighted graphs, you may add the weights to the edges of the graph by setting the properties of the edge to represent weights. Figure 2.3 shows how the property graph model can be transformed into other graph models (Rodriguez, 2010).

The ability to transform a graph model into a different model allows the freedom to adapt and change the database to meet its data requirements. For example, a simple undirected graph can be mimicked on a graph database by removing properties and creating two directed edges between every pair of nodes, as shown in fig 2.3.

2.1.2 Types of graph databases

Graph databases can currently be typically sorted into two different types of databases. The first of which is a *client-server* database. The second is an *embedded* database.

Client-Server Databases

Definition 2.1.10. A **client-server** Database Management System (DBMS) allows clients to send query requests to a server. Clients can be external or internal to the machine where the server is stored. A common relational database that uses this model is MySQL.

Client-server databases either take advantage of a query language, or they provide libraries that use the query language to query the database engine.

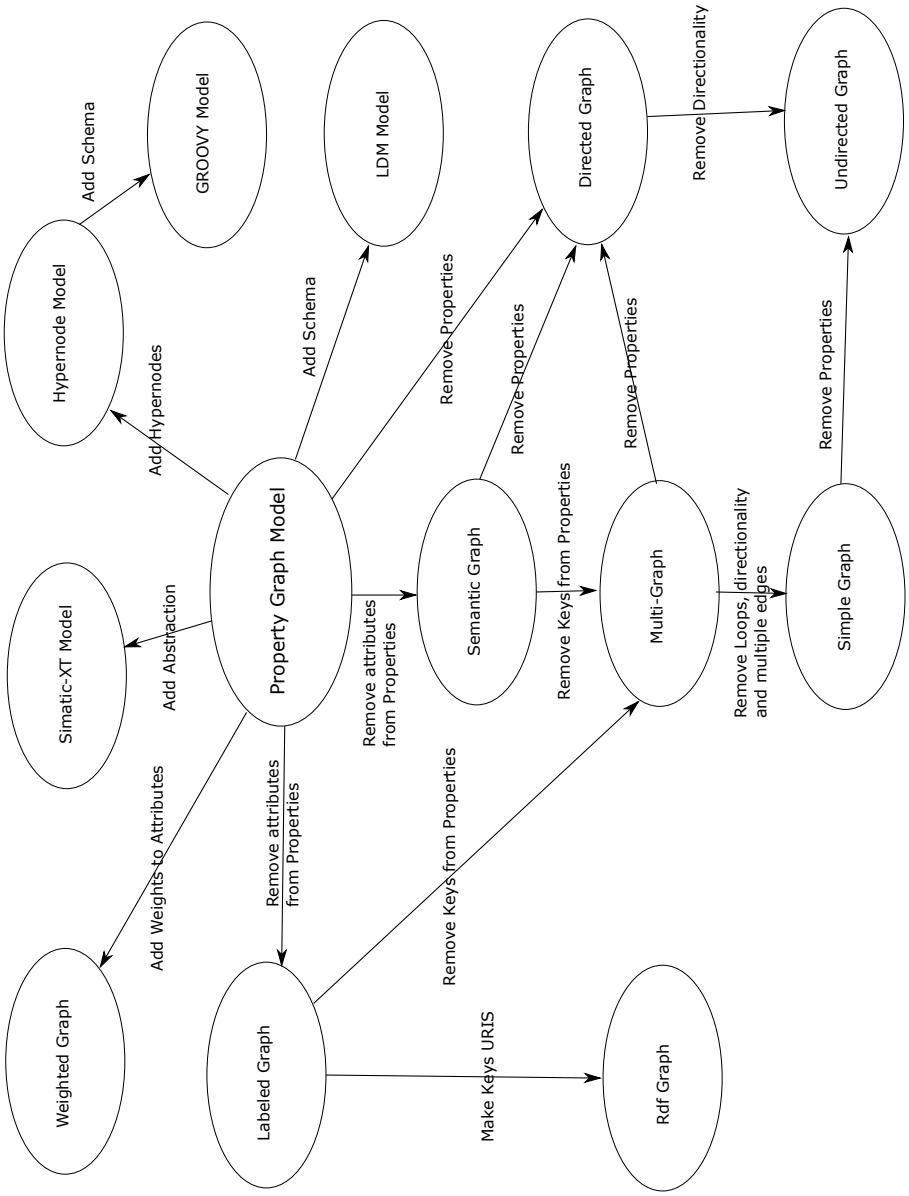

Figure 2.3: A graph to show how the property graph can be modified and changed into other graph models. Adapted from Rodriguez (2010).

At the time of writing, Neo4j is the most popular graph database vendor (DBEngine, 2017b,d). Neo4j is a Java Virtual Machine (JVM) based graph database. It uses a JVM server to process the queries sent to it either through its own query language Cypher, or through various APIs that can be integrated with a selection of high level languages (Neo4j, 2018). Miller (2013) provides an introduction to Neo4j. Neo4j is an active advocate of the property graph model. It is worth noting that Neo4j redefines the term "edges" with the term "relationship" (Neo4j, 2018).

Neo4j is a fully transactional database that supports ACID functionality.

As mentioned before, Neo4j has a unique query language called Cypher. Cypher is a declarative language, and has been described as "expressive" especially when compared to other graph query languages (Rath et al., 2012). Cypher is explored in greater detail in sub-section 2.2.1.

Embedded Database

Definition 2.1.11. An **embedded** DBMS allow the user to embed a database into their own high level language program. A similar relational example can be *SQLite*.

The second model is an embedded model. The database is created and stored locally, typically through library calls in a high-level language. An example system of which would be Sparsity. Sparsity is a system written in c and c++.

Embedded databases do not typically have or use a query language, instead they provide library calls which act as a query language. Embedded databases are stored local to a program.

As such, only an imperative language can be used to query an embedded database.

At the time of writing, the most popular embedded graph database is Sparsity (DBEngine, 2017d).

Sparsity is one of the more popular embedded graph databases. It began life as DEX (Martínez-Bazan et al., 2007) where it then re-branded as Sparsity. It does not currently have a dedicated query language, instead the database can be queried by using some of the in-built library function when combined with a high-level language. It also fully supports ACID transactions. One special feature of sparsity is the ability to disable ACID transactions, should they not be required.

2.1.3 Graph database popularity

At the time of writing, graph databases have seen a rise in popularity (DBEngine, 2017b). In fact, in recent years graph database usage has increased by over 500% (DBEngine, 2017a).

This is due to a number of factors. More consumers are realising the limitations of relational databases. The data used by the consumer is very complex, can be modelled in a graph-like manner (Yannakakis, 1990) and requires a less restrictive schema. In contrast to relational databases, graph databases are schema free. This allows any type of data to be stored within the database. Coupled with the property graph model, this gives scope to store any type of data within a graph.

Some major companies have seen this, with the likes of Google (Bronson et al., 2013; Malewicz et al., 2010), Facebook (Bronson et al., 2013) and Microsoft (Shao et al., 2013). Facebook, in particular, stated the limitations of SQL and why they created their own graph storage mechanism (Bronson et al., 2013) .

However, it is worth noting the trend of multi-modal databases. In 2016, some of the most popular Graph Database vendors became multi-modal databases. Multi-modal databases combine two or more database families into one single package. At the time of writing the most popular multi-modal database is OrientDB (DBEngine, 2017b; OrientDB, 2017). OrientDB was previously a "pure" graph database, however it is now a combination of a graph and document database (OrientDB, 2017).

It is worth mentioning that in this thesis we focus only on "pure" graph database systems. Multi-Modal databases may introduce an additional layer of complexities that may effect the graph aspect due to the addition of database types.

2.2 Graph query languages

Querying graphs are computationally hard (Barceló et al., 2011). As such, querying graph databases are also hard (Barceló Baeza, 2013). Due to the challenge of querying graphs, at the time of writing, a formalised query language for graph databases does not exist. There have been attempts at finding more efficient ways to query graphs (Lee and Chung, 2014; Zheng et al., 2014). However it is still seen as computationally hard.

Graph queries also have unavoidable overheads (Stonebraker and Cattell, 2011). These overheads alone cause the computational and time performance of graph database queries to be effected.

Because of this, a range of graph query languages exist. Some of the query languages from the late 1980's and early 1990's include G (Cruz et al., 1987), G+

(Cruz et al., 1988; Mendelzon and Wood, 1995), Gram (Amann and Scholl, 1992) and GOQL (Sheng et al., 1999). Some of the features of these languages were used as an inspiration for more recent languages. For example, G and G+ introduced recursion within a query.

Some of the more recent query languages include Gremlin (Gremlin, 2017), Cypher (Neo4j, 2017b), openCypher (Marton et al., 2017), SLQ (Yang et al., 2014), GraphQL (He and Singh, 2008), ProGQL (Tausch et al., 2011) and PGQL (van Rest et al., 2016) with others (Angles and Gutierrez, 2005). A survey of these languages has been explored in (Wood, 2012; Holzschuher and Peinl, 2013), however these are now quite dated.

It is worth noting that there has been some attempt at formalising a query language, such as openCypher (Marton et al., 2017) and SPARQL (Seaborne and Prud'hommeaux, 2008). Some aspects of these languages have been adapted by some graph database systems, but in general they have not yet been fully adapted. The most popular system at the time of writing is Neo4j. Neo4j's query language is Cypher. A specification to drive how future graph languages are implemented have been discussed (Angles et al., 2017).

There have been some studies into the performance of some graph query languages (Holzschuher and Peinl, 2013), as well as some studies into query optimisation by using algebra (Hölsch and Grossniklaus, 2016).

2.2.1 Cypher

Definition 2.2.1. A **declarative** language is a programming paradigm that expresses the logic of a computational process without having the need to describe the control flow (Lloyd, Lloyd).

Cypher is a declarative scala-based language. Cypher has been proven to be quite versatile, being used as a back-end of a Domain-Specific Language (DSL) for querying source code data (Urma and Mycroft, 2015).

Cypher aims to expressively show graph queries by using ASCII art. An example of ASCII art would be using two parentheses *()* to indicate a node. A basic Cypher query can be found in alg. 1. A basic Cypher query consists of a clause which is then followed by a pattern, restriction or expression. A formal semantic definition of Cypher has been produced, though this only focuses on read-only query results (Francis et al., 2018).

A clause is a keyword, whereas the pattern and expression can be a number of items.

Algorithm 1 A Skeleton Cypher query in its most basic form. Items in capital letters are key words. Optional clauses are surrounded by [ ]. Adapted from Drakopoulos (2016).

```
1: MATCH <pattern>
2: [WHERE <restriction>]
3: RETURN <expression> | <pattern>
4: [ORDER BY <pattern> [DESC/ASC]]
5: [LIMIT <number>]
6: [WITH <variables>]
```

Algorithm 1 shows a Cypher query at a skeleton stage. A *pattern* is a set

of symbols used to represent nodes and edges within a graph. A *restriction* is a restriction on the sub-graph found by the query. It could be searching for certain nodes within a particular set of properties for example. An *expression* is a singular variable that can be returned by the query.

Algorithm 2 A basic Cypher query. This finds all of the nodes in a graph with the property "name" and who's name has been set to "test". The query returns the neighbourhood of that node.

```
1: MATCH (n) -[r]->(m)
2: WHERE n.name = "test"
3: RETURN n,r,m
```

Using algorithm 1 and 2, a Cypher query can be dissected. The simple Cypher query in alg. 2 essentially states *" Find all of the nodes in the graph that have a name property with the name test, and return the neighbourhood of that node".*

A Cypher query begins with a *MATCH* clause. This is followed by a pattern. In alg. 2, the pattern is *(n)-[r]->(m).* The n and m in the pattern represent a node variable. The r represents an edge variable. This allows the ability to specify a certain unique node.

After the *MATCH* clause, an optional clause can be used. In alg. 2, an optional clause called *WHERE* is used. A *WHERE* clause can be used to restrict the query. In the example, *n.name* is used. The *name* represents a property of node *n.*

The final required clause in a Cypher query is *RETURN.* This essentially states what should be returned from the query. In the example query, the variables *n, r* and m are returned.

The advanced Cypher algorithm in 3 showcases some of the more advanced query features of Cypher. A *WITH* clause allows the ability to continue a query while

Algorithm 3 A more advanced Cypher query. Here we are finding all nodes with the property "age". All nodes with the age property set to 21 are then connected to all of the nodes with the age property which has been set to 22.

```
1: MATCH(h)
2: WHERE h.age = 21
3: WITH h
4: MATCH(j)
5: WHERE j.age = 22
6: WITH h,j
7: CREATE (j)–>(h)
```

Symbol	Meaning
()	Node
(n)	Variable n which represents node n
[]	Edge
[r]	Variable r which represents edge r
$<--$ or $-->$	Indicates if the query requires a direction

Table 2.1: Some of the common symbols within Cypher

saving some of the variables from the previous *MATCH* query. Cypher has proven to be quite versatile. It has been used as a Domain Specific Language for analytical processing (Bachman, 2013).

A feature of Cypher is that it contains in-built functions that are essentially algorithms. An example of this is the shortest path. At the time of writing, other common graph algorithms are not implemented into the language itself. The algorithms, or functions, that are built-in tend to be related to finding paths between nodes.

Cypher's ASCII-like language uses a number of symbols which each represent different parts of a graph. Some of the most common symbols used in Cypher are explained in table 2.2.1.

2.2.2 Sparsity's embedded library calls

In contrast to query languages, embedded databases tend to use library calls that are equivalent to query calls. We use Sparsity's library calls as an example of how they may be similar.

Algorithm 4 Creating a simple *Friend_with* relationship between two nodes.

```
1:  function SetUpGraph(Graph g)
2:                                  ▷ First we set up the Nodes, Edges and Attributes
3:      Person = new Nodetype("Person Node")
4:      name = new g.NewAttribute(Person, "Name", DataType.String,
    AttributeKind.Unique)
5:      age = new g.NewAttribute(Person, "Age", DataType.Long,
    AttributeKind.Unique)
6:      Friends_with = g.NewEdgeType("Friends_with", true,true)
7:                                                      ▷ Now we create Nodes
8:      Node Sam = g.newNode(Person)
9:      Node Bill = g.newNode(Person)
10:     g.setAttribute(Sam, name, value.setString("Sam")
11:     g.setAttribute(Bill, name, value.setString("Bill")
12:     g.setAttribute(Sam, age, value.setInt(21))
13:     g.setAttribute(Bill, age, value.setInt(22))
14:                                 ▷ Now we create an edge between Sam and Bill
15:                                              ▷ If Multi-Edges are allowed
16:     if g.hasMultiEdges then
17:         Edge = g.newEdge(Friends_with, Sam, Bill)
18:                                                  ▷ If they are not allowed
19:     else
20:         Edge = g.FindOrCreateEdge(Friends_with,Sam,Bill)
21:     end if
22: end function
```

As shown in alg. 2.2.2, creating a simple relation between two nodes takes a number of lines of code. This is to be expected as it is an imperative language, as opposed to Cypher's declarative style. Each of the nodes, properties and edges in

the graph must be given a unique type. This allows a more definitive schema to be set, in comparison to Cypher, where these can be created more freely.

2.3 Current graph database literature

Current literature of graph databases focus on comparisons with relational databases (Holzschuher and Peinl, 2013). This could be through language performance of query languages in basic queries (Holzschuher and Pein, 2016), or through general methods (Jouili and Vansteenberghe, 2013). There has also been some insight into optimising query languages (Sarwat et al., 2012; Barceló Baeza, 2013).

Other literature has also benchmarked some of the databases (Macko et al., 2013). With this, some performance evaluation of database systems in terms of queries (McColl et al., 2014) have also appeared. A review of current database systems have been released (Buerli, 2012; Angles and Gutierrez, 2008) however, these are now very dated. There has been some exploration into efficient correlation searching (Ke et al., 2008).

2.4 Problems found in graph theory

Graph theory began with the famous seven bridges of Königsberg problem (Euler, 1736). Whereby within the city of Königsberg, there existed two large islands which were connected to the mainland. Seven bridges separated the islands. A problem was set to see whether it was possible to cross each bridge once when walking around the island.

Euler proved that it was not possible to cross all bridges just once. This was found by representing the islands as nodes, and links between the islands as edges. The field of graph theory was born. To begin with, we set some definitions that will be used in this thesis. Any definition not given will be assumed to be from (Gross et al., 2013).

Definition 2.4.1. A **simple undirected Graph** G with nodes $n_1, n_2...n_n$ can be defined as $G = (N, E)$ with $n_n \in N$ and with an edge set such that $E \subseteq V \times V$. An unordered edge can be represented as (n_n, n_{n+1}) such that $(n_n, n_{n+1}) \in E$. A **simple directed Graph** G contains ordered edges, which are defined as $[n_n, n_{n+1}] \in E$ with the source of the edge being n_n and the sink of the edge being n_{n+1}. An edge may also be **weighted**. If so, the weight of the edge is denoted by $w(E)$. The **Induced Subset** of graph g is defined as $G(n_i)$, or as S. It consists of a subset of nodes from graph G, such that $S \subset V$.

Definition 2.4.2. An **undirected path** within the graph G can be defined as $v_1, e_1, v_2.....v_n, e_n, v_{n+1}$ such that $(v_n, v_{n+1}) \in E$ and the endpoints for edge e are v_n and v_{n+1}. For a **directed path**, the source of the path p is n_1 and the sink of the path is v_n. The initial node of any path is always n_1 and the final node is always n_n. As well as that, no internal node can be repeated. The nodes in-between the source and sink node in the path are known as **immediate** nodes. The path between any two nodes is represented by $p(a, b)$.

Definition 2.4.3. The **degree** of a node n , denoted as $deg(n)$, is the number of edges which are incident to n plus twice the number of self-loops. For directed graphs

we also have the in and out degree of a node. The **in-degree** of node n is the number of edges which are directed into node n or the number of edges whose sink node is node n. The set of nodes which direct into node n is also known as the **in-set** of the node n. The **out-degree** of node n is the number of edges which are directed from n, or the number of edges whose source node is node n. The set of nodes which are directed from the node n is also known as the **out-set** of node n.

Definition 2.4.4. A **cycle** within graph G is a closed path with a length of at least one. The initial and final node of the path must be the same for it to be a cycle. A graph is said to have a **Hamiltonian cycle** if the cycle contains every node in the graph once. A graph is said to contain an **Euler Cycle** if every edge e in a graph can be crossed only once within a cycle.

Definition 2.4.5. The **neighbourhood** of a vertex n within an undirected graph are the nodes which are adjacent to the nodes. Denoted as N_i. In a directed graph, the neighbourhood of node n is the **out-set** of the node n. In both cases, it can be denoted as $N(v)$.

Definition 2.4.6. An **isolated node** is a node without any edges. It has a degree of 0 ($deg(n) = 0$).

Definition 2.4.7. A **self-loop** is a cycle of length one. Essentially the source and sink of a single edge is a single node.

Definition 2.4.8. The **eccentricity** of a node is the longest shortest path for that node to any other node in a graph. It is denoted as $e(N)$.

It is worth noting that the terminology used in the different fields can vary. The term nodes is most commonly used in the graph database field, whereas in graph theory they are more commonly referred to as vertices. As well as that, edges are referred to as arcs. Certain graph database systems refer to edges as relationships.

2.4.1 Complexity

A defining problem within computer science as a field is the *P verses NP* problem. Which essentially asks can every problem that can be verified in polynomial time be solved in polynomial time? (Cook, 1971; Biggs et al., 1986)

A problem can typically be classed into three different types of complexity; Polynomial (P), Non-deterministic Polynomial-time Completeness (NP-Complete) and Non-deterministic Polynomial-time Hardness (NP-Hard). While this is a gross simplification as there are other classifications, these are most common and can apply to the algorithms found in this thesis.

Definition 2.4.9. A **Non-deterministic polynomial time (NP)** is a set of problems where a solution to the problem can be found in polynomial time. Within NP comes P and NP-Complete.

Definition 2.4.10. A problem defined as **Polynomial time (P)** is one where an optimal solution can be found and verified to be the "best" solution in polynomial time. For example, the shortest path problem finds the shortest path between two nodes in polynomial time, and the solution found will always be the "best" solution.

Definition 2.4.11. A problem found with the **Non-deterministic Polynomial Completeness time (NP-Complete)** is one where a solution can be found in polynomial time, however it cannot be verified to be the "best" solution. For example, a solution to the minimum dominating set problem can be found in polynomial time, but it is not necessarily the best set, as another run of the same algorithm could provide a different set of nodes. Should a single problem in NP-Complete have a polynomial solution, all of the problems NP-Complete can be solved in polynomial time, and thus P = NP.

Definition 2.4.12. All **Non-deterministic Polynomial Hardness time (NP-Hard)** problems are at least as hard as the hardest problems in NP. NP-Hard problems may not be in NP, as solutions may not be found in polynomial time.

The definitions found in 2.4.9, 2.4.10, 2.4.11 and 2.4.12 provide an overview of the P=NP problem. The diagram in figure 2.4 shows how the complexity area would be if $P \neq NP$ and the diagram in figure 2.5 shows $P = NP$.

2.4.2 Shortest Path and the All-Shortest Path problems

Definition 2.4.13. The **Shortest Path Problem** consists of finding the shortest possible path p_1 between any two given nodes (n_1, n_n) such that the path $p_n = n_1....n_n$. It can be denoted by $sp(N)$.

There have been multiple algorithms used to solve this problem. One of the most famous is (Dijkstra, 1959). Dijkstra's original algorithm is a greedy algorithm, and has the complexity of $O(n^2)$. However this can be improved with the addition of a

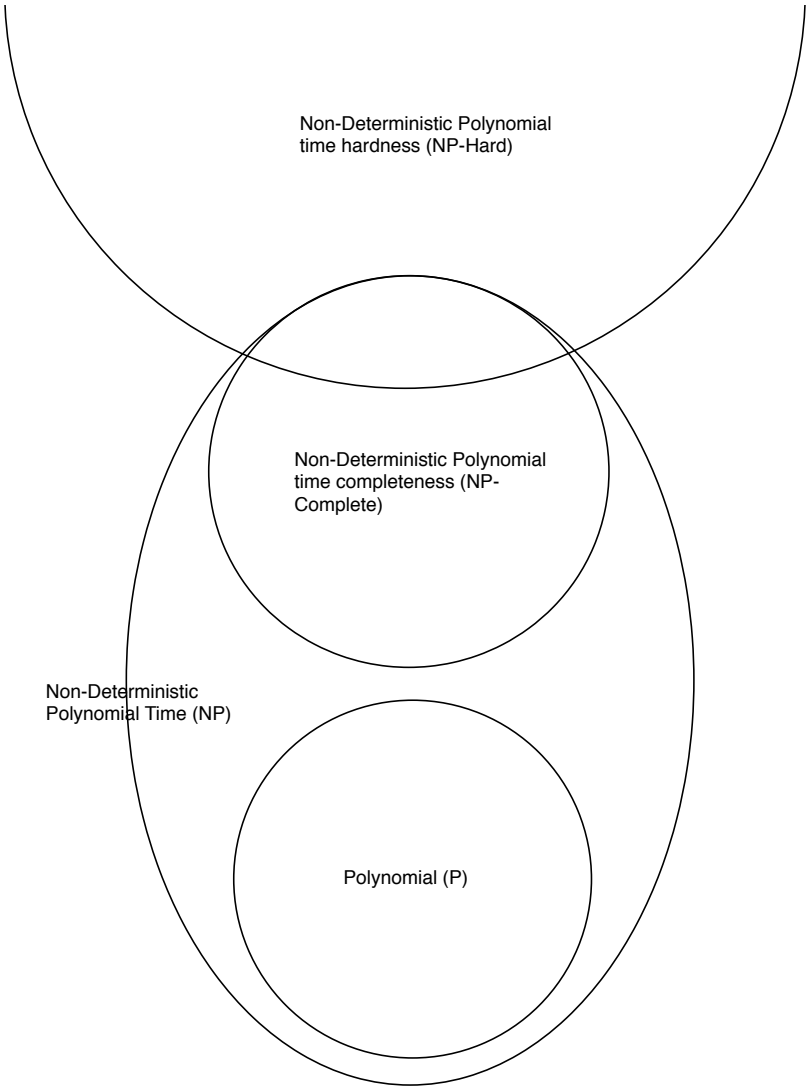

Figure 2.4: A diagram representing the NP Problem. The above scenario would be the case if $P \neq NP$.

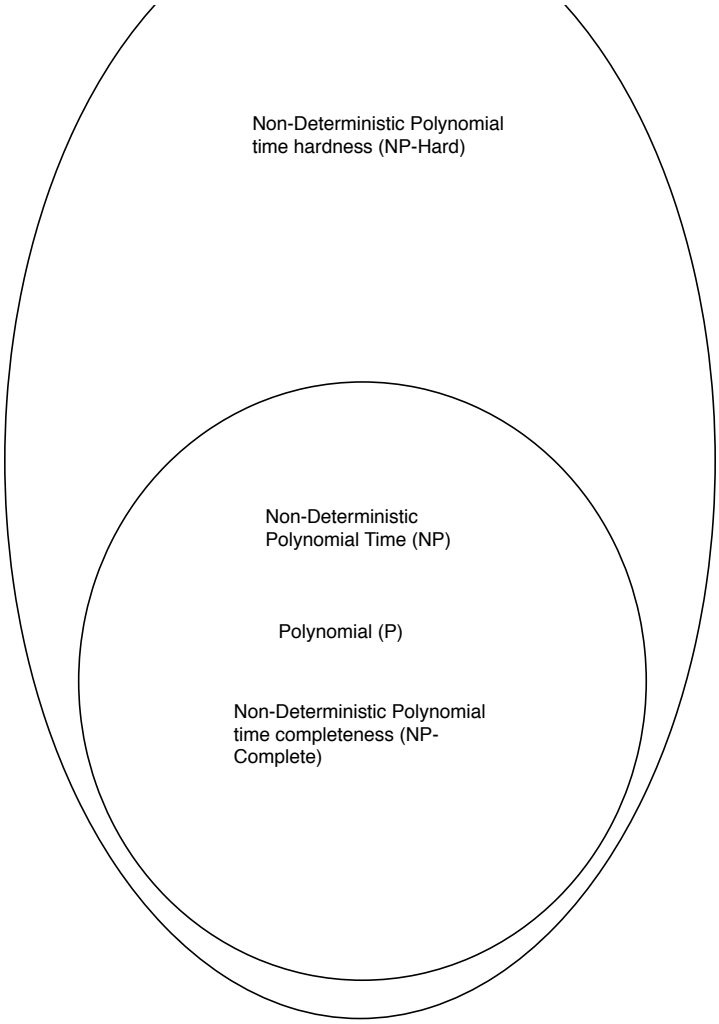

Figure 2.5: A diagram representing the NP Problem. The above scenario would be the case if $P = NP$.

priority queue. In algorithm 2.4.2, a pseudo code of the algorithm is given.

Algorithm 5 Dijkstra's original algorithm. This has a runtime complexity of $O(n^2)$ (Dijkstra, 1959). However, this can be reduced to $O(n)$ by using a priority queue.

```
1: function DijsShortPath(Graph,Source)
2:     Create node set Q
3:     for all nodes in Q do
4:         dist[n] ← ∞                          ▷ Set all Distances to Infinity
5:         prev[n] ← null                       ▷ Set all of the Parents to Infinity
6:         add n to Q
7:     end for
8:     dist[source] ← 0                  ▷ Set the distance of the source-source to 0
9:     while Q is not empty do
10:        u ← node in Q with min dist[u]
11:        remove u from Q
12:        for all neighbour v of u do
13:            alt ← dist[u] + length(u, v)
14:            if alt < dist[v] then
15:                dist[v] ← alt
16:                prev[u] ← u
17:            end if
18:        end for
19:    end while
20:    return dist[], prev[]
21: end function
```

An example of the shortest path algorithm in an instance can be found in figure 2.6. The orange coloured nodes are part of the shortest path.

The shortest path algorithm then leads to a more computationally complex algorithm. The all shortest paths algorithm finds all the shortest paths between any two nodes.

Definition 2.4.14. The **All-Shortest Path Algorithm** finds all of the shortest paths between two nodes (n_1, n_n). The fastest algorithm for this is currently the

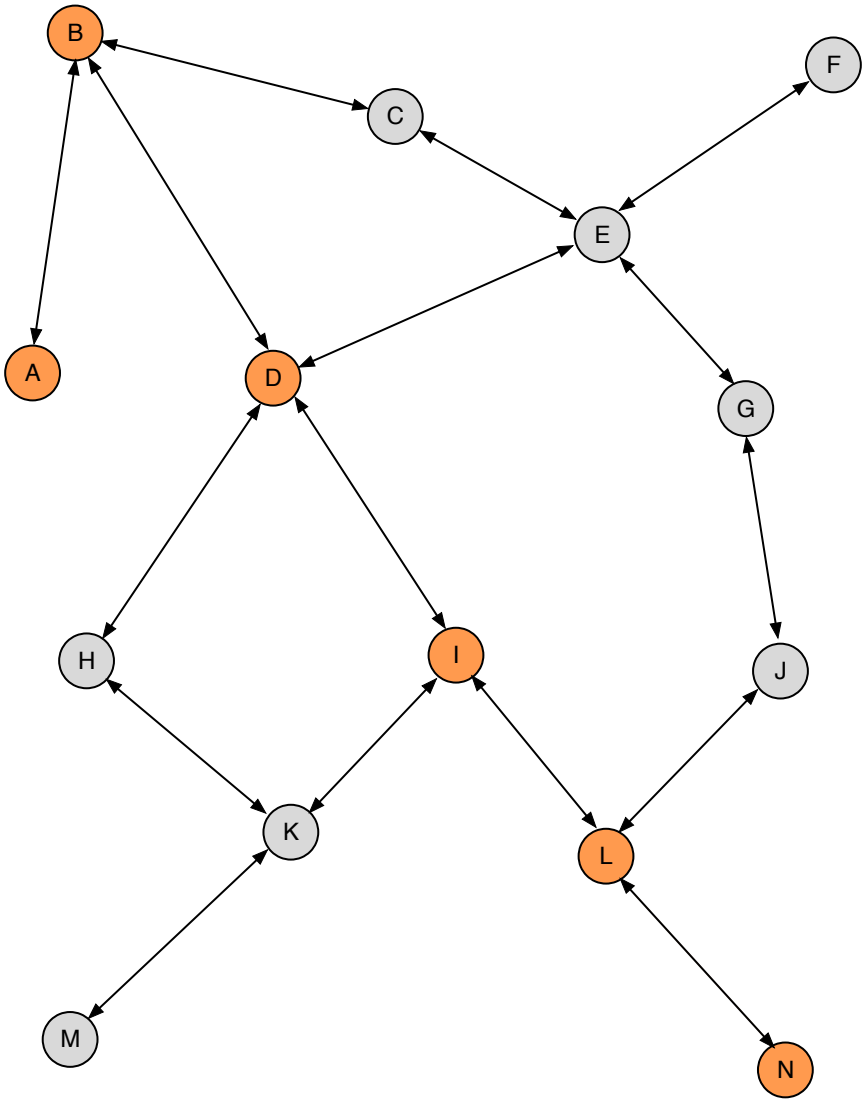

Figure 2.6: A solution for finding the shortest path between nodes A - N. The nodes coloured in orange are part of the shortest path

Floyd-Warshall algorithm, which can be found in algorithm 6 (Floyd, 1962).

Algorithm 6 The Floyd-Warshall Shortest Path Algorithm. It has a complexity of $O(n^3)$ and is generally considered the fastest algorithm in calculating all the shortest paths between two nodes as well as calculating the graph diameter (Floyd, 1962).

```
1:  function FloydWarshall(Graph g)
2:      dist[][] ← ∞                          ▷ Initialise minimum distances to infinity
3:      for all nodes n in g do dist[v][v] ← 0
4:      end for
5:      for all edge (u, v) in g do           ▷ Set the weights for each edge
6:          dist[u][v] ← w(u, v)
7:      end for
8:      for k in g do
9:          for i in g do
10:             for j in g do
11:                 if dist[i][j] > dist[i][k] + dist[k][j] then
12:                     dist[i][j] ← dist[i][k] + dist[k][j]
13:                 end if
14:             end for
15:         end for
16:     end for
17: end function
```

The all shortest path algorithm returns all shortest paths between two nodes of the same length. An example of this can be found in figure 2.7. The nodes coloured green are part of at least one single shortest path between nodes A and M.

2.4.3 Graph Radius and Diameter

Definition 2.4.15. The **Graph Diameter** problem comprises of finding the maximum length of any shortest path of a graph i.e. $max(e(N))$. The Graph Radius is the shortest length of any shortest path of a graph i.e. $min(sp(N))$. The Floyd-Warshall

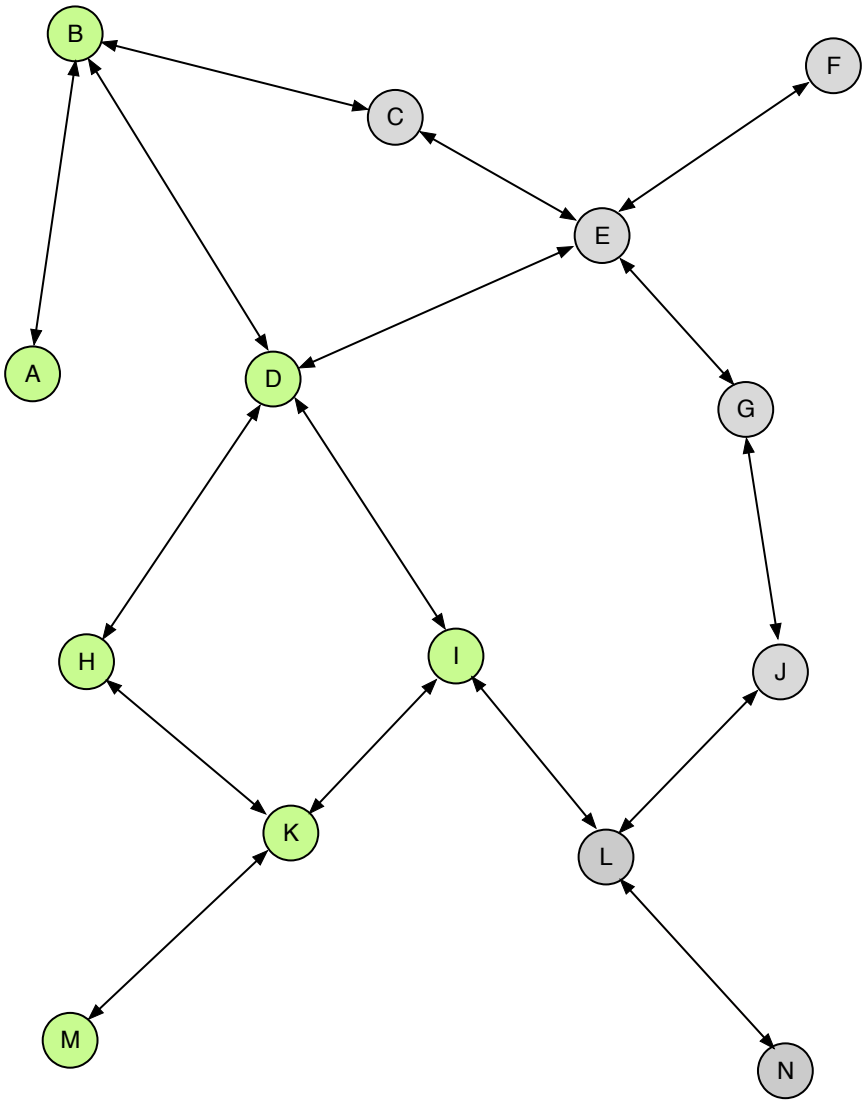

Figure 2.7: A solution for finding all of the possible shortest paths between nodes A - M. The nodes coloured in green are part of at least one of the shortest paths. There are two possible paths - (ABDIKM) and (ABDHKM).

shortest path algorithm is considered the optimal route to finding this as it has a run time of $O(n^3)$. Algorithm 6 gives the pseudocode for the algorithm.

2.4.4 Betweenness Centrality

$$g_k = \sum_{i \neq j} \frac{c_k(i,j)}{c(i,j)} \tag{2.1}$$

The Betweenness Centrality is a measurement of the centrality of vertices in a graph based on shortest paths. It is useful in finding the vertices that are the most "central" and therefore could be the most critical nodes in a graph (White and Borgatti, 1994; Freeman, 1977, 1978). The Betweenness Centrality problem was found to be NP-Complete by Opatrny (1979).

Equation 5.1 gives a formulae for Freeman's Betweenness Centrality. Let $G =$ *(N,E)* be either an undirected or directed where N is a non-empty set of nodes and E is a set of edges subject to N x N.

Given two nodes, say i and j, both of which are in N, we find all possible shortest paths in between i and j. The *count* of these are denoted as $c(i,j)$. Within the paths, the immediate nodes are denoted as $c_k(i,j)$. The weight given to each of the immediate nodes is $1/c(i,j)$. This means that the more possible shortest paths between two nodes, the less important the immediate nodes are, as there are other possible routes (Freeman, 1977).

For example, given nodes i and j, and the $count(sp(i,j)) = 1$, then each of the immediate notes in $sp(i,j)$ are each given a weighting of 1. Should $count(sp(i,j)) = 2$, then each of the immediate notes in $sp(i,j)$ are each given a weighting of 1/2.

This process is repeated for every single shortest path in the graph, with all of the weights added up. The node with the highest weighting is the most "critical" node in the graph.

Algorithm 7 A high level implementation of the betweenness centrality algorithm. It is assumed that an *allshortestpaths* algorithm that returns all shortest paths for two specific nodes exists.

```
1: function BetweennessCentrality(Graph g)
2:     double[][] nodeList
3:     for all Node i in g do
4:         for all Node j in g do
5:             List <Paths> p = AllShortestPaths(i,j)
6:             int numberOfPaths ← p.size;
7:             for all paths k in p do
8:                 currentpath ← k
9:                 currentpath.remove(0) && currentpath.remove(size(currentpath)-
   1)          ▷ Remove the first and last node of the path to leave the immediate
   nodes
10:                for all Nodes n in currentpath do
11:                    oldweight ← nodeList.at(n)
12:                    newweight ← (oldweight + (1 / amountOfPaths))
13:                    nodeList.at(n) ← newweight
14:                end for
15:            end for
16:        end for
17:    end for
18: end function
```

Algorithm 7 gives a high-level implementation of Freeman's betweenness centrality algorithm (Freeman, 1977, 1978). Figure 2.8 gives an example of betweenness centrality when it has been run on an instance. Each of the nodes in the graph are given a ranking, which is displayed to the right of each of the nodes.

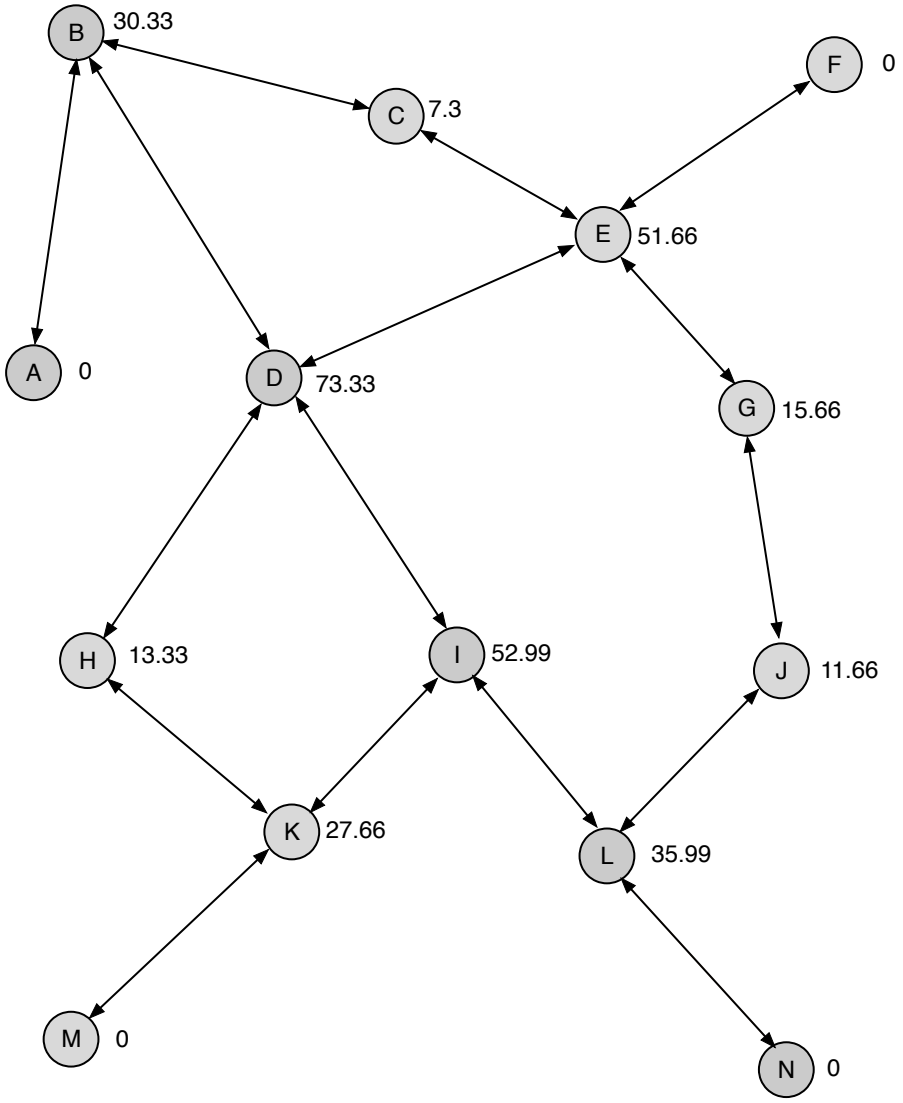

Figure 2.8: The numbers next to each of the nodes represent the "ranking" of the nodes as calculated by the betweenness centrality algorithm. From this we can see that D is ranked the highest, followed by I and then E.

2.4.5 The Minimum Dominating Set

Let $G = (N,E)$ be either an undirected or directed (digraph) were N is a non-empty set of nodes and E is a set of edges subject to N x N.

Definition 2.4.16. A dominating set in a graph G is a subset S of nodes such that every node in G is either in S or is adjacent to a node in S (Gross et al., 2013). A node is said to dominate itself, as well as its adjacent nodes. The dominating set of a graph is referred to as $\gamma(G)$.

Finding the minimum dominating set of a graph is a classical NP-Complete problem (Garey and Johnson, 1990). Therefore, there are no efficient algorithms to find the smallest dominating set of a graph.

However, some heuristics exist that can find a minimal dominating set, but not the guaranteed optimum. One such heuristic is the greedy algorithm. (Chvatal, 1979) This is defined in algorithm 8.

Algorithm 8 The greedy heuristic algorithm for finding a minimum dominating set of a graph

1: $S := \emptyset$
2: **while** $\exists$ *white nodes* **do**
3: *choose* $v \in \{x | w(x) = max_{u \in V}\{w(u)\}\}$
4: $S := S \cup v$
5: **end while**

All nodes in the graph are initially coloured white. Every isolated node is then coloured black. After this, a calculation is run to find the node with the most white-node connections. This calculation includes the current node. The chosen node's adjacent nodes are coloured grey, with itself then coloured black. This process is

repeated until all nodes are either grey or black.

The final dominating set are all of the nodes which are coloured black. Algorithm 9 gives an implementation in a high level language of the heuristic. Figure 2.9 shows an example of the dominating set in an instance. The nodes coloured blue are part of the minimum dominating set for the instance.

The dominating set algorithm has been useful in many different application areas. They have been used to find positive influence in social networks (Dinh et al., 2014), as well as efficient routing in wireless networks (Wu and Li, 1999).

The minimum dominating set is covered in more detail in chapter 4.

2.4.6 Component Labelling

The component labelling problem is one that occurs in many different application areas. The problem finds the a cluster, or components, of nodes within a graph (Vincent and Soille, 1991). A cluster refers to a group of nodes with the same label after a n hop of the algorithm.

To begin with, each node is given a unique numerical ID (essentially its label). To begin with, a random node v_1 is chosen. This is the starting node. For every neighbour of v_1, the "label" of the node is compared against the starting node. The node with the lowest label then replaces the node with the higher label. This process is repeated for every node in the graph.

The algorithm is essentially performed in *time-steps*. In an *un-directed* graph, if the algorithm were to continuously run, it will eventually label all nodes the same. Whereas in a *directed* graph, the starting node is key to the outcome of the algorithm.

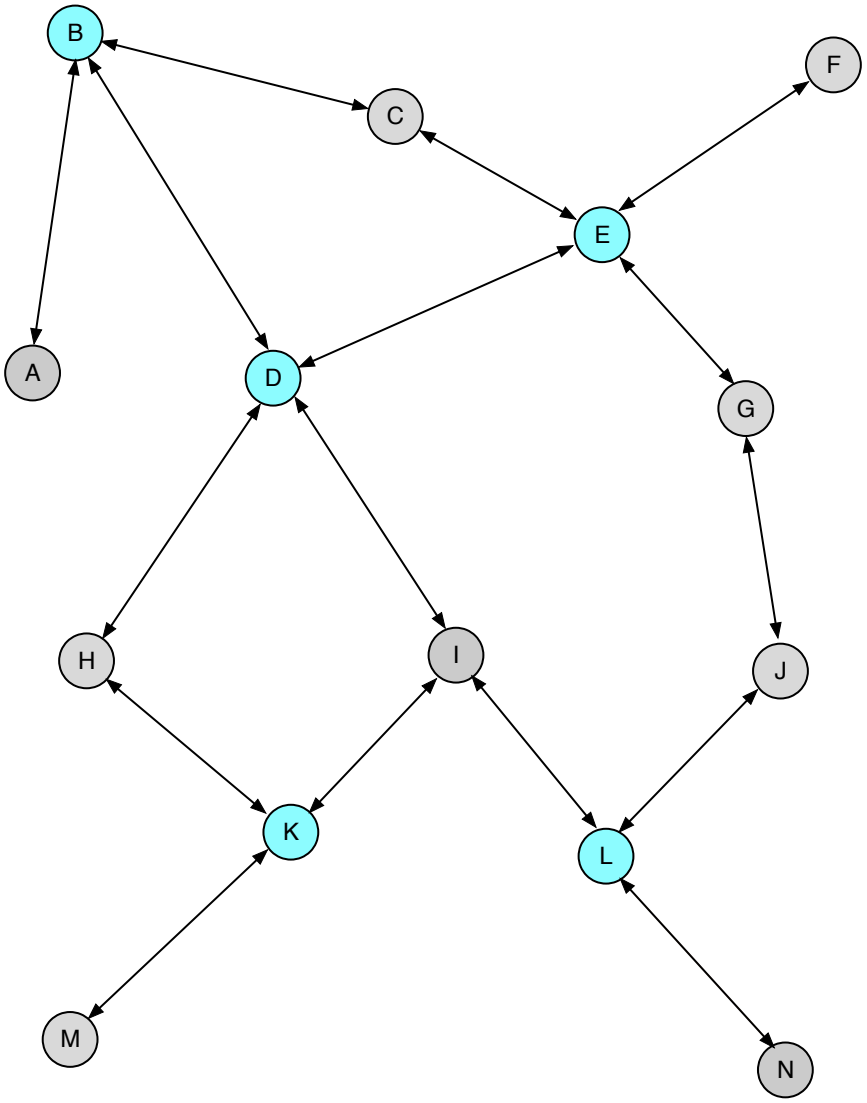

Figure 2.9: The above instance gives an example of the minimum dominating set of the instance. The nodes coloured blue are part of the dominating set.

Algorithm 9 A high level language implementation of the greedy heuristic algorithm. In the end, every node with weighting 1 is a grey node, and every node with weighting 0 is a black node.

```
1: function DominatingSet(Graph g, nodeList[])
2:     blackNodes[]
3:     whiteNodes[] ← nodeList[]
4:     weightings[] ← 0
                                        ▷ Find all Isolated nodes and make them black
5:     for all node in whiteNodes do
6:         if deg(node) < 1 then
7:             blacksNode.add(node)
8:             whitesNodes.remove(node)
9:             weighting[node] ← 0
10:        else
11:            weighting[node] ← 2
12:        end if
13:    end for
14:    while whiteNodes.size > 0 do
15:        highestweight ← 0
16:        curHighNode ← null
17:        for all node in nodeList do   ▷ Find the node with the most white-node
   connections
18:            if weightings[node] != 0 then                    ▷ If not a black node
19:                totalweight ← weightings[node]
20:                for all neighbour n of node do
21:                    totalweight += weightings[n]
22:                end for
23:                if totalweight > highestweight then
24:                    highestweight ← totalweight
25:                    curHighNode ← node
26:                end if
27:            end if
28:        end for
29:        blackNodes.add(curHighNode)
30:        whiteNodes.remove(curHighNode)
31:        for all (neighbour n of curHighNode) do
32:            weighting[n] ← 1
33:            whiteNodes.remove(n)
34:        end for
35:    end while
       return blackNodes[]
36: end function
```

Algorithm 10 The component labelling algorithm.

```
1: Input: G
2: for all Nodes k in G do
3:     k.colour = k.id                          ▷ Give a unique numerical ID value
4: end for
5: Node h = RandomNode()                        ▷ Choose a random starting node
6: for all Nodes l in Neighbourhood(h) do
7:     if l.colour ¡ h.colour then
8:         h.colour ← l.colour
9:     end if
10: end for
```

It is worth noting that other algorithmic implementations of the algorithm exist. They involve a GPU-implementation (Hawick et al., 2010a; Playne and Hawick, 2018).

2.4.7 Longest Simple Cycle

A computationally expensive problem is the Travelling Salesman Problem (TSP). The TSP asks, *given a set of cities, what is the shortest distance between each pair of cities. Each city can only be travelled to once, and the origin city must also be the final city.* It is an NP-Hard problem. The TSP problem can be solved by using an Integer Linear Programming (ILP) program (Little et al., 1963). ILP is explored further in section 2.5.

Definition 2.4.17. A *simple cycle* is a closed path without repetition of nodes or edges apart from the starting or ending node. Hence the *longest simple cycle* is a maximisation of the simple cycle.

A generalisation of the TSP is finding the longest simple cycle in a graph (Miller

et al., 1960). Finding the longest simple cycle in a graph is a NP-Complete graph problem. It consists of finding the longest cycle in a graph without repeat of edges or nodes. Figure 2.10 shows the longest cycle in an instance, with all of the nodes coloured purple which are part of the longest cycle.

In chapter 3, we create an exact algorithm using an ILP formulation to find the longest simple cycle of a graph. As well as that, we create a heuristic which initially finds a cycle with a simple Depth-First Search (DFS) and then improved the found cycle by using perturbation operators.

2.4.8 Depth-First Search

Depth-First Search (DFS) began as an investigation in the 19th century as a strategy for exploring a maze (Tarry, 1895; Gross et al., 2013). A DFS search begins from a starting node, and builts up a tree. The tree contains every possible node in g that is reachable from the starting node. In a fully connected undirected graph, every node in g would be present in the tree.

Algorithm 11 A simple Depth-first search algorithm (Gross et al., 2013)

```
1: Input: G
2: v as a starting node
3: for all v,w ∈ E do
4:     if w has not been discovered yet then
5:         Make w the child of v
6:         DSF(w)
7:     end if
8: end for
```

As DFS produces a tree, it can be used as a base for approximation heuristics.

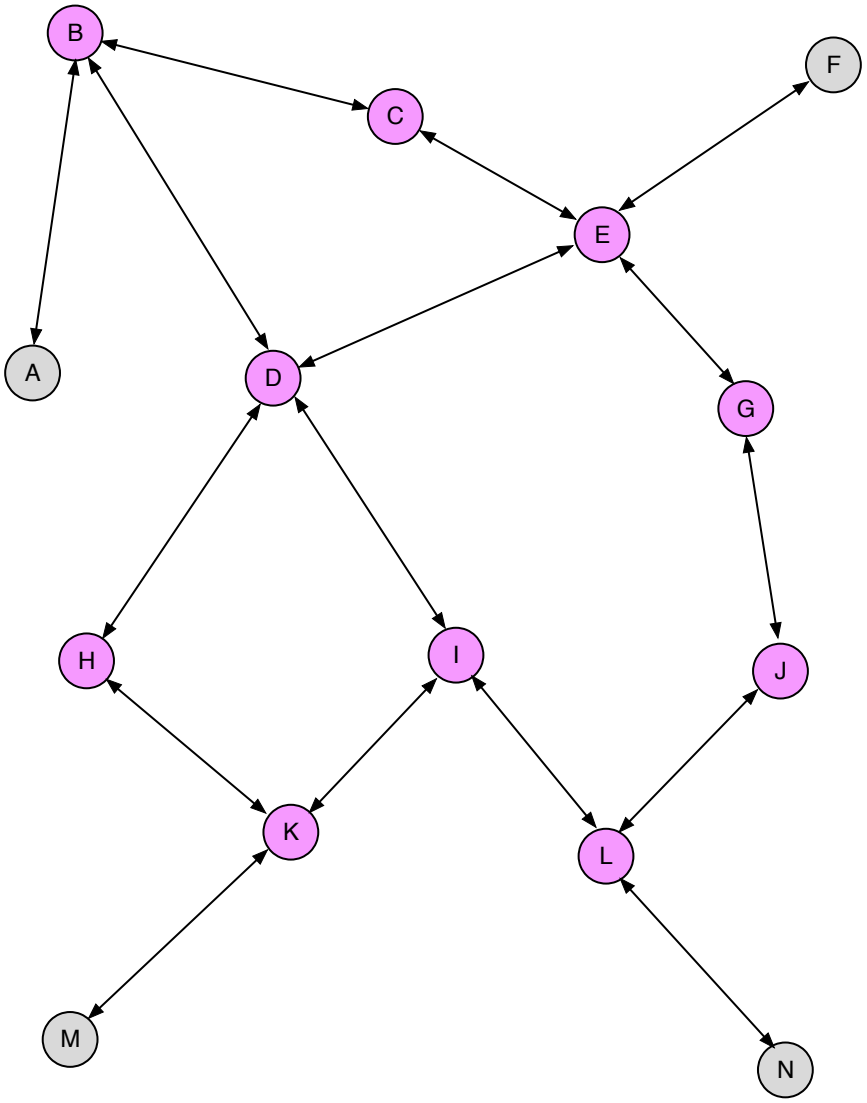

Figure 2.10: The above instance shows the longest simple cycle. The nodes coloured purple are part of the longest cycle.

Some approximation algorithms use DFS to produce a base tree and then use procedures, such as dynamic programming, to improve this (Bodlaender, 1993). In chapter 3 we create an approximation heuristic of the longest cycle. We do this by creating a base cycle by using DFS, and then improving this with perturbation operators.

2.5 Integer Linear Programming

ILP is a mathematical optimisation that reduces a problem into purely integers (Land and Doig, 2010). As well as that, the objective function and constraints are all linear (Papadimitriou and Steiglitz, 1982). A special case of ILP, which is the 0-1 integer linear, is famously one of Karp's 21 NP-Complete problems (Karp, 1972).

ILP programs can be used with branch and bound techniques to provide exact algorithms for computationally hard problems. At the time of writing, it is the most common tool used to solve NP-Hard problems (Clausen, 2003). ILP has been used to solve the TSP (Little et al., 1963), k-Nearest Neighbours (Fukunaga and Narendra, 1975), prediction in computer vision (Nowozin and Lampert, 2011) and others.

2.6 Conclusion

In this chapter we have drawn into aspects of Graph Databases and Graph Theory. We have found a considerable overlap, which was to be expected. However it is clear that there are further routes of exploration. One of which is combining the graph problems with graph databases.

As discussed in section 2.1, there seems to be limited study into combining

complex problems in graph theory and graph databases. This could be because of the relative infancy of graph databases as a whole.

Another exploration avenue is finding the longest simple cycle in an undirected graph problem, which is one that has also had limited studies. While some heuristics have been created for problems with similar aspects, they do not tackle the longest cycle in an undirected graph problem directly.

We're kept from our goal not by obstacles, but by a clear path to a lesser goal.

The Bhagavad Gita

CHAPTER 3

Finding Long Simple Cycles in undirected Graphs

Finding the longest simple cycle in a graph is a Non-deterministic Polynomial-time Hardness (NP-Hard) problem. The longest simple cycle in a graph can be defined as a cycle beginning with node n_1 and ending with the same node, with no repeated edge or node.

This chapter builds upon the work that has been previously published by the author in the journal *Knowledge-Based Systems* (Chalupa et al., 2017).

3.1 Introduction

There is a growing volume of information available as connected data. As such, the need for efficient algorithms to solve graph problems has increased (Gross et al., 2013).

The decision problem of finding if a Hamiltonian cycle exists within a graph (Karp, 1972) has been widely studied (Bianconi and Marsili, 2005; Ejov et al., 2004; Wagner et al., 1999). The longest simple cycle is a generalisation of this problem.

There have been some studies in statistical mathematics in finding the longest cycle in a graph, with an example study based on message passing and Monte Carlo procedures (Marinari et al., 2007). However exact and heuristic approaches have been limited.

Some applicational uses of the longest simple cycle include automatic drawing of planar graphs (Tamassia, 2013) and layout algorithms for metabolic pathways in bioinformatics (Becker and Rojas, 2001). There is also a close resemblance to community structure (Liu et al., 2015), its hierarchy (Chen and Li, 2015) and propagation processes in real world networks (Yang et al., 2016).

Previous studies into the longest cycle problem have been focused on the theoretical properties. These include the complexity and approximability of the problem in sparse graphs (Feder et al., 2002), bounded degree graphs (Chen et al., 2005), triangle-free graphs (Aung, 1989) and small graph classes (Uehara and Uno, 2007).

Superpolylogarithimically long cycle detection (Gabow, 2007), treewidth-based approximation (Arnborg and Proskurowski, 1989; Bodlaender, 1993) and matrix-based approximation (Monien, 1985) have been explored as potential avenues for

efficient finding of approximate solutions of the problem.

Exact methods for the longest simple cycle problem include enumerative techniques (Hawick and James, 2008; Johnson, 1975; Tarjan, 1973) and a brand and bound algorithm, which is based on a Integer Linear Programming (ILP) formulation of the problem (Dixon and Goodman, 1976).

These techniques allow exploration of the distribution of the cycle numbers as a function of the cycle length. This has been successfully explored for Kauffman networks (Hawick et al., 2007; Leist and Hawick, 2009). The techniques used are computationally demanding. An interesting problem is scaling these up for larger sized graphs.

The longest simple cycle problem is a generalisation of the Hamiltonian cycle problem. As this is the case, we explore the literature associated with the Hamiltonian cycle problem as it may be relevant. Some approximation algorithms for maximal planar graphs have been explored (Nishizeki et al., 1983). Some heuristics exist. These include some that have been ant-inspired (Wagner et al., 1999) or have an interior point (Ejov et al., 2004). Finding the Hamiltonian cycle in a scale-free network has also been explored (Bianconi and Marsili, 2005).

Another similar area to the longest simple cycle problem is the longest path problem, for which some efficient algorithms (Uehara and Uno, 2007) and approximations (Karger et al., 1997) exist. Other studies have explored directed longest cycles (Björklund et al., 2004).

The main contribution of this chapter is in proposing a new ILP formulation for the longest simple cycle problem. We also design a pipeline for an efficient exact

approach to find the longest cycle. In addition, a hybrid heuristic is proposed which combines the construction of a long cycle using depth-first search with four local search operators to improve the initial cycle found.

3.2 Exact Approach to the Longest Simple Cycle problem

Firstly, we review the current ILP formulation for the longest simple cycle problem with a fixed initial node (Dixon and Goodman, 1976). Following on, we introduce our own formulation of this problem. We also introduce a pipeline that creates a sequence of ILP problems to efficiently solve the problem solving of the problem.

3.2.1 Dixon and Goodman's formulation of the Longest Simple Cycle problem

The formulation created by Dixon and Goodman finds the longest cycle in a graph, given a fixed initial node (Dixon and Goodman, 1976). The formulation uses a "trick" in that a dummy node is created. this dummy node has the same adjacencies as the fixed node, and rather than searching for the longest cycle from this node, the longest path between the fixed node and the dummy node is then found.

Dixon and Goodman's ILP formulation of the longest cycle problem with a fixed initial node (Dixon and Goodman, 1976) Define an undirected graph $G = [V, E]$ without loops, with vertices $v_1, v_2, ..., v_n$, and assume that the starting

node for our cycle is known. This starting node is referred to as v_1. Consider a graph obtained from G by adding node v_{n+1}, which is a "copy" of node v_1, i.e. in that it has the same adjacencies as the fixed node. Let $x_{ij} \in \{0, 1\}$ represent the transitions from the node v_i to v_j, i.e. $x_{ij} = 1$ whenever the edge $\{v_i, v_j\}$ is in the cycle and $x_{ij} = 0$ otherwise. Then, we solve the following problem to obtain the longest cycle, which includes v_1:

$$\max \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij}, \tag{3.1}$$

where $c_{ij} = 1$ if vertices v_i and v_j are adjacent and $c_{ij} = 0$ otherwise:

$$c_{ij} = \begin{cases} 1 & \{v_i, v_j\} \in E \\ 0 & \{v_i, v_j\} \notin E \end{cases}, \tag{3.2}$$

subject to:

$$\sum_{i=1}^{n+1} c_{ik} x_{ik} = F_k, \quad k = 2, 3, ..., n, \tag{3.3}$$

$$\sum_{i=1}^{n+1} c_{kj} x_{kj} = F_k, \quad k = 2, 3, ..., n, \tag{3.4}$$

$$\sum_{j=2}^{n} c_{1j} x_{1j} = \sum_{i=1}^{n} c_{i(n+1)} x_{i(n+1)} = 1, \tag{3.5}$$

$$y_i - y_j + n x_{ij} \leq n - 1, \quad i, j = 1, 2, ..., n + 1, \tag{3.6}$$

$$y_i \leq n+1, \quad i = 1, 2, ..., n+1, \tag{3.7}$$

where:

$$F_i = \begin{matrix} 1 & if\ v_i\ is\ used\ in\ the\ longest\ cycle \\ 0 & otherwise \end{matrix}. \tag{3.8}$$

In this formulation, y_i are dummy integer variables introduced for each node v_i. Constraints (3.6) and (3.7) ensure that one cycle is detected, instead of a set of disjoint cycles. Dixon and Goodman state that the formulation has a highly constrained feasible region of search space. This means that the formulation can struggle with instances that stretch to tens of nodes. The formulation was tested on graphs with at most 40 nodes. In the results section of this chapter, we show that the problem can be formulated more efficiently.

3.2.2 Our formulation of the Longest Simple Cycle problem

We now present our ILP formulation of the longest simple cycle problem. The formulation adapts a flow-based formulation of the Travelling Salesman Problem (TSP) (Gavish and Graves, 1978).

The Integer Linear Programming formulation of the longest simple cycle problem with an initial fixed node To begin with, we define an undirected graph $G = [V, E]$ without self-loops. A self-loop is defined as a node with a edge to itself. Nodes are defined as $v_1, v_2, ..., v_n$, and that the starting node for our cycle has

been defined. The known starting node is referred to as v_1. Let $x_{ij} \in \{0, 1\}$ represent the transitions from the node v_i to v_j, i.e. $x_{ij} = 1$ whenever the edge $\{v_i, v_j\}$ is in the cycle and $x_{ij} = 0$ otherwise. Then, we solve the following problem to obtain the longest cycle, which includes v_1:

$$\max \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij}, \tag{3.9}$$

where $c_{ij} = 1$ if vertices v_i and v_j are adjacent and $c_{ij} = 0$ otherwise:

$$c_{ij} = \begin{cases} 1 & \{v_i, v_j\} \in E \\ 0 & \{v_i, v_j\} \notin E \end{cases}, \tag{3.10}$$

subject to:

$$\sum_{i=1}^{n} x_{ij} - \sum_{i=1}^{n} x_{ji} = 0, \quad j = 1, 2, ..., n, \ \{v_i, v_j\} \in E, \tag{3.11}$$

$$\sum_{i=1}^{n} x_{ij} + \sum_{i=1}^{n} x_{ji} \leq 2, \quad j = 1, 2, ..., n, \ \{v_i, v_j\} \in E, \tag{3.12}$$

$$\sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \geq 3, \ \{v_i, v_j\} \in E, \tag{3.13}$$

$$y_{ij} \leq (n-1) x_{ij}, \quad i = 2, 3, ..., n, \ j = 1, 2, ..., n, \ \{v_i, v_j\} \in E, \tag{3.14}$$

$$2\sum_{j=1,j\neq i} y_{ij} - 2\sum_{j=2,j\neq i} y_{ji} - \sum_{j=1,j\neq i} x_{ij} - \sum_{j=2,j\neq i} x_{ji} = 0, \tag{3.15}$$

$$i = 2, 3, ..., n,\ \{v_i, v_j\} \in E,$$

where $y_{ij} \geq 0$. Values y_{ij} represent a flow on the edges of the graph, which is used to ensure that the resulting subgraph is a single cycle, rather than a union of several disjoint cycles.

The ILP problem can be adapted to multiple starting nodes. We now prove that this ILP formulation corresponds to the problem of the longest cycle containing v_1.

Theorem Let the nodes of an undirected graph $G = [V, E]$ without self-loops be $v_1, v_2, ..., v_n$. Then, our ILP formulation of the longest simple cycle problem with a fixed initial node corresponds to the problem of finding the longest simple cycle $C = [V_C, E_C]$ such that $v_1 \in V_C$.

Proof Let $C = [V_C, E_C]$ be a simple cycle of length k in graph G, with nodes ordered $v_{c_0} = v_1$, v_{c_1}, v_{c_2}, ..., $v_{c_{k-1}}$. This means that constraints 3.11, 3.12 and 3.13 are met. This is because that for each $v \in V_C$, there is exactly one transition from v and one transition to v in our ordering. There are no transitions from and to v_i if $v_i \notin V_C$, which also meets the constraints (3.11) and (3.12). In addition, constraints 3.11 and 3.12 ensure that the flow of the cycle are of a single flow line. It follows a one edge in, one edge out flow that ensures that a cycle without repetitions of nodes and edges is found. Constraint 3.13 ensure that the length of the final cycle found is greater then or equal to 3 i.e. $k \geq 3$.

It is worth noting that constraints 3.11, 3.12 and 3.13 are used to define the cycle that is to be eventually found.

Constraint 3.14 is a capacity constraint that ensures $k-1 \leq n-1$.. A flow on the edges of G such that $y_{c_i c_{i+1}} = i$ for each $i = 0, 1, 2, ..., k-2$ and $y_{ij} = 0$ if $\{v_i, v_j\} \notin E_C$ ensures that this is met. For the constraint (3.15), consider the transitions for a node $v_i \in V_C$ first. Let $v_i = v_{c_\ell}$, i.e. let it be the $(\ell+1)$-st node in the ordering of vertices in the cycle. Then, the difference in the flow out and in the node is:

$$\sum_{j=1, j\neq i} y_{ij} - \sum_{j=2, j\neq i} y_{ji} = \ell - (\ell - 1) = 1, \tag{3.16}$$

which is equal to:

$$\frac{1}{2}\left[\sum_{j=1, j\neq i} x_{ij} + \sum_{j=2, j\neq i} x_{ji}\right] = \frac{1}{2}(1+1) = 1. \tag{3.17}$$

Every single cycle that is found meet the constraints of the ILP program.

Inversely to this, the constraints (3.11) and (3.12) can only be met if two situation takes place.

In the 1st situation follows that if there is exactly one j' and one j'' such that $x_{ij'} = 1$ and $x_{j''i} = 1$, with other values x_{ij} and $x_{ji} = 0$ for our node v_i and for $j \notin \{j', j''\}$.

The 2nd situation is that all values x_{ij} and $x_{ji} = 0$ for the node v_i. We can assume that $x_{ij'} = 1$ and $x_{j''i} = 0$ for some j' and j'', this could satisfy constraint (3.12) but would not satisfy constraint (3.11). We are able to find all of the nodes that are leaf nodes.

As all simple cycles must have minimum of 3 transitions, we have introduced constraint (3.13). The first three constraints are combined and with an assignment of values to x_{ij} , can lead to a simple cycle in G or a union of multiple disjoint simple cycles in G.

For the flow-based constraints 3.14 and 3.15, we first consider a union of two disjoint simple cycles of lengths k and t such that $k + t \leq n$ and node v_1 belongs to the cycle of length k. Consider the ordering $v_{c_0} = v_1, v_{c_1}, v_{c_2}, ..., v_{c_{k-1}}$ of nodes in the cycle of length k. Then, a flow $i \leq y_{c_i c_{i+1}} \leq n - 1$, with each node incrementing the flow by one for each $i = 0, 1, ..., k - 2$, will fulfil both of the flow-based constraints, similarly to the arguments above.

However, we can explore a simple cycle of length t, with the ordering $v_{c_k}, v_{c_{k+1}}, ..., v_{c_{k+t-1}}$ of its nodes. Then, we have that $i \leq y_{c_{k+i} c_{k+i+1}} \leq n-1$ represent the feasible values of flow for this cycle, with each node incrementing the flow by one. However, this means that the difference between the incoming and the outcoming flow for v_{c_k} will be:

$$\sum_{j=1, j \neq i} y_{c_k j} - \sum_{j=2, j \neq i} y_{j c_k} = t - 1 > 1, \tag{3.18}$$

since each node on the cycle adds one to the flow. This is in contradiction with our original premise of no repeated nodes and edges, apart from v_1. ■

The formulation establishes that our ILP formulation is correct for the variant of the longest simple cycle problem that has a fixed initial node. The general longest simple cycle problem will then be solved as a sequence of consecutive ILP problems

with different fixed initial vertices. Finding the longest simple cycle in an undirected graph is NP-hard. Therefore the computational complexity of this approach is, in worst case scenario, exponential. The maximum possible feasible and infeasible assignments of binary values to x_{ij} are 2^m, where m is the number of edges in the graph.

A general formulation to solve the Longest Cycle problem

An item of note is that it is possible to change the formulation presented above, and extend it so that a general ILP for the longest simple cycle problem can be solved. However, we found that the ILP solver seems to be more efficient when solving a sequence of ILP problems with a fixed initial node. This is probably due to the fact that large infeasible regions are induced in the search space by the addition of more constraints.

Dixon and Goodman's formulae introduces a dummy node which acts as a source for the flow, and therefore the cycle. We extend this to our formulation. Using this dummy node, we are able to search for the longest cycle from this source. A long cycle that contains the source node can be transformed into the longest cycle of the original graph. This is only the case if the neighbours of the source node on the cycle are adjacent, thus forming a triangle.

Let the source node be v_0 and let it be adjacent to all other nodes. Then the ILP problem can solve the graph with the initial node v_0 and introduce dummy variables $z_{ij} \in \{0, 1\}$. These variables represent the choice of the single edge, which ensures that our longest cycle in the modified graph can be transformed into the longest cycle of the original graph by substituting transitions x_{0j} and x_{i0} with transition x_{ij},

for which $z_{ij} = 1$. The additional constraints, which assure that v_0, v_i and v_j form a triangle, are the following:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} z_{ij} = 1, \tag{3.19}$$

$$x_{0j} + x_{i0} - 2z_{ij} \geq 0, \quad i, j = 1, 2, 3, ..., n. \tag{3.20}$$

Because of the dummy variable, the length of the cycle found by this formulation is higher than the actual optimal longest cycle by one.

3.2.3 Pipeline design for the exact approach

After the creation of the general formulation for the longest cycle problem, we found that this approach was less efficient at finding the longest cycle that the ILP program with a fixed initial node. As the fixed initial node formulation finds the longest cycle for the starting vertex, and not necessarily the longest cycle for the instance, a pipeline was created. The pipeline would handle a sequence of ILP programs that used our fixed initial node formulation above.

Definition 3.2.1. Pruning the leaves of a graph is a process that essentially finds all of the nodes on the "outskirts" of a graph, essentially the nodes with a degree of 1 or less. A **leaf** node is a node with degree 1. These nodes cannot be part of any cycle in a graph, as a node requires a degree of 2 or more.

Figure 3.1 shows the process describing the data mining pipeline. Algorithm 12 shows the data mining pipeline as an algorithm. Firstly, we prune the leaves of the

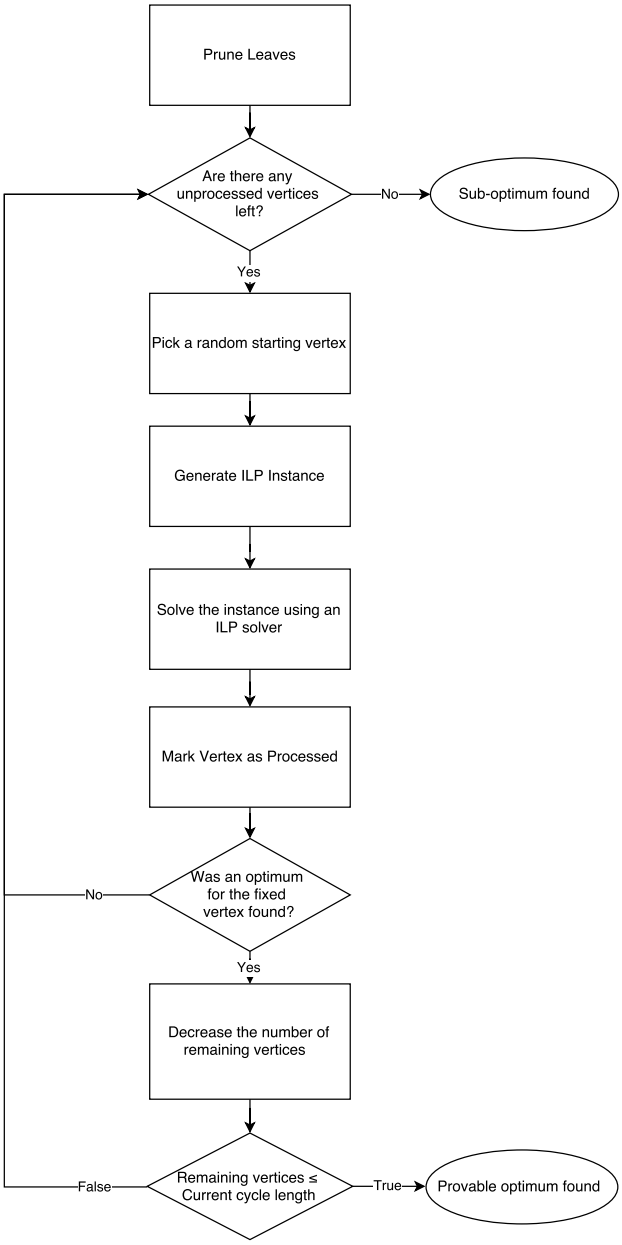

Figure 3.1: The pipeline approach for finding the longest cycle with a fixed initial node. Firstly, the leaves of the graph are pruned as they cannot be part of the longest cycle. We then generate a sequence of ILP instances for fixed nodes that are randomly chosen.

graph, as a node must have a degree of at least two be a part of the longest cycle. We then check for any unprocessed nodes, should none exist then a suboptimal cycle has been found. This is possible in a instance where all nodes have a max(deg(2)). If an unprocessed node has been found, we pick a random node from the remaining unprocessed nodes to be our fixed initial node.

We used the same methodology for our fixed initial node and Dixon and Goodman's formulations.

We then create an ILP instance with the randomly selected fixed initial node based on either our formulation, or on Dixon and Goodman's formulation. A branch-and-cut solver is used to solve the instance created. We then read the output from the solver. The may contain three different outcomes:

- An optimal cycle , i.e. the longest simple cycle where the starting node is part of the cycle is found
- A sub-optimal cycle may be found
- A time-out has occurred

Definition 3.2.2. When an *optimal* cycle has been found,it is the proven longest cycle for the fixed initial node. A *optimal solution* is where the longest cycle for the whole instance is found.

An optimal cycle is one that is proven to be the longest possible for the fixed initial vertex. A sub optimal cycle is where a cycle has been found, however it cannot be proven to be the longest for that vertex because a search for a larger cycle

Algorithm 12 The data mining pipeline algorithm. It is assumed that the instance has already been pruned prior to this. generateInstance is a method call that creates a generated instance that can be passed into CBC to solve.

```
Timelimit = 1
startingNode, currentLongestCycle, remainingNodes = -1
int[] nodeList, edgeList
provableOptimum = false
nodeList ← g.N(), edgeList ← g.E()
for all i in nodeList do
    startNode = random.next(0, nodeList.size())
    output = generateInstance(startNode, Timelimit)        ▷ Create CBC instance with startNode and time limit
        ▷ The output of the instance is a String with a result, and the longest cycle found
    outputString, outputLongestCycleFound ← output
    if outputString.equals("Optimal Cycle Found") then
            ▷ Reduce the amount of remaining possible nodes that can be part of the longest cycle
        remaining–
    end if
    if outputLongestCycleFound > currentLongestCycle then
        currentLongestCycle = outputLongestCycleFound
    end if
    if remaining <= currentLongestCycle then
            ▷ If the amount of nodes remaining is less that the optimum longest cycle found, then a provable longest cycle has been found
        provableOptimum = true
        break
    end if
end for
```

has caused the solver to time-out. A time out is one where a single cycle for that fixed initial vertex cannot be found at all.

We mark the initial node as processed. We then check for an optimal solution, which may be possible at this stage. Should the optimal solution not be found, we process the next available node, should the list not be empty.

Throughout the process, we check for an optimal solution. Once found, we check the outcome of the instance and compare its findings with the highest length found so far. If it is higher, we replace the current best cycle. After this, the remaining nodes to process count is decreased. The count is then compared to the current longest cycle length. If it is less than the current longest cycle length, a provable optimum has been found and the process ends. Otherwise we continue to process the next available node, should one exist.

3.3 Heuristic methods for the Longest Cycle problem

While the exact approaches provide a solution to finding the longest simple cycle in a graph, the methods are very computationally demanding. This is expected,as they are attempting to solve a NP-Hard problem. A high interest area would be creating a new heuristic, or an approximation algorithm. Some aspects of approximation algorithms in other areas use recursive disjoint cycle detection procedures (Björklund et al., 2004). They also may employ detection of cycles to enlarge other cycles. These methods used previously can possibly be used in an iterated way (Gabow, 2007).

Some earlier approximation algorithms have used two phases, combining Depth-

First Search (DFS) for sampling of the initial solution with dynamic programming as the second phase (Bodlaender, 1993).

Combining these different approaches, that have been proven to improve a given function within there respective areas, allows for a novel implementation of a search heuristic for the simple longest cycle problem.

In the following section, we present an approximation algorithm which uses a DFS procedure to sample an initial long cycle. The cycle that has been found can be improved with four perturbation operators.

To begin with, a DFS is used to sample long paths. In each of the branching steps of the DFS search, we scan the neighbours in a random order. This means that the use of DFS is in a randomised algorithm. The paths found by the previous step are then filtered to those which can be "closed". A closed path representing a cycle.

The pseudocode of our approximation algorithm can be found in algorithm 13. Firstly, we start with an empty cycle C and set the initial cycle length l to 0. An iterative procedure over all nodes follows.

We choose a fixed initial node, which is shown as $v_{central}$. A cycle is then constructed by using this initial fixed node. Function $d(v)$ denotes the distance of v from $v_{central}$, while function $p(v)$ is a binary value that indicates if the node has already been processed by DFS. The nodes of the instance are put into a stack S. We begin by popping the current node from the stack. If it has not yet been processed, we then scan the neighbours of the node.

Let w be the neighbour that is currently being scanned. Then, if the path from $v_{central}$ from v to w is longer than the one previously sampled, then w is moved to

the top of the stack S. This change brings a preferential for longer paths. The array *parent* is used to record predecessors of nodes in DFS. This allows the final long cycle to be traced. The current longest cycle C if $v_{central}$ is updated if a cycle is reached and tyhe length of the cycle d_c is the highest length found so far.

This initial long cycle is then improved using a selection of perturbation operators. We use a selection of local search approaches to enlarge the initial long cycle. These search approaches become our perturbation operators. In total. we use four perturbation operators to improve the cycle. Two of these operators serve as improvement operators. These enlarge the current found cycle by substituting subpaths of length 1 with subpaths of length 2 or 3 by identifying triangles or rectangles with each node pair in the cycle. The plateau exploration operators substitute paths in the longest cycle.

Definitions 3.3.1, 3.3.2, 3.3.3 and 3.3.4 formally define the perturbation operators. Before this, we provide some terms for all operators. Let $C = [v_{c_1}, v_{c_2}, ..., v_{c_k}]$ be a cycle of length k of nodes on graph $G = [V, E]$. Let V_C be the set of nodes in C and let E_C be the set of edges in C.

Definition 3.3.1 gives the formal definition of the triangular improvement operator. The rectangular operator is defined in definition 3.3.2. The plateau exploration operators are in definitions 3.3.3 and 3.3.4.

Definition 3.3.1. *Perturbation operator 1 (triangular improvement operator)* - If there is a node $w \in V \backslash V_C$ such that for some i it holds that $\{v_{c_i}, v_{c_{i+1}}\} \in E_C$ and vertices $v_{c_i}, w, v_{c_{i+1}}$ form a triangle, then G also contains a cycle $C' = \{v_{c_1}, v_{c_2}, ..., v_{c_i}, w, v_{c_{i+1}}..., v_{c_k}\}$ of length $k + 1$. This allows for high improvements in instances with

Algorithm 13 The heuristical algorithm for finding a long cycle in a graph using DFS.

```
G
C = [], I = 0
for all v_central ∈ V do
    ∀v ∈ V let d(V), p(v) = 0
    S = [v_central]
    while S contains at least one node do
        v = pop(s)
        if p(v) = 0 then
            d_c = d(v), p(v) = 1
            for all w such that v, w ∈ E in a random order do
                if p ≠ 0 ∧ w ≠ v_central ∧ d(w) ≤ d_c + 1 then
                    remove w from S if w ∈ S
                    S = push(w)
                    d(w) = d_c + 1, parent(w) = v
                end if
                if w then = v_central ∧ d_c ≥ 2 ∧ l ¡ d_c + 1
                    l = d_c +1
                    use array to update the current longest cycle C to v_central
                end if
            end for
        end if
    end while
end for
return C
```

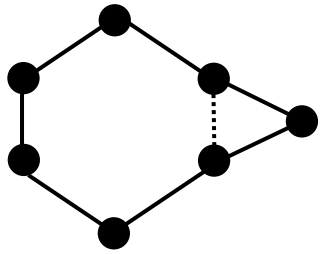

Figure 3.2: The figure above shows the Perturbation operator one. This represents the triangle improvement operator. If an edge exists between two nodes in a cycle, and if a further node connects to these two to form a triangle, then the found node must be part of the longest simple cycle.

a high level of triangles. Figure 3.2 shows a representation of this.

Definition 3.3.2. *Perturbation operator 2 (rectangular improvement operator)* - If there are nodes $w_1, w_2 \in V \backslash V_C$ such that $\{w_1, w_2\} \in E$ and for some i it holds that $\{v_{c_i}, v_{c_{i+1}}\} \in E_C$ and the sequence $[v_{c_i}, w_1, w_2, v_{c_{i+1}}]$ represents a cycle of length 4, then G also contains a cycle $C' = \{v_{c_1}, v_{c_2}, ..., v_{c_i}, w_1, w_2, v_{c_{i+1}} ..., v_{c_k}\}$ of length $k+2$. This allows for improvement in instances that are grid based. Figure 3.3 explains the operator in further detail.

Definition 3.3.3. *Perturbation operator 3 (plateau exploration operator 1)* - If there is a node $w \in V \backslash V_C$ such that for some i it holds that $\{v_{c_i-1}, v_{c_i}\} \in E_C$ and $\{v_{c_i}, v_{c_{i+1}}\} \in E_C$, then if $\{v_{c_i-1}, w\} \in E$ and $\{w, v_{c_{i+1}}\} \in E$ as well, then G also contains a cycle $C' = \{v_{c_1}, v_{c_2}, ..., v_{c_{i-1}}, w, v_{c_{i+1}}, ..., v_{c_k}\}$ of length k. This allows further avenues of the instance to be explored. Figure 3.4 explores this in detail.

Definition 3.3.4. *Perturbation operator 4 (plateau exploration operator 2)* If there

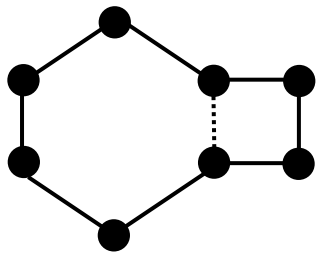

Figure 3.3: The figure above shows the Perturbation operator two. This represents the rectangle improvement operator. In a similar fashion to the triangular operator, If an edge exists between two nodes in a cycle, and if two further nodes connects to these two to form a rectangle, then the found nodes must be part of the longest simple cycle.

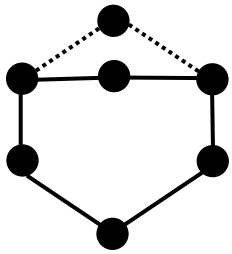

Figure 3.4: The figure above shows the Perturbation operator three. This represents the plateau exploration operator 1. As the heuristic is an iterative process, one such way of exploring the search space of the instance is by identifying other paths that can exist between two nodes in the cycle. This then opens up the opportunity for the improvement operators to find a larger cycle.

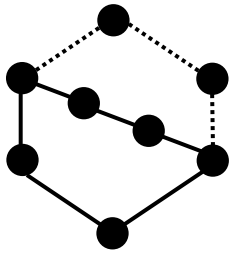

Figure 3.5: The figure above shows the Perturbation operator four. This represents the plateau exploration operator two. In similar vein to the plateau exploration operator one, a way of exploring the search space of the instance is by identifying other paths that can exist between two nodes in the cycle. This then opens up the opportunity for the improvement operators to find a larger cycle.

are vertices $w_1, w_2 \in V \backslash V_C$ such that and edge exists between them; $\{w_1, w_2\} \in E$ and for some i it holds that $\{v_{c_{i-2}}, v_{c_{i-1}}\} \in E_C$, $\{v_{c_{i-1}}, v_{c_i}\} \in E_C$ and $\{v_{c_i}, v_{c_{i+1}}\} \in E_C$, then if $\{v_{c_{i-2}}, w_1\} \in E$, and $\{w_2, v_{c_{i+1}}\} \in E$ as well, then G also contains a cycle $C' = \{v_{c_1}, v_{c_2}, ..., v_{c_{i-2}}, w_1, w_2, v_{c_{i+1}}, ..., v_{c_k}\}$ of length k. This allows further avenues of the instance to be explored.

The perturbation operators shown above use a variety of properties from real world sourcers to improve the long cycle. Operator one and two uses the properties of real-world complex networks, such as the clustering coefficient metric or grids. This metric is defined as finding the ratio of the number of triangles to the number of all connected triplets of nodes (Barabási and Albert, 1999). The computational complexity of DFS is O($|V| + |E|$) (Gross et al., 2013). For every node in the graph we are running a DFS search, therefore the computational complexity of the heuristic is O(V*E).

Many real world networks have high clustering coefficients. The property also holds for generated graphs, either the Watts-Strogatz model (Watts, 1999; Watts and Strogatz, 1998), or by the Barabási-Albert model (Barabási and Albert, 1999; Albert and Barabási, 2002).

For grid-based networks, the rectangular operator seems to be a much more valid choice. The underlying structure of these networks rely on a mesh-like instance. For these networks, triangular connections will be rare, therefore a rectangular perturbation operator will be used.

The exploration operators exploit different properties of different kinds of complex networks. Operator three substitutes one triplet with another, while operator four substitutes a 4-tuple with another.This allows an exploration of the instance in a new area.

3.4 Results

The following section presents the results from the computational experiments performed. Firstly, we explain the experimental design, then the numerical results. In this section, we compare the longest cycle found and the time taken to find these results. We compare Dixon and Goodmans's ILP program with our exact solver and our heuristic.

3.4.1 Experimental design

For our and Dixon and Goodman's ILP approaches, we have used an open-source branch-and-cut ILP solver CBC from the COIN-OR package (Bonami et al., 2008; Linderoth and Lodi, 2011). By combining the approaches with the pipeline, we were able to solve the problem as a sequence of programs. The programs were solved using CBC with a predefined time limit.

We applied 3 different time limits. *1* minute per node, *10* minutes per node and *1* hour per node.

For each of the separate integer programs with the fixed initial node, the ILP can:

- Find an optimum cycle
  - This is the longest possible cycle for that fixed initial node.
  - However, it is not necessarily the longest cycle in the graph.
- Obtain a timeout
  - This is due to the time limit restriction.
- Obtain a sub-optimal solution
  - A long cycle was found, but within the time limit it could not guarantee that this is the longest cycle for that node.

From this, we can deduce that if the number of vertices which can still potentially form a cycle is not higher than the best sub-problem solution found so far, then this

solution can be deemed optimal. These time limit restrictions allow us to categorise the problem instances:

- *Easy instances* are those where the optimal longest cycle was found with a *1* minute time limit
- *Medium difficulty instances* are those where the optimal longest cycle was found with a *10* minute time limit
- *Hard instances* are those where the optimal longest cycle was found with a *1* hour time limit

Any instance that could be solved with a 1 minute time limit was declared easy. Similarly any that could be solved in 10 minutes was declared medium. The rest of the instances were classified as hard.

The pipeline was implemented as a Python script. This script chooses an initial node, and handles the ILP program generation. The handling includes launching CBC and maintaining if the optimum has been found.

The heuristic approach was implemented in C++ using the Qt toolkit. Similarly to the exact approach, the heuristic was also applied to the graphs obtained by iterative pruning of the leaves.

All experiments were run on an Apple Mac Pro running OS X El Capitan with a 3.5GHz 6-Core Intel Xeon E5 CPU and with 16 GB 1866 MHz DDR3 RAM.

As leaf nodes cannot be part of the longest cycle, we prune them. This can reduce the size of an instance quite significantly. For example, the instance $gplus\_200$ of

200 nodes representing the public circles data from Google+ was reduced from 200 to 118 nodes.

Table 3.1: A table to show a summary of the data sets used. For each of the data set, we show the name, the amount of nodes and edges inside of the instance and the type.

Instance	Nodes	Edges	Type of Instance
gplus_200	200	527	Social Media
gplus_500	500	1346	Social Media
pokec_500	500	1276	Social Media
soc_52	52	822	Social Media
adjnoun	112	425	Adjective-noun adjacencies for David Copperfield
football	115	616	American college football network
lesmis	77	254	Coappearance network for Les Misérables
netscience	1589	2742	Network science collaborations,
zachary	34	78	Social network of a karate club
celegansneural	297	2359	Neural network for nematode worm Caenorhabditis elegans
dolphins	62	159	Social network of bottlenose dolphins
polbooks	105	441	Network of books about US politics
Celeg20160114CR	198	155	A protein-protein interaction network for Caenorhabditis elegans
Dmela20160114CR	584	627	A network for the fruit fly, Drosophila melanogaster.
Ecoli20160114CR	1091	1639	A protein-protein interactions for Escherichia coli
Hpylo20160114	700	1372	Network for Helicobacter pylori, a bacterium associated with chronic gastritis
Hsapi20160114HT	342	436	A High-throughput human protein-protein interaction network
Mmusc20160114CR	2127	2283	Protein-protein interaction network for the house mouse
anna	138	986	Network for Anna Karenina
david	87	812	Coappearance network for David Copperfield
homer	561	3258	Network for Iliad and Odyssey
huck	74	602	Network for Huckleberry Finn.

We have used a wide range of real-world networks with varying sizes. These are shown in table 3.1. The type of the network is also described in the table. We now describe the data in more detail.

We have a selection of social media networks. A small-sample instance is *soc_52*. Some of the larger samples include data from Google+ and a slovak social network from Pokec (Takac and Zabovsky, 2012). A larger snapshot of the data comes from the SNAP data set (Leskovec and Krevl, 2014). We also use various networks from Newman's data repository. These include a adjective-noun adjacencies for

David Copperfield (Newman, 2006), an American college football network (Girvan and Newman, 2002), a coappreance network for Les Misérables (Knuth, 1993) . Other networks within the repository include *netscience* which is a representation of network science collaborations Newman (2006). The network *zachary* is a social network of a karate club (Zachary, 1977). Another network, *Celegansneural* is a neural network for nematode worm Caenorhabditis elegans (Watts and Strogatz, 1998) while *dolphins* is a social network of bottlenose dolphins (Lusseau et al., 2003). Finally, *polbooks* represents a network of books about US politics.

The next subset of instances are the protein-protein interaction networks from the UCLA database of interacting proteins (Salwinski et al., 2004; Xenarios et al., 2000, 2001, 2002). The networks can represent two different states of the data. Either it is the full data set, which is represented by the *CR* suffix, or it is a high-throughput data, which is represented with a *HT* suffix.

The network *Celeg20160114CR* is a representation of a protein-protein interaction network for Caenorhabditis elegans. *Dmela20160114CR* is a network for Drosophila melanogaster, the common fruit fly. *Ecoli20160114CR* represents the protein-protein interactions for Escherichia coli. There is also a network for Helicobacter pylori, *Hpylo20160114*.This is a bacterium associated with chronic gastritis. *Hsapi20160114HT* is a high-throughput human protein-protein interaction network. The final instance is a protein-protein interaction network for the housemouse, *Mmusc20160114CR*.

The last group represents the coappearance networks for several literary classics from the DIMACS graphs (Johnson and Trick, 1996). *Anna* is a network for Anna Karenina, *david* is a coappearance network for David Copperfield, *homer* is the

network for Iliad and Odyssey and *huck* represents the network for Huckleberry Finn.

3.4.2 Results of the comparison between Dixon and Goodman's and our own ILP formulation

To begin with, we compare our approach with Dixon and Goodman's formulation. The approach to finding the longest cycle from a fixed initial node are the same for both of these approaches. We set the time limit per node to 1 minute as an initial comparison. The results of this comparison can be found in table 3.2.

Each of the rows in the table represent an instance. And the data in the columns is the longest cycle found, with the time taken (in seconds) to find this. If the result is a number, then the proven optimum has been found. If the number begins with a $\geq$, then a sub-optimum result has been found. Essentially this is when a cycle has been found, but it cannot be proven to be the optimum. If N/A is the result, then no cycle has been found within the time limit.

The results of the comparison can be found in table 3.2. We find that our formulation significantly outperforms Dixon and Goodman's. Out of the 22 instances, our formulation found a solution for 11, however Dixon and Goodman's found only 3. Our formulation also took significantly less time. Our flow based constraints allow for a constrained search space, which could explain the difference in time. We have also performed equivalent experiments with 10 minute time limit.

The results for these can be found in table 3.3. In these results, the formulation led to 5 out of 22 solutions being found. This is still considerably less successful than

Table 3.2: A table to show a comparison of our formulation and Dixon and Goodman's formulation (Dixon and Goodman, 1976) set with a 1 minute time limit per node. A N/A indicate that a result has not been found. A $\geq$ indicates that a sub-optimal solution has been found. The results indicate that our formulation outperforms Dixon and Goodman's.

Graph	CBC (1 min / node) our ILP Formulation		CBC (1 min / node) Dixon and Goodman's ILP Formulation	
	optimum	time	optimum	time
Social networks				
gplus_200	≥ 70	5117 s	≥ 60	6887 s
gplus_500	≥ 8	16353 s	N/A	18375 s
pokec_500	≥ 127	17485 s	N/A	20005 s
soc_52	51	15 s	51	2034 s
Graphs from Newman's network data repository				
adjnoun	101	207 s	≥ 100	6063 s
football	115	4 s	≥ 113	7147 s
lesmis	49	490 s	≥ 49	3683 s
netscience	≥ 88	25495 s	N/A	66573 s
zachary	20	19 s	≥ 20	1616 s
celegansneural	≥ 279	17959 s	N/A	17150 s
dolphins	53	≤ 1 s	53	22 s
polbooks	105	60 s	≥ 103	6003 s
Protein-protein interactions from UCLA database of interacting proteins				
Celeg20160114CR	6	≤ 1 s	6	199 s
Dmela20160114CR	14	49 s	N/A	5342 s
Ecoli20160114CR	≥ 8	18380 s	N/A	21695 s
Hpylo20160114	N/A	26484 s	N/A	25364 s
Hsapi20160114HT	64	1959 s	≥ 61	5167 s
Mmusc20160114CR	≥ 170	22012 s	N/A	32981 s
DIMACS graphs				
anna	≥ 77	6490 s	≥ 75	6522 s
david	≥ 72	4621 s	≥ 66	4685 s
homer	≥ 114	18309 s	N/A	19412 s
huck	48	858 s	N/A	3734 s

our approach with 1 minute time limit.

3.4.3 In-depth results of our ILP formulation and our heuristic

We now explore the numerical results of our own ILP formulation. Table 3.4 presents the results we obtained. We split the three time limitations in separate columns, with 1 minute, 10 minute and 1 hour time limit being put into different columns. The meanings of numerical values and symbols in the table are equivalent to their respective meanings in table 3.2.

We use the results of the experiment to categorise the instances. We categorise them into easy, medium and hard. The results from table 3.4 show that for the 22 networks we studied, eleven of the instances were classified as easy. Only three of the instances were classified as medium difficulty. Eight of the instances were put into the hard category. In the table, if an instance has been solved within a time limit that is less than the higher time limits, then we omit results with a higher time limit.

The instances that were classified as easy are *soc_52*, *adjnoun*, *football*, *lesmis*, *zachary*, *dolphins*, *polbooks*, *Celeg20160114CR*, *Dmela20160114CR*, *Hsapi20160114HT* and *huck*. This is because the longest simple cycle can be found in these instances in less than a hour. All four types of networks have a representative between the easy instances. From these results we can determine that there seems to be no correlation between the application domain from which the instance comes and the difficulty of the instance.

Once we reduce the time limit to 10, we are able to solve three more instances.

Table 3.3: Following on from table 3.2, the time taken to find a long cycle using Dixon and Goodman's formulations with a time-out of 10 minutes. For results found with a one minute time-out, they are coloured red. Any result with a $>$ symbol states that a provable optimum has not been found.

Graph	CBC (10 min / node) our ILP Formulation		CBC (10 min / node) Dixon and Goodman's ILP Formulation	
	optimum	time	optimum	time
Social networks				
gplus_200	70	30850 s	≥ 65	52660 s
gplus_500	≥ 202	157858 s	N/A	111007 s
pokec_500	≥ 163	168396 s	N/A	141220 s
soc_52	51	15 s	51	2034 s
Graphs from Newman's network data repository				
adjnoun	101	207 s	101	8468 s
football	115	4 s	≥ 114	53762 s
lesmis	49	490 s	≥ 49	27692 s
netscience	≥ 104	194185 s	N/A	428665 s
zachary	20	19 s	20	6432 s
celegansneural	280	2785 s	≥ 273	85522 s
dolphins	53	≤ 1 s	53	22 s
polbooks	105	60 s	105	2425 s
Protein-protein interactions from UCLA database of interacting proteins				
Celeg20160114CR	6	≤ 1 s	6	199 s
Dmela20160114CR	14	49 s	N/A	10381 s
Ecoli20160114CR	≥ 242	179389 s	N/A	63888 s
Hpylo20160114	≥ 291	260322 s	≥ 271	119186 s
Hsapi20160114HT	64	1959 s	≥ 64	48469 s
Mmusc20160114CR	≥ 256	215912 s	N/A	173966 s
DIMACS graphs				
anna	≥ 79	64498 s	≥ 78	42979 s
david	72	4714 s	≥ 72	31221 s
homer	≥ 223	171157 s	N/A	78549 s
huck	48	858 s	≥ 46	26986 s

These are classified as medium. They include *gplus_200*, *celegansneural* and *david*.

Instances that could not be solved regardless of the time limit were classified as hard. They include *gplus_500*, *pokec_500*, *netscience*, *Ecoli20160114CR*, *Hpylo20160114*, *Mmusc20160114*, *anna* and *homer*.

From these results we can speculate that the structure of the instance seems to be the main factor that influences the difficulty. *Anna* is a instance that has a low number of nodes, however it has been classified as a hard instance.

Social networks instances where quite interesting. The instances with 200 nodes where solvable with a time limit of 10, while the larger instances with 500 nodes where not solvable. The size of the instance may be a way of determining the difficulty of solving the instance, but it is not the sole factor.

In summary, by using the exact ILP program combined with the pipeline, we were able to find the optimal solutions for 14 instances.

In table 3.5, we include the results of the heuristic approach. We have split the heuristic into two different variants. Variant one is with perturbation operators one, two and three, which is under the column named MSLS-10000-III. Variant two is the heuristic with all four perturbation operators , which is represented by MSLS-10000-IV.

Each version of the heuristic was run over 10000 consecutive runs with many different starting nodes. We found that for some instances, operator 4 (MSLS-10000-IV) provides better results. However, the majority of instances had better results with perturbation operator 3 (MSLS-10000-III). We then extended the amount of consecutive runs to 100000. However, only perturbation operators 1 - 3 were used

due to the previous findings.

For 8 of the instances categorised as easy, the heuristic found optimal solutions. In these cases, both MSLS-10000-III and MSLS-10000-IV found the optimum. For the instance *Hsapi20160114HT*, MSLS-100000-III was able to find the longest cycle. For *adjnoun* and *polbooks*, only suboptimal solutions were found by the heuristic approach.

However, for the medium classified instances, the heuristic produced cycles with varied lengths. For two instances, *gplus_200* and *david*, the length of the found cycle where close to the optimum. For the instance *celegansneural* however, the found cycle was still 9 nodes shorter than the provable optimum.

In contrast to the easy instances, the cycles found by the heuristic for the hard instances were all suboptimal, even when compared to the solutions found by the exact solver. The only instance where this wasn't applicable was for *netscience*. The gaps between the lengths found by the heuristic and the exact solver varied by the instance. For the smaller instance, *anna*, the difference is 2. Whereas for *Hpylo20160114*, the difference is 58 nodes.

One instance that bucks the trend is *netscience*. The heuristic found a cycle of length 108, whereas the exact approach, even with a time limit of 1 hour, found only 107. This seems to be related to the structure of the graph, and is quite surprising. It is worth noting that the two heuristics MSLS-10000-III and MSLS-10000-IV where able to find the cycle of length 107 in minutes, rather than the week needed by the exact solver. This indicates that the heuristic can perform surprisingly well in some instances.

Table 3.4: A table to show the numerical timings of our ILP approach used with CBC. Three versions of the approach were used with time limits of 1 minute, 10 minutes and 1 hour per node for each instance. If a instance has been solved with a lower time limit, we have omitted the higher time limit results as the result would be the same, or similar in time. This strategy allowed us to categorise the instances to easy, medium and hard. Such a categorisation is based on whether the more restrictive versions of the exact approach were successful in finding the proven optimum or not. N/A indicates that no cycle has been found within the time limit.

graph	CBC (1 min / node)		CBC (10 min / node)		CBC (1 hour / node)	
	optimum	time	optimum	time	optimum	time
Social networks						
gplus_200	≥ 70	5117 s	70	30850 s		
gplus_500	≥ 8	16353 s	≥ 202	157858 s	≥ 206	942816 s
pokec_500	≥ 127	17485 s	≥ 163	168396 s	≥ 166	1003664 s
soc_52	51	15 s				
Graphs from Newman's network data repository						
adjnoun (Newman, 2006)	101	207 s				
football (Girvan and Newman, 2002)	115	4 s				
lesmis (Knuth, 1993)	49	490 s				
netscience (Newman, 2006)	≥ 88	25495 s	≥ 104	194185 s	≥ 107	868606 s
zachary (Zachary, 1977)	20	19 s				
celegansneural (Watts and Strogatz, 1998)	≥ 279	17959 s	280	2785 s		
dolphins (Lusseau et al., 2003)	53	≤ 1 s				
*polbooks**	105	60 s				
Protein-protein interactions from UCLA database of interacting proteins (Salwinski et al., 2004; Xenarios et al., 2000, 2001, 2002)						
Celeg20160114CR	6	≤ 1 s				
Dmela20160114CR	14	49 s				
Ecoli20160114CR	≥ 8	18380 s	≥ 242	179389 s	≥ 244	1068437 s
Hpylo20160114	N/A	26484 s	≥ 291	260322 s	≥ 299	1551768 s
Hsapi20160114HT	64	1959 s				
Mmusc20160114CR	≥ 170	22012 s	≥ 256	215912 s	≥ 267	1282218 s
DIMACS graphs (Johnson and Trick, 1996)						
anna	≥ 77	6490 s	≥ 79	64498 s	≥ 79	387458 s
david	≥ 72	4621 s	72	4714 s		
homer	≥ 114	18309 s	≥ 223	171157 s	≥ 234	1015438 s
huck	48	858 s				

* Network *polbooks* has not been published in a past paper. It is available from Newman's network data repository: `http://www-personal.umich.edu/~mejn/netdata/`.

Table 3.5: A table to show the results of the heuristic. We show Variant 1, which uses only perturbation operators 1,2 and 3 as MSLS-10000-III and Variant 2 which uses all 4 operators as MSLS-10000-IV. We run these both 10000 times. We find that Variant 1 out-performs Variant 2, and thus run 100000 trials of this variant, which is displayed as MSLS-100000-III. Their comparison to the proven longest cycle lengths (or their lower bounds) found by the exact approach based on CBC is included.

graph	CBC	MSLS-10000-III		MSLS-10000-IV		MSLS-100000-III	
		cycle length	time	cycle length	time	cycle length	time
Social networks							
gplus_200	70	67	65 s	66	50 s	68	729 s
gplus_500	≥ 206	186	606 s	186	560 s	192	6891 s
pokec_500	≥ 166	155	663 s	151	598 s	159	7158 s
soc_52	51	51	25 s	51	23 s	51	280 s
Graphs from Newman's network data repository							
adjnoun (Newman, 2006)	101	91	78 s	92	62 s	93	967 s
football (Girvan and Newman, 2002)	115	115	103 s	115	93 s	115	1246 s
lesmis (Knuth, 1993)	49	49	18 s	49	14 s	49	204 s
netscience (Newman, 2006)	≥ 107	107	531 s	107	524 s	108	5944 s
zachary (Zachary, 1977)	20	20	4 s	20	2 s	20	52 s
celegansneural (Watts and Strogatz, 1998)	280	270	2147 s	267	1960 s	271	27953 s
dolphins (Lusseau et al., 2003)	53	53	7 s	53	7 s	53	94 s
*polbooks**	105	104	73 s	103	65 s	104	802 s
Protein-protein interactions from UCLA database of interacting proteins (Salwinski et al., 2004; Xenarios et al., 2000, 2001, 2002)							
Celeg20160114CR	6	6	1 s	6	6 s	6	5 s
Dmela20160114CR	14	14	3 s	14	3 s	14	43 s
Ecoli20160114CR	≥ 244	207	894 s	205	849 s	211	11208 s
Hpylo20160114	≥ 299	239	1818 s	235	1674 s	241	21357 s
Hsapi20160114HT	64	63	22 s	63	17 s	64	253 s
Mmusc20160114CR	≥ 267	243	728 s	248	712 s	246	8353 s
DIMACS graphs (Johnson and Trick, 1996)							
anna	≥ 79	76	96 s	77	112 s	77	1316 s
david	72	71	50 s	70	45 s	71	608 s
homer	≥ 234	206	1508 s	205	1468 s	209	17981 s
huck	48	48	24 s	48	21 s	48	284 s

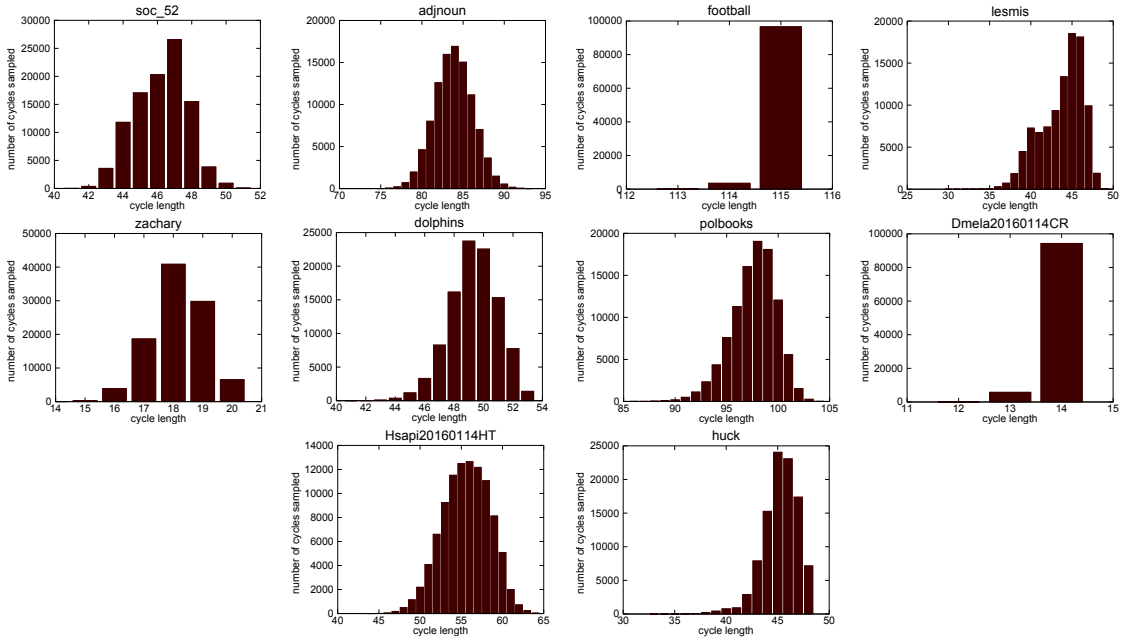

Figure 3.6: Distributions of cycle lengths sampled by the multi-start local search approach MSLS-100000-III for the easy instances.

3.5 Discussion

As the heuristic has a sequence of samplings of long simple cycles using DFS, we are able to plot the distribution of lengths sampled in each of these runs. We have explored these distributions for MSLS-100000-III.

Figure 3.6 illustrates the distributions we obtained for the easy instances. Most of the instances that fall into the easy category show that the peaks of the distributions are near the far end of the distributions. It seems that the probability of sampling a cycle which is longer than the peak declines quite rapidly. As such, we have measured the statistical properties of the distributions. These properties include the mean, standard deviation, skewness and kurtosis.

Table 3.6: Statistical properties of cycle length distributions obtained by MSLS-100000-III. This includes the longest cycle length found, the average cycle length μ, the standard deviation σ, the skewness γ_1 and the "excess" kurtosis γ_2 for each distribution.

graph	optimum	MSLS-100000-III				
		best	μ	σ	γ_1	γ_2
Social networks						
gplus_200	70	68	59.46	2.8421	-0.42985	-0.040028
gplus_500	≥ 206	192	161.33	7.2337	-0.031828	-0.083499
pokec_500	≥ 166	159	130.83	8.1637	-0.41967	-0.058592
soc_52	51	51	46.202	1.5434	-0.15057	-0.44118
Graphs from Newman's network data repository						
adjnoun (Newman, 2006)	101	93	83.852	2.3339	0.0088051	-0.066728
football (Girvan and Newman, 2002)	115	115	114.96	0.19147	-5.313	28.502
lesmis (Knuth, 1993)	49	49	43.772	2.5782	-0.67178	-0.27657
netscience (Newman, 2006)	≥ 108	108	92.438	5.4751	-0.21830	-0.25129
zachary (Zachary, 1977)	20	20	18.684	0.74029	-0.15538	-0.053173
celegansneural (Watts and Strogatz, 1998)	280	271	260.33	2.884	-0.11364	0.005146
dolphins (Lusseau et al., 2003)	53	53	49.323	1.6584	-0.30721	0.081956
polbooks	105	104	97.632	2.1877	-0.54051	-0.058592
Protein-protein interactions from UCLA database of interacting proteins (Salwinski et al., 2004; Xenarios et al., 2000, 2001, 2002)						
Celeg20160114CR	6	6	6	0	N/A	N/A
Dmela20160114CR	14	14	13.943	0.23244	-3.8124	12.548
Ecoli20160114CR	≥ 244	211	183.21	6.1459	-0.01982	-0.013013
Hpylo20160114	≥ 299	241	214.22	5.2453	0.23223	0.10657
Hsapi20160114HT	64	64	55.581	2.8587	-0.17028	-0.30979
Mmusc20160114CR	≥ 267	246	207.74	9.4577	0.077919	-0.074851
DIMACS graphs (Johnson and Trick, 1996)						
anna	≥ 79	77	67.081	3.1581	-0.41567	0.243
david	72	71	63.918	2.5469	-0.61604	0.54436
homer	≥ 234	209	183.03	6.457	-0.045365	-0.0058615
huck	48	48	45.273	1.6975	-0.85419	1.6703

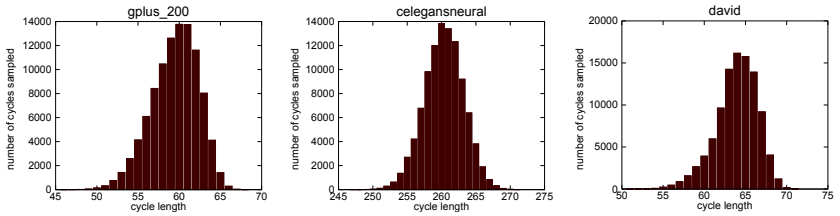

Figure 3.7: Distributions of cycle lengths sampled by the multi-start local search approach MSLS-100000-III for the medium difficulty instances.

Table 3.6 presents these properties for each distribution X obtained for each graph. The columns of the table represent the optimum, the best result obtained by MSLS-100000-III, the mean (μ), the standard deviation (σ), the skewness ($\gamma_1 = E[(X-\mu)^3]/\sigma^3$) and the "excess" kurtosis ($\gamma_2 = E[(X-\mu)^4]/\sigma^4 - 3$).

The table shows that distributions with a low value of γ_1 and a high value of γ_2 indicate an easy instance. However, *adjnoun* has a low value for γ_1 and γ_2, which seems to buck this trend. Since *adjnoun* was the only easy instance for which MSLS-100000-III was relatively far away from the optimum, a distribution close to a normal distribution may indicate that the heuristic is less efficient. It is also worth noting that apart from *adjnoun*, $\gamma_1 < 0$ for all easy instances.

On a related note, the distributions seem to differ from instance to instance. The profile for *polbooks* suggests that sampling of the longest cycle may be possible with an increased number of runs.

Following on, figure 3.7 shows the distributions obtained for all of the medium instances. The plots for these seems to be more fine grained than the easy instances. For all of these instances we have that $\gamma_1 < 0$. Similar to the easy instances, there is an outlier to this. Instance *celegansneural* has a γ_1 and γ_2 close to 0. In similarity to

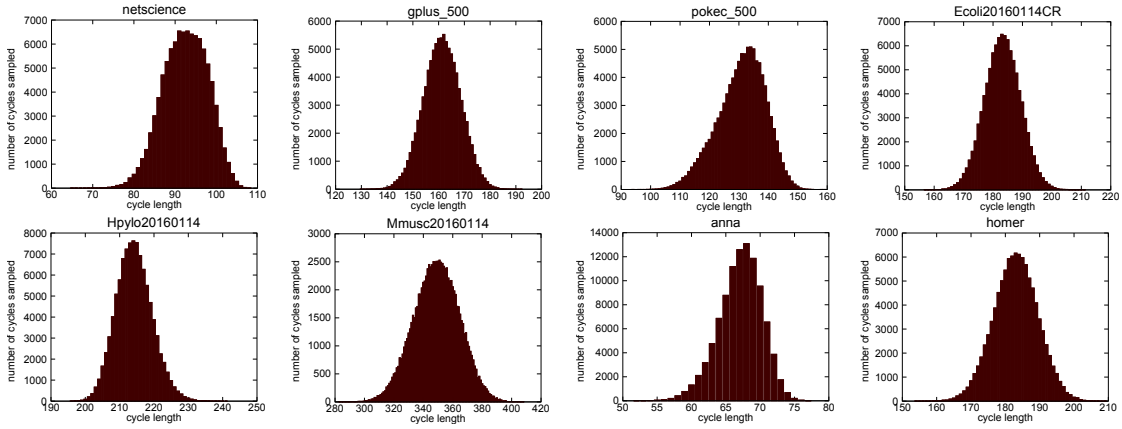

Figure 3.8: Distributions of cycle lengths sampled by the multi-start local search approach MSLS-100000-III for the hard instances.

adjnoun, the heuristic has not managed to find the optimum for this instance. This indicates that a more sophisticated approach may be more suitable for this type of difficulty.

A further avenue of exploration could be combining the heuristic with swarm intelligence as evolutionary algorithms as a tool to overcome the optima found. It is also worth noting that the distributions obtained by different heuristics can be used as a promising tool to compare the efficiency of different approaches.

These algorithms could also be beneficial for hard instances. We show the distributions of the hard instances in figure 3.8. The distributions again seem to be fine-grained, however there is an outlier in instance *anna*. The cycles found for hard instances are quite moderate. The results of the exact approach indicate that providing significantly longer cycles seems to require quite extensive computational efforts.

An item of note is that the perturbation operators contributed to the final long cycle found quite significantly. When DFS was used as an initialisation, the cycles found were of a moderate size. Perturbation operator 1 seems to be at its most advantageous when the instances have a high community structure. Whereas perturbation operator 2 is more advantageous for instances that have an underlying structure of a grid. Operators 3 and 4 were used to expand the search space of the DFS algorithm to find more nodes. We found that for majority of the instances, perturbation operator 3 was more efficient in finding the longest cycle.

3.6 Conclusion

In this chapter we have proposed two approaches to find the longest simple cycle in instances. The first approach is an exact solver based on the fixed starting node. This is an ILP formulation combined with our data mining pipeline. This pipeline solves the problem by sequencing different ILP problems for different fixed initial nodes. Our experiment results have shown that our approach outperforms a previous ILP formulation of the longest simple cycle problem. The approach can be extended to find a probable longest cycle for larger instances, while being able to solve smaller instances.

The second approach is a DFS-based heuristic. We create an initial long cycle through a DFS search. This initial cycle is then improved with four perturbation operators. We show that multi-start versions of the heuristic provide relatively long cycles in a mostly shorter time in comparison to the exact solver. For easy

and medium instances, the heuristic was able to produce optimal or near-optimal solutions to the longest cycle problem. Where the heuristic shines is with larger instances. The exact solve can require weeks, or months of computation to solve large instances, whereas the heuristic is able to produce a non-optimal result in a fraction of the time. For these instances the heuristic approach is much more valuable for real-world applications.

The study of long cycles in this chapter can form a basis for further exploration of the problem area. Applicational use in social and biological is one such area. For large scale instances of the problem new heuristics combining different ideas and operators may contribute to develop more scalable approaches suitable for finding even longer cycles.

One such area of exploration is introducing evolutionary algorithms to the problem. An ant-based evolutionary search improvement was introduced, and thus the work in this chapter has been built upon by the authors in (Chalupa et al., 2018).

Further avenues of exploration of this chapter include topics such as layout algorithms for social and biological networks (Becker and Rojas, 2001; Girvan and Newman, 2002), drawing of planar graphs (Tamassia, 2013) or the closely related longest path problem (Karger et al., 1997; Uehara and Uno, 2007).

In summary this chapter has laid down the fundamental ideas for practical algorithms which can be used to find long cycles in real-world networks.

First learn computer science
and all the theory. Next develop
a programming style. Then
forget all that and just hack.

George Carrette

CHAPTER 4

An exploration of graph databases by implementing the Minimum Dominating Set problem

As discussed in chapter 2, many different application areas are finding relational databases to be limited in a number of different areas. As graph databases focus on graph data (Yannakakis, 1990), the opportunity to implement and apply graph algorithms within these databases has occurred.

This graph-like structure allows the implementation of various computational algorithms that would have been difficult to implement into a relational database. These algorithms can be created using query methods provided by graph database systems, or through a combination of a high level language and query methods. In

this chapter we take a computationally demanding problem, such as the Minimum Dominating Set and implement an unoptimised greedy heuristic into two different graph database systems. This chapter builds upon the work covered in (Balaghan et al., 2017).

The main contribution in this chapter is a novel exploration of graph database systems when the minimum dominating set algorithm is implemented. This shows how efficient the back-end of the graph database systems are in relation to the implementation of the problem and the limitations of query methods provided by these systems. With these limitations, workarounds may need to be applied. The impact of these workarounds on timings or on efficiency of the systems are evaluated.

Section 4.1 introduces the topic of discussion, followed by a introduction to the systems used in the experiments in section 4.2. Some repetition of previous information given in chapter 2 may occur in those sections. Sections 4.3 and 4.4 give an in-depth evaluation of the Minimum Dominating Set problem by providing three different implementations. Finally, results and further discussions are given in sections 4.5 and 4.6 which are then concluded in 4.7.

4.1 Introduction

Classic relational databases use the relational data model. This model stores data in the form of records in stores. Relationships between objects are between primary and foreign keys. If a relationship between two tables exists, a *JOIN* is used. *JOIN* commands are typically very computationally inefficient. Ironically, relational

databases found querying data which required a lot of relations are very computationally expensive and sometimes infeasible (Robinson et al., 2013).

Storing data in a graph-like structure was explored in the 1980's, where a selection of data models were introduced (Angles and Gutierrez, 2008, 2005). This tended to slow down in the 1990's, with the focus being on relational databases. But once the limitations of relational databases were realised, storing data in a graph-like form became more popular. Big companies like Google, Facebook and others noticed this limitation and created their own graph-like data storage (Bronson et al., 2013; Malewicz et al., 2010). At the same time, graph gatabase systems were forming.

There has been some comparison between relational and graph databases (Vicknair et al., 2010; Jouili and Vansteenberghe, 2013; Robinson et al., 2013). They generally show that graph databases are more efficient in simple relation queries. These are queries that focus on "hops". For example, searching the neighbourhood of a node, or the neighbourhood of the neighbourhood of the node.

Graph databases have also found many different applicational areas. They have been used to find terrorist networks (Gutfraind and Genkin, 2016), rogue behaviour detection (Castelltort and Laurent, 2016) and identify users with limited data (Vesdapunt and Garcia-Molina, 2015). And more recently, have been used to analyse the Panama Papers that have been released (Neo4j, 2017a).

The graph structure of graph databases opens up the possibility of performing computationally complex algorithms on the database. These could include finding long cycles (Chalupa et al., 2017), node failure (Hawick and James, 2007; Hawick, 2012b), Betweenness Centrality (Freeman, 1977, 1978) and many others.

The most prominent model used by graph databases is the property graph model. The property graph model has been built upon a selection of previous graph models, which have been explored in detail by Angles (Angles and Gutierrez, 2008).

It is worth noting the increase in popularity of Multi-Modal databases. Multi-modal databases support either two or more database models while keeping a single integrated backend. An example of this would be OrientDB which combines a graph database with a document database (OrientDB, 2017). A selection of systems which encompass graph and multi model databases have been explored in Lissandrini et al. (2017). In this paper we focus on "pure" graph databases.

The contributions that can be found in this chapter are that we evaluated the general efficiency of implementing the dominating set algorithm into the different types of databases. From this, we found that the methods of querying provided by the database systems does not provide the means to efficiently implement the algorithm. We also found that the expressiveness of the graph database languages are not as concise and expressive as expected. We evaluated the contributions by implementing an unoptimised greedy heuristic of the Minimum Dominating Set problem into graph database systems.

4.2 Graph database systems

As previously discussed , graph database systems can typically be sorted into two different types of models. The first model is a client-server model. A *client* typically sends a query request to a *server*. An example of such systems would be Neo4j

(Neo4j, 2017c) and OrientDB (OrientDB, 2017).

At the time of writing, Neo4j (Neo4j, 2018) is the most popular client-server graph database according to a database ranking website (DBEngine, 2017b,d). It is an open-source JVM-based database system. It fully supports Atomicity, Consistency, Isolation and Durability (ACID) transactions.

The second model is an embedded model. The database is created and stored locally, typically through library calls in a high-level language. An example system of which would be Sparsity. Sparsity is a system written in C and C++.

Sparsity is one of the more popular embedded graph databases. It began life as DEX (Martínez-Bazan et al., 2007) where it then re-branded as Sparsity. It does not currently have a dedicated query language, instead the database can be queried by using some of the in-built library function when combined with a high-level language. It also fully supports ACID transactions, with the ability to disable if necessary.

Each of the different models have provided different methods to query a database. Client-server databases typically provide a query language. In contrast, embedded databases tend to provide a selection of library calls which act as query replacements.

At the time of writing, a formalised query language for graph databases does not exist. This is because querying graph patterns is computationally hard (Barceló et al., 2011) and thus querying graph databases are also computationally hard (Barceló Baeza, 2013). Although there have been attempts at countering this (Lee and Chung, 2014), there are some unavoidable overhead when querying graphs (Stonebraker and Cattell, 2011). There have been some methods to efficiently query subgraphs (Zheng et al., 2014), but the problem still remains hard.

In response to this, a selection of different query languages have been created. These include G (Cruz et al., 1987), G+ (Cruz et al., 1988), Gram (Amann and Scholl, 1992), Gremlin, Cypher, openCypher (Marton et al., 2017), SLQ (Yang et al., 2014), ProGQL (Tausch et al., 2011), PGQL (van Rest et al., 2016), GOQL (Sheng et al., 1999), GraphQL (He and Singh, 2008) and others (Angles and Gutierrez, 2005). A survey on the functionality of these has been explored by (Wood, 2012). There has also been an attempt to compare the languages, however the study is now quite dated (Holzschuher and Peinl, 2013). It is worth noting that there has been some attempt at formalising a query language, such as openCypher (Marton et al., 2017) and SPARQL (Seaborne and Prud'hommeaux, 2008). These languages have either not been or have partially been adapted by graph database systems.

Neo4j only officially supports a single query language. This language is called Cypher. Cypher is a declarative scala-based language. Cypher has been proven to be quite versatile, being used as a back-end of a DSL for querying source code data (Urma and Mycroft, 2015).

An example of a basic Cypher query is given in algorithm 14. Within the algorithm, keywords are shown in capitals. Optional clauses are surrounded by square brackets. To begin a query a *MATCH* clause is required. This is then followed by a pattern that represents what is being queried. An optional *WHERE* clause can be given to restrict the query. The final clause of a query is the *RETURN* clause. This can also be limited by the optional clause *LIMIT* and sorted by the *ORDER BY* clause. Cypher queries were covered in more detail in chapter 2.

The Cypher query can be "extended" to include multiple *MATCH* clauses by

using the optional clause *WITH*. The *WITH* clause allows the user to store variables from the previous *MATCH* clause. After the *WITH*, a *MATCH* clause can then be used to start a new query.

Algorithm 14 A Cypher query in its most basic form. Items in capital letters are key words. Optional clauses are surrounded by [ ]. Adapted from (Drakopoulos, 2016).

```
1: MATCH <pattern>
2: [WHERE <restriction>]
3: RETURN <expression> | <pattern>
4: [ORDER BY <pattern> [DESC/ASC]]
5: [LIMIT <number>]
6: [WITH <variables>]
```

As a representative of client-server databases, we have used Neo4j and as a representative of embedded databases, we have used Sparsity. There has been some previous comparisons between these two databases. One such study focused on the scalability of the systems by implementing the HPC scalable graph analysis benchmark (Dominguez-Sal et al., 2010), which found Neo4j and DEX (now Sparsity) to be the most efficient. As well as this, there has been an attempt to benchmark both DEX and Neo4j (Macko et al., 2013) for simple graph procedures.

4.3 The Minimum Dominating Set problem

While some repetition of sub-section 2.4.5 may occur in this section, the Minimum Dominating Set problem is explored in more detail in this chapter. To begin, we give some definitions of G. Let $G = (N,E)$ be either an undirected or a digraph where N is a non-empty set of nodes and E is a set of edges subject to N x N.

Definition 4.3.1. A dominating set in a graph G is a subset S of nodes such that every node in G is either in S or is adjacent to a node in S (Gross et al., 2013). A node is said to dominate itself, as well as its adjacent nodes. The dominating set of a graph is referred to as $\gamma(G)$.

Definition 4.3.2. The neighbourhood of a node n is the subset of nodes in graph G which are directly adjacent to node n. In a digraph, this is the out-set of a node n (i.e. all out-going edges for n).

Definition 4.3.3. An isolated node is a node without any edges. It has a degree of 0.

Finding the minimum dominating set of a graph is a classical Non-deterministic Polynomial-time Completeness (NP-Complete) problem (Garey and Johnson, 1990). An efficient solution to finding the optimum minimum dominating set of a graph is currently not available. However, some heuristics exist that can find a minimal dominating set, but not the guaranteed optimum. One such heuristic is the greedy algorithm (Chvatal, 1979). This is defined in algorithm 15.

Algorithm 15 The greedy heuristic algorithm for finding a minimum dominating set of a Graph

1: $S := \emptyset$
2: **while** $\exists$ *white nodes* **do**
3: *choose* $v \in \{x | w(x) = max_{u \in V}\{w(u)\}\}$
4: $S := S \cup v$
5: **end while**

To begin with, all nodes in the graph are initially coloured white. After this, a calculation is run to find the node with the most white-node connections, including

itself. This node is then coloured black, and each of its adjacent nodes are coloured grey. This is repeated until no white nodes exist. Finally, all of the nodes which are black are the final dominating set. Algorithm 16 gives an implementation in a high level language of the heuristic. Any isolated nodes are also eventually coloured black.

The dominating set algorithm has been useful in many different application areas. They have been used to find positive influence in social networks (Dinh et al., 2014), as well as efficient routing in wireless networks (Wu and Li, 1999).

One difference to the greedy heuristic in the high level implementation is that all isolated nodes are coloured black at the beginning of the algorithm's run. This allows some time to be saved, and less nodes to be processed as the nodes that are isolated and their neighbourhoods are already coloured and are therefore in the dominating set.

4.4 Implementing the Minimum Dominating Set problem into graph database systems

We implement the greedy heuristic algorithm using the query methods provided by Sparsity and Neo4j. For Neo4j we use the query language Cypher, and for Sparsity we use libraries supplied with the API. As well as that, we implement the high-level algorithm 16 in C++ as a direct comparison.

The Cypher query in algorithm 17 shows how to run one iteration of the greedy algorithm. In lines 2-4, we are setting up the graph by initialising every node to white.

Algorithm 16 A high level language implementation of the greedy heuristic algorithm. In the end, every node with weighting 1 is a grey node, and every node with weighting 0 is a black node.

```
1: function DominatingSet(Graph g, nodeList[])
2:     blackNodes[]
3:     whiteNodes[] ← nodeList[]
4:     weightings[] ← 0
                                        ▷ Find all Isolated nodes and make them black
5:     for all node in whiteNodes do
6:         if deg(node) < 1 then
7:             blacksNode.add(node)
8:             whitesNodes.remove(node)
9:             weighting[node] ← 0
10:        else
11:            weighting[node] ← 2
12:        end if
13:    end for
14:    while whiteNodes.size > 0 do
15:        highestweight ← 0
16:        curHighNode ← null
17:        for all node in nodeList do   ▷ Find the node with the most white-node
   connections
18:            if weightings[node] != 0 then                    ▷ If not a black node
19:                totalweight ← weightings[node]
20:                for all neighbour n of node do
21:                    totalweight += weightings[n]
22:                end for
23:                if totalweight > highestweight then
24:                    highestweight ← totalweight
25:                    curHighNode ← node
26:                end if
27:            end if
28:        end for
29:        blackNodes.add(curHighNode)
30:        whiteNodes.remove(curHighNode)
31:        for all (neighbour n of curHighNode) do
32:            weighting[n] ← 1
33:            whiteNodes.remove(n)
34:        end for
35:    end while
       return blackNodes[]
36: end function
```

Algorithm 17 The greedy heuristic algorithm in Cypher - One Step. At the end, every node with blackness = 1 is in the dominating set, and every node with whiteness = 0 and blackness = 0 is a grey node.

```
1: function CYPHERGreedyAlgorithm
2:     MATCH(h)
3:     SET h.whiteness = 1
4:     SET h.blackness = 0
5:     WITH h
6:     OPTIONAL MATCH(j)        ▷ Find all Nodes without any Edges(i.e. with a
   degree of 0)
7:     WHERE NOT (j)–>()
8:     SET j.blackness = 1
9:     SET j.whiteness = 0
10:    WITH j
11:                             ▷ This is where the repeated part of the query begins.
12:    MATCH (n)–>(m)
13:    WHERE n.blackness <> 1
14:    WITH collect(m) as neighbourhood, n
15:    WITH reduce(totalweight = n.whiteness, j in neighbourhood — totalweight
   + j.whiteness) as weightings, n
16:    WITH n, weightings
17:    ORDER BY weightings desc limit 1
18:                               ▷ This is where the repeated part of the query ends.
19:    MATCH(n)–>(m)
20:    WHERE m.blackness <> 1
21:    SET n.blackness = 1
22:    SET n.whiteness = 0
23:    SET m.whiteness = 0
24:    WITH n
25:    MATCH (k)
26:    WHERE k.whiteness = 1
27:    RETURN count(distinct(k)) as countOfRemainingWhiteNodes
28: end function
```

Then an *OPTIONAL MATCH* is used to find any isolated nodes. The *OPTIONAL MATCH*, as opposed to a *MATCH* is required as the query can return a NULL value should the graph be a fully connected graph.

The "bulk" of the query begins from line 11. At first, every node that is not currently black is found. Then by using the in-built *reduce* function in Cypher, we are able to find the most white neighbours. The nodes are then sorted by the weightings found in line 16. Finally, by using this node, we are then able to find its adjacent neighbours in lines 17 and 18.

An issue we encountered when implementing the algorithm into Cypher was the inability to iterate a part of the query. In order for the heuristic to find a dominating set for the graph, iteration is required for the query in lines 11 to 21. This must be repeated until all nodes in the graph are either black or grey. The only iterator that we are aware of is the in-built *FOREACH* function. However this function can not be paired with any *MATCH* query.

Because of this, we created a pipeline which would build a single Cypher query that could find the dominating set. As far as we are aware, the length of a Cypher query is limited by the size of a string. By using the process shown in fig 4.1, we were able to create this Cypher query. We used Python as a method of building this query.

The original query in algorithm 17 is split into three separate queries. The first of which is the "set-up" graph query. This encapsulates lines 2 to 10. This would only need to be run once per query, as it colours all nodes white and then colours all isolated nodes black. The "bulk query" encapsulates lines 11 to 22. And finally the

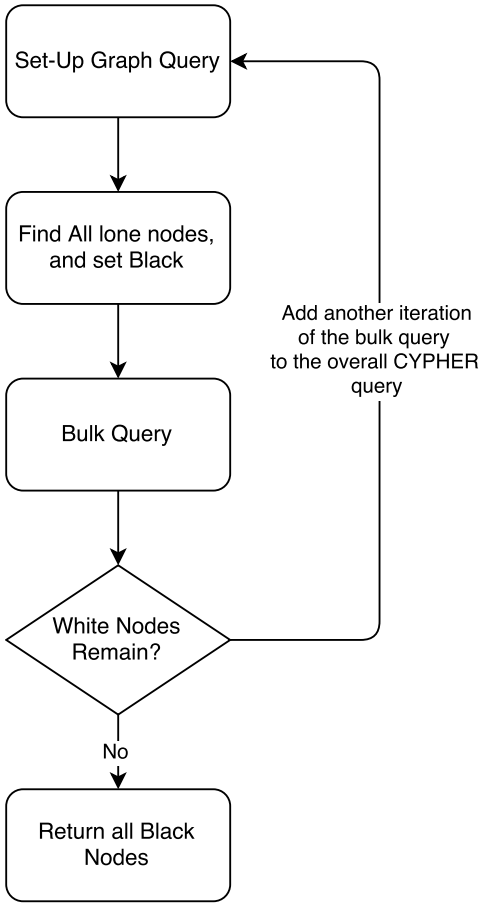

Figure 4.1: A diagram showing how the one Cypher query is built. A first iteration of the greedy algorithm is run. If any white nodes remain, then another iteration of the bulk query is added onto the original query, and the whole query is re-run.

"end" query is the query to find how many white nodes remain in lines 23 to 25.

In order to build the single Cypher query we would first run the "set-up" query. Then the "bulk" query and finally the "end" query to check to see if any white nodes remain. If any white nodes remain, we add an iteration of the bulk query to the overall query. It should be noted that the variable names are changed when the bulk query is added. For each variable in the query, such as (n) or (m), its name is changed to (n(n+1)) or (m(n+1)) (i.e. (n1),(m1),(n2),(m2).....(nn)(mn)). This is to ensure that the previous variables in the query are not effecting the new query. This is repeated until no white nodes remain. An example final cypher query can be found in algorithm 18.

This has the potential to be a computationally costly program. Therefore we decided to create a wrapper in a high level language that encapsulates around the "bulk query". We would essentially be adding something similar to a for loop.

To do this, we split the query in algorithm 17 into two sections. We ran the "set-up" query, which was from lines 2 to 9. As the query sets up the graph by setting all nodes to white, and finds all nodes which are isolated and sets them black, it only needs to be run once. Then the "end" query, which essentially finds any remaining white nodes from lines 23 to 25. The next section is the iterated section, from lines 11 to 25.

It is worth noting that a final query, as shown in algorithm 19, returns a list of the final dominating set. This is run once there are no remaining white nodes.

In comparison, we then replicated the same tests in Sparsity. Sparsity uses an imperative query language. All of the query functionality are defined in different

Algorithm 18 An example final Cypher query for finding the minimum dominating set.

```
1: function CYPHERGreedyAlgorithm
2:     MATCH(h)
3:     SET h.whiteness = 1
4:     SET h.blackness = 0
5:     WITH h
6:     OPTIONAL MATCH(j)        ▷ Find all Nodes without any Edges(i.e. with a degree of 0)
7:     WHERE NOT (j)–>()
8:     SET j.blackness = 1
9:     SET j.whiteness = 0
10:    WITH j
11:    MATCH (n)–>(m)
12:    WHERE n.blackness <> 1
13:    WITH collect(m) as neighbourhood, n
14:    WITH reduce(totalweight = n.whiteness, j in neighbourhood — totalweight + j.whiteness) as weightings, n
15:    WITH n, weightings
16:    ORDER BY weightings desc limit 1
17:    MATCH (n1)–>(m1)
18:    WHERE n1.blackness <> 1
19:    WITH collect(m1) as neighbourhood, n1
20:    WITH reduce(totalweight = n1.whiteness, j in neighbourhood — totalweight + j.whiteness) as weightings, n1
21:    WITH n1, weightings
22:    ORDER BY weightings desc limit 1
23:    MATCH(n1)–>(m1)
24:    WHERE m1.blackness <> 1
25:    SET n1.blackness = 1
26:    SET n1.whiteness = 0
27:    SET m1.whiteness = 0
28:    WITH n1
29:    MATCH (k)
30:    WHERE k.whiteness = 1
31:    RETURN count(distinct(k)) as countOfRemainingWhiteNodes
32: end function
```

Algorithm 19 The final Cypher query to find all black nodes in the graph

```
1: MATCH(n)
2: WHERE n.blackness = 1
3: RETURN n
```

libraries, which can be called in-line.

Algorithm 20 Setting up the graph in Sparsity

```
1: function SetUpGraph(Graph g, int EdgeID, nodeIDs[])
2:     whiteNode = new Attribute()
3:     blackNode = new Attribute()
4:     for all id in nodeIDs do
5:         degree = g.degree(id, EdgeID, EdgeDirection.Any)
6:         if degree > 0 then
7:             g.setAttribute(id, whiteNode, 1)
8:             g.setAttribute(id, blackNode, 0)
9:         else
10:            g.setAttribute(id, whiteNode, 0)
11:            g.setAttribute(id, blackNode, 1)
12:        end if
13:    end for
14: end function
```

The equivalent queries in a single Cypher cannot be replicated using just the libraries in Sparsity. Therefore, we implemented a direct comparison to the Cypher and wrapper. We combined a high level language and the equivalent query calls to Cypher in Sparsity. Algorithm 20 gives the equivalent "set up" queries used in Algorithm 17. We assume the Sparsity graph object is called *g*.

Each node, edge and attribute type is given a unique ID. In algorithm 20, we first set the attributes of the nodes to white and black depending on whether they are an isolated node.

In order to find the node with the most white node connections, we use the

function defined in algorithm 21. The wrapper language holds the values of the current HighestNode and HighestWeight. All of the current white nodes are stored into a Sparsity object on line 4. The *g.select* call is used, where the white node attribute we created at the beginning of the function limits the query call.

On line 6, we check to see whether the current node is already in the dominating set. A *g.getAttribute* call is used to find the value of the selected attribute. After this, we create another Sparsity object that stores the neighbours of the current node. A *g.neighbours* call is used.

To find the total weight of white nodes, an *object.intersection* call can be used by finding the intersection between the *neighbourOfNodes* and *AllWhiteNodes* such as on line 9 in algorithm 21.

The C++ implementation is the same as defined algorithm 9. To ensure accuracy of the C++ implementation, we compare the results of the implementation with the same datasets in other literature, such as in (Chalupa, 2017). It is worth noting that other, more efficient approximation algorithms for finding the dominating set exist. These include an order based algorithm (Chalupa, 2017), a hybrid genetic and local search algorithm (Hedar and Ismail, 2010), an ant colony optimisation (Potluri and Singh, 2013) and many others.

We focus on the unoptimised greedy algorithm as the optimisations required by these algorithms may themselves be a challenge for the query methods. Therefore, the c++ implementation is also the unoptimised greedy algorithm to allow for a fair comparison.

In the following paragraph we define terminology to be used in this section.

Algorithm 21 Find the node with the most connected white nodes in Sparsity

```
1: function GetWeightings(Graph g, int BlackNodeAtt, int WhiteNodeAtt, int
   edgeID, nodeIDs[])
2:     HighestNode ← ∞
3:     HighestWeight ← 0
4:     AllwhiteNodes = g.select(WhiteNodeAtt, Condition.Equal, true)
5:     for all id in nodeIDs do
6:         isBlack = g.getAttribute(id, BlackNodeAtt)
7:         if isblack == false then
8:             neighbourOfNodes = g.neighbours(id,edgeID,EdgeDirection.Any)
9:             long weight = neighbourOfNodes.intersection(AllWhiteNodes)
10:            val = g.getAttribute(i, WhiteNodeAtt)      ▷ Checks the weight of the
   current node
11:            if val = true then                          ▷ If White, add 1 to it's weighting
12:                weight++
13:            end if
14:            if weight > HighestWeight then
15:                HighestWeight ← weight
16:                HighestNode ← id
17:            end if
18:        end if
19:    end for
       return HighestNode
20: end function
```

Building a Cypher query is the process whereby the pipeline builds a long cypher query in order to solve the instance. The *built* query is the final query from this process. *Cypher with a wrapper* is where the iterative part of the cypher query is instead replaced with a *for* loop in a high level language. Similarly, *Sparsity with a wrapper* is also where high level language featrures are used in conjunction with the sparsity query calls. The *c++* implementation is an implementation of the unoptimised greedy heuristic.

4.5 Results

We ran four varieties of the greedy heuristic. We created a Python program that creates a single Cypher query by following the process in fig 4.1. For the iterative Cypher program, we created a wrapper in Python. For the implementation in Sparsity we used Java as the high level language. And finally we also implemented the algorithm into C++ to represent the high level language.

These algorithms have been run on a selection of real-world graphs as well as a selection of modelled graphs based on the Barabási-Albert model (Barabási and Albert, 1999). We used the preferential attachment property, and gave each graph either 100, 1000 or 10000 nodes. The initial seed number is always 2. The 2, 3 and 4 represent the amount of edges added each time step. The real world graphs include a selection of real-life snapshots of real-world social networks (Chalupa, 2017; Chalupa et al., 2017).

Table 4.5 shows the results of the different runs. The first column shows the

amount of time taken for the program to create the single Cypher query. The second column shows the amount of time taken for the final single query to run. The third column shows the time taken for the Cypher with wrapper, the fourth column shows the time taken for Sparsity with wrapper and the final columns shows the equivalent in C++. N/A represents the database crashing or being unable to complete the query due to any restrictions.

The experiments were run on a Mac Pro with a 3.5GHz 6-Core Intel Xeon E5 Processor. We used Neo4j 3.2 Alpha 8 Enterprise Edition. The client requests were sent from an internal machine to limit the effect of network lag. For Sparsity we used version 5.2 with a High and Large accessibility license. For all of the implementations, we took an average of three runs.

The results in table 4.1 show that when building a single Cypher query, 5 out of the 22 instances could not be solved due to a database error in Neo4j. The database error occurs due to the database running out of heap memory. However, once a wrapper is introduced, this is reduced to only 3 instances. The two instances that were solved by the introduction of a wrapper were *Barabasi_1000_2*, and *gplus_ 2000*. *pokec_2000* is a similar graph size, but a dominating set could be found without a wrapper.

Figure 4.2 has two plots that show the results of all the experiments. The left hand plots the raw data, whereas the right hand plots the data to a log scale in terms of time taken. The plots have been sorted by the time taken by the high-level language. In the plots the red line represents the last single Cypher query, the yellow line represents the Cypher with a wrapper, the blue line represents Sparsity with a

Table 4.1: The results show how long it took to run one instance of the greedy algorithm to find the dominating set of a graph. The first column named "Cypher" is how long it took to build the single Cypher query. The second column is how long it took the single Cypher query to run. The third column is Cypher with a wrapper. The fourth column is Sparsity with a wrapper. And the final column is a high-level language implementation of the heuristic.

Data Set	Cypher without wrapper		Cypher	Sparsity	C++
	Whole Program	Final Query	with wrapper	with wrapper	Implementation
Barabasi_100_2	16.05s	1.28s	0.45s	0.03s	<1ms
Barabasi_100_3	7.27s	0.63s	0.43s	0.02s	<1ms
Barabasi_100_4	6.27s	0.89s	0.46s	0.02s	<1ms
Barabasi_1000_2	N/A	N/A	20.87s	2.28s	0.1s
Barabasi_1000_3	9305.22s	245.01s	23.1s	1.74s	0.01s
Barabasi_1000_4	5696.14s	63.91s	27.24s	1.42s	0.01s
Barabasi_10000_2	N/A	N/A	N/A	238.21s	8.94s
Barabasi_10000_3	N/A	N/A	N/A	188.52s	9.52s
Barabasi_10000_4	N/A	N/A	N/A	157.77s	10.04s
adjnoun(Newman, 2006)	20.54s	1.48s	1.17s	0.03s	<1ms
anna(Johnson and Trick, 1996)	31.46s	3.35s	1.36s	0.02s	<1ms
Dolphins (Lusseau et al., 2003)	3.05s	0.22s	0.22s	0.01s	<1ms
polbooks	11.91s	1s	0.55s	0.02s	<1ms
homer (Johnson and Trick, 1996)	7108.54	92.77	42.58s	0.57s	0.05s
huck (Johnson and Trick, 1996)	7.98s	1.24s	0.52s	0.01s	<1ms
lesmis (Knuth, 1993)	3.89s	0.66s	0.29s	0.01s	<1ms
soc52 (Chalupa, 2017)	0.28s	0.28s	0.27s	<1ms	<1ms
pokec_500 (Chalupa, 2017)	122.21s	9.87s	5.10s	0.1s	<1ms
pokec_2000 (Chalupa, 2017)	11175.53s	198.71s	91.89s	1.90s	0.13s
gplus_200 (Chalupa, 2017)	27.86s	1.89s	1.16s	0.05s	<1ms
gplus_500 (Chalupa, 2017)	363.64s	11.21s	5.76s	0.25s	0.01
gplus_2000 (Chalupa, 2017)	N/A	N/A	94.83s	4.26s	0.26s

wrapper and finally the black line represents the high level language implementation.

The first item of note is that pure Cypher queries take longer to complete than any other method. This could be due to the overheads and declarative style of the language which itself brings in overheads that Sparsity and C++ implementations do not have. As well as that, not using a wrapper causes the query to struggle with instances that had more than 2000 nodes, failing on *gplus_2000*, *barabasi_1000_2*, *barabasi_10000_2*, *barabasi_10000_3* and *barabasi_10000_4*.

Once a wrapper has been introduced, it is still quite inefficient when compared to C++ and Sparsity. With the addition of the wrapper however, more instances were able to be solved. This could be due to the memory build up with the larger instances being severely reduced by the introduction of a loop. As well as that, the database may be more optimised to work with a quick succession of short queries, rather than a single long one. *barabasi_1000_2* and *gplus_2000* were solvable with the introduction of a wrapper. Noticeably, the bigger *barabasi_10000* instances could not be solved due to the system running out of cache space in the JVM.

To follow on from that, the plot in figure 4.3 shows only the graphs that took the C++ implementation <1ms to complete. The key is the same as the key in figure 4.2. Cypher and Sparsity take more time to find the dominating set than the C++ implementation, apart from the instance *soc52* in which Sparsity equals C++. This could be due to the overheads the databases invariably have in comparison to a vanilla implementation.

It is also worth noting that for the *dolphins*, *soc52* and *adjnoun* instances, Cypher as a single query and Cypher with a wrapper both find the dominating set in a very

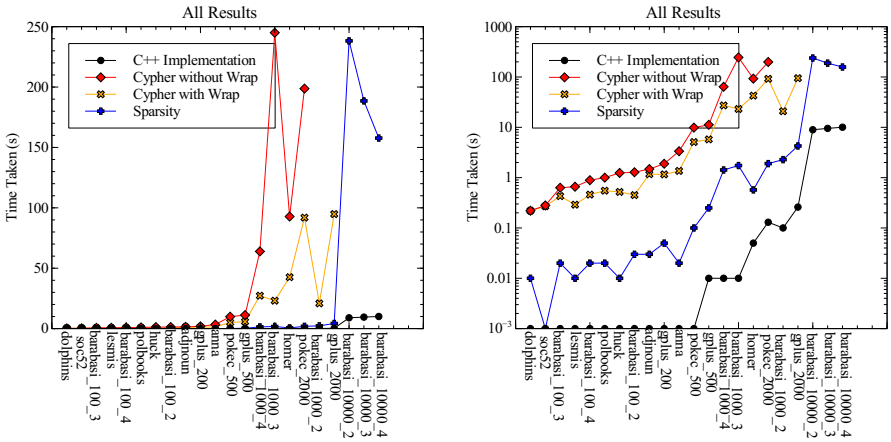

Figure 4.2: The above plots show all of the results combined and sorted by the time taken in C++. The plot on the left are the raw results, whereas the plot on the right shows the results in a log-log scale.

similar time. This shows that for smaller instances, introducing a wrapper does not effect the time taken.

In general, the single Cypher query is rather inefficient when finding the minimum dominating set, even with the introduction of a wrapper. When Sparsity is compared to the C++ implementation, it is slower. This was to be expected as the multiple over-heads which come with a graph database may effect the time it takes to query the graph. For the instance *soc_52*, Sparsity and C++ had the same time taken, but for all other instances it was slower.

4.6 Discussion

Table 4.2 gives a summary of the implementations when compared to the size of the instances.

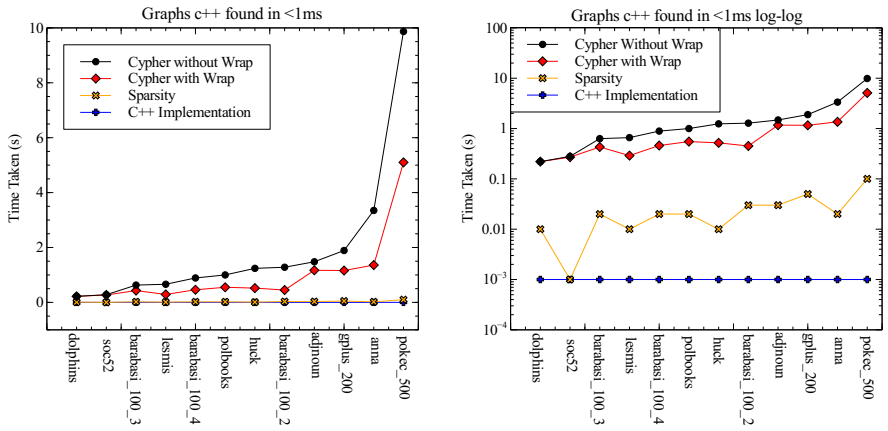

Figure 4.3: The above plots show the graphs that took <1ms in C++ to complete. The plot on the left shows the raw results, whereas the plot on the right shows the results to a log scale.

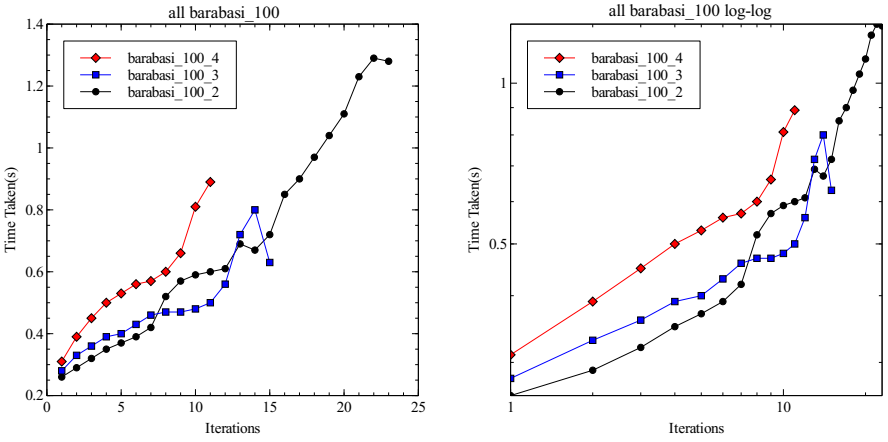

Figure 4.4: A comparison of the three different barabasi_100 instances and the iterating time taken to find a dominating set with the Cypher and wrapper. The left plot shows the raw results, whereas the right plot shows the results to a log-log scale. Each iteration is the addition of a node into the dominating set.

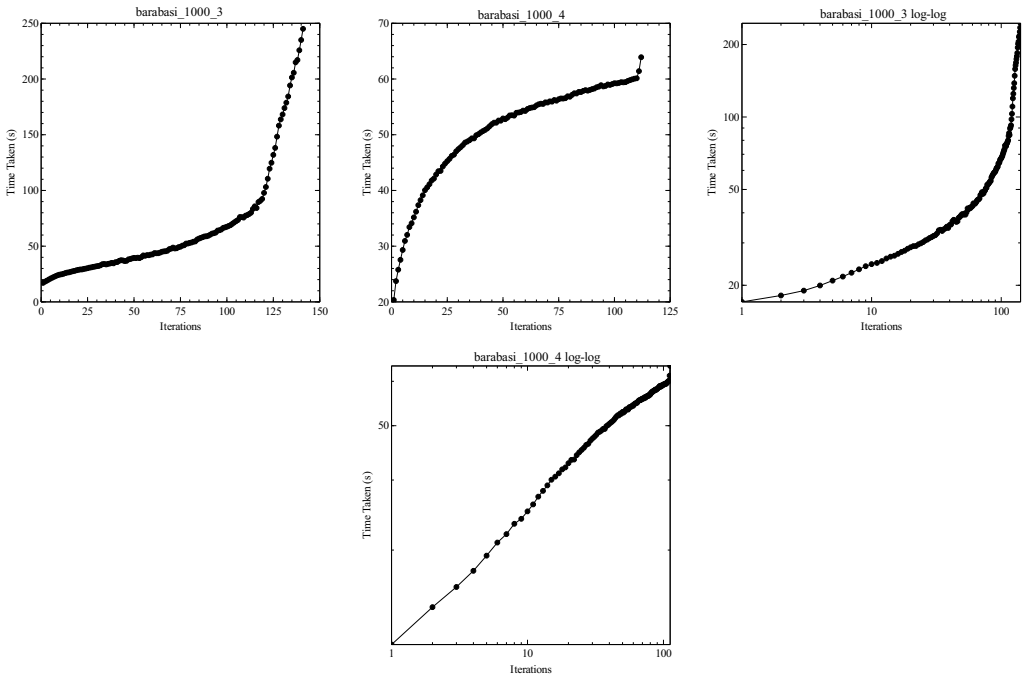

Figure 4.5: The increase in time for each iteration of the single Cypher query in the barabasi_1000 plots that finished. The two left plots show the raw results, whereas the two right plots show the results to a log-log scale. Each iteration is the addition of a node into the dominating set.

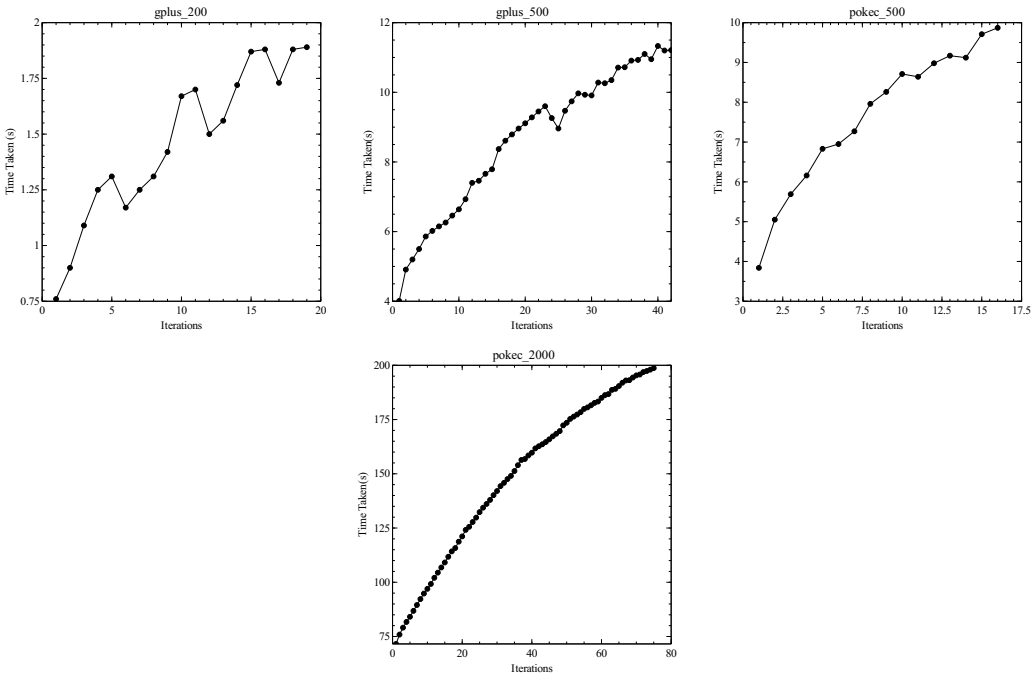

Figure 4.6: The increase in time for each iteration of the single Cypher query in the social media graphs. Each iteration is the addition of a node into the dominating set.

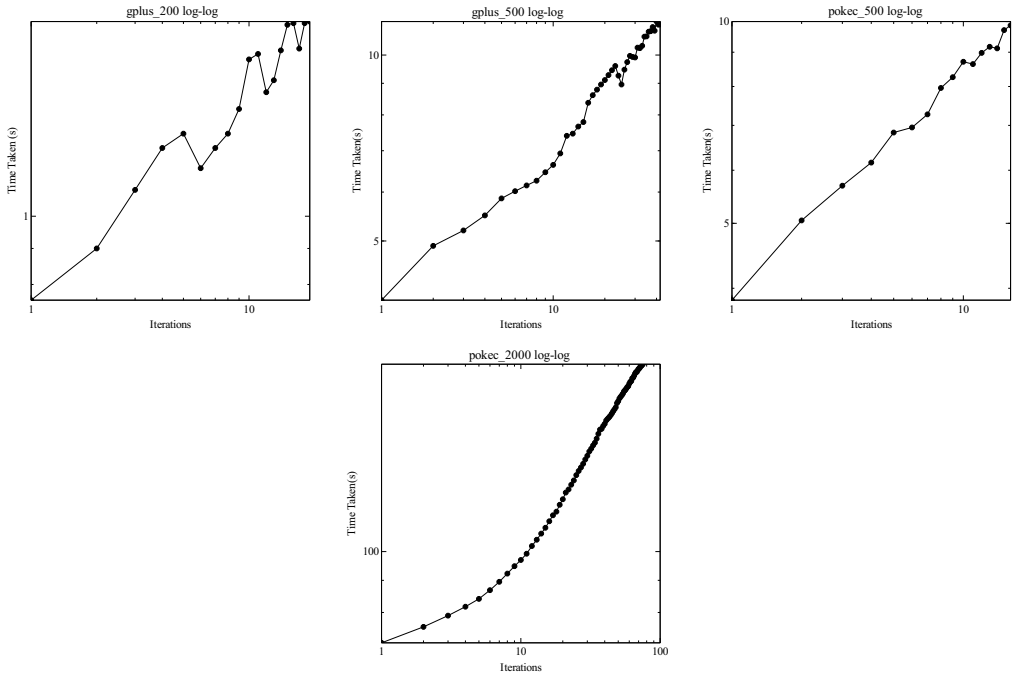

Figure 4.7: The increase in time for each iterations of the single Cypher query in the social media graphs displayed in log-log plots. Each iteration is the addition of a node into the dominating set.

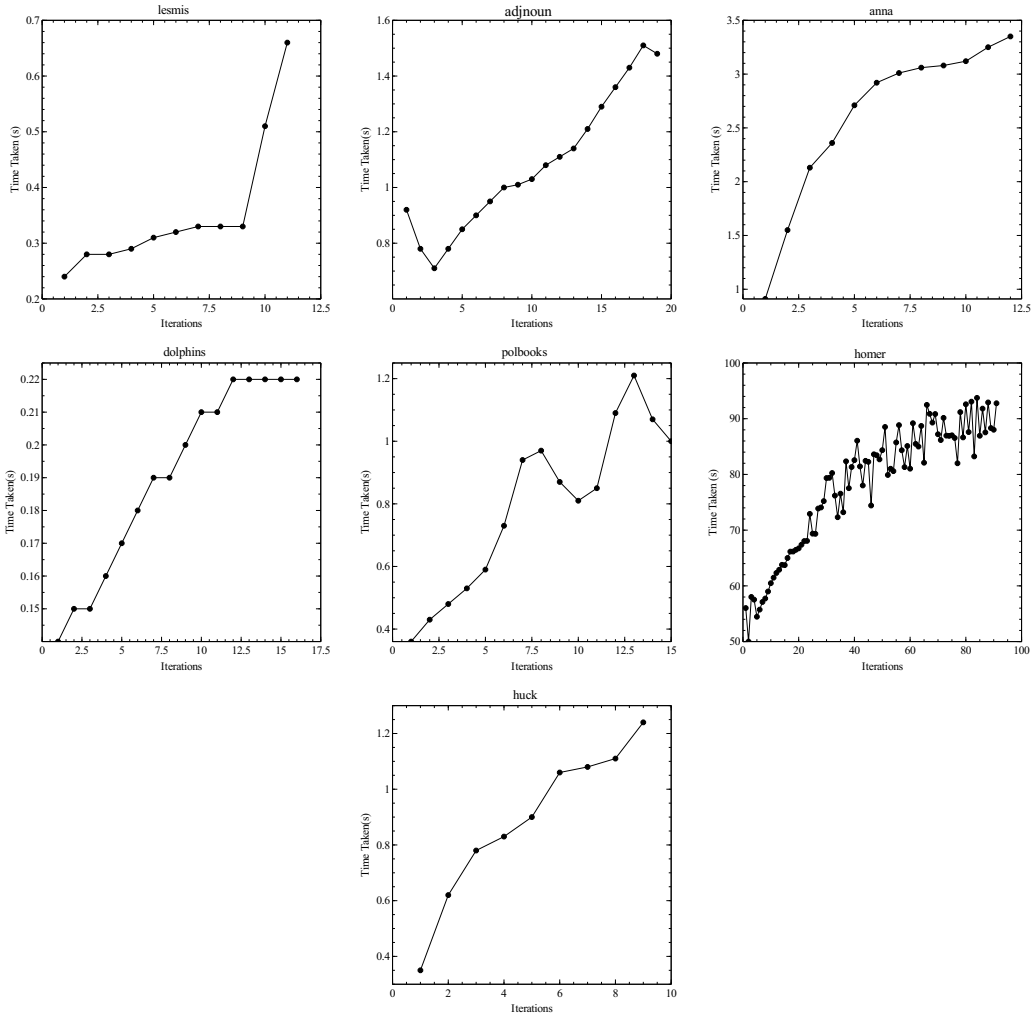

Figure 4.8: The increase in time for each iteration of the single Cypher query in Newman's repository . Each iteration is the addition of a node into the dominating set.

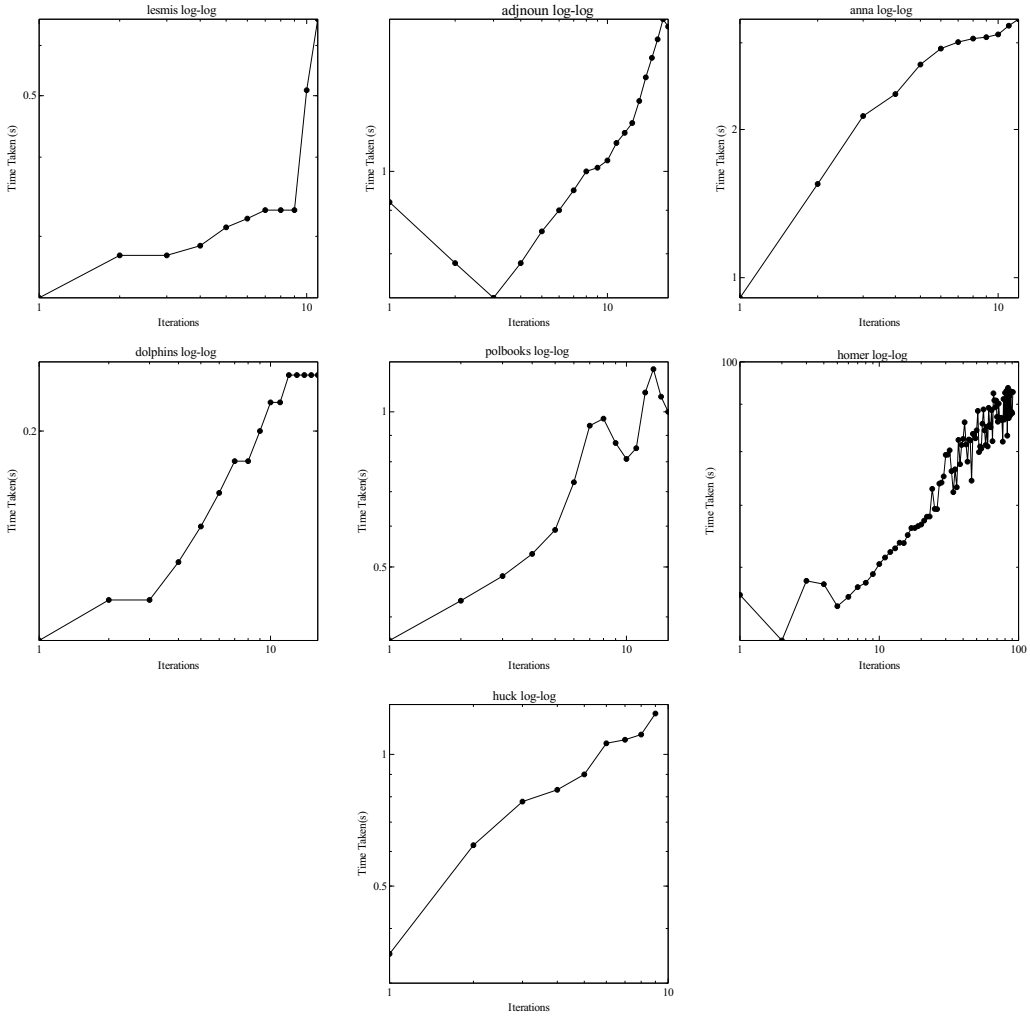

Figure 4.9: The increase in time for each iteration of the single Cypher query in Newman's repository displayed in log-log plots. Each iteration is the addition of a node into the dominating set.

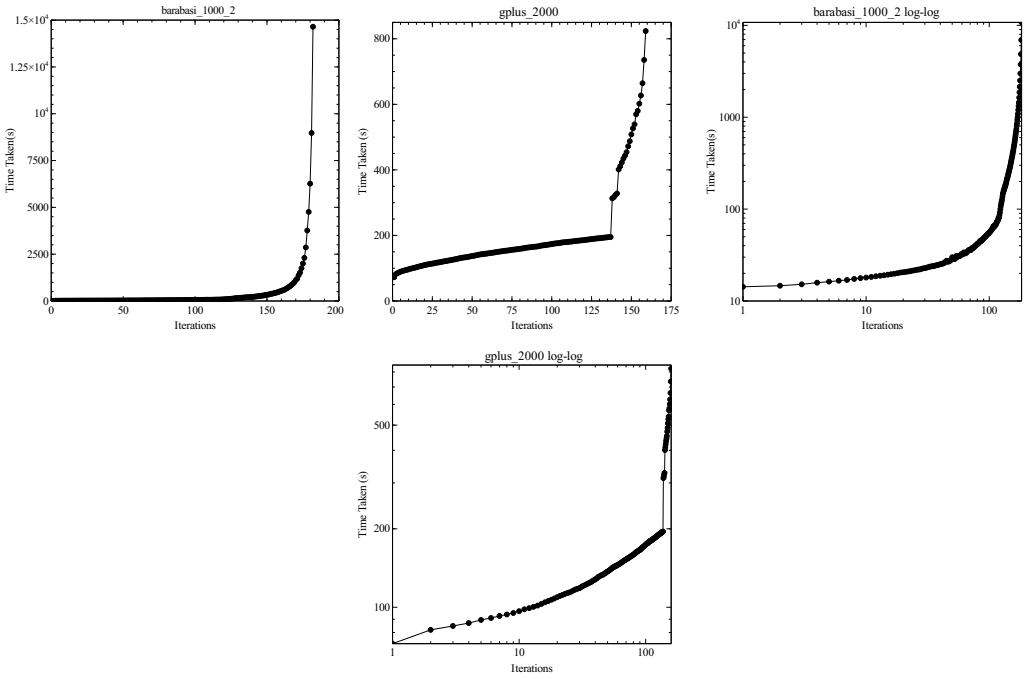

Figure 4.10: The increase in time for each iteration of the single Cypher query in each of the graphs that were not found. The plots show both the raw and the log-log plots for the instances. Each iteration is the addition of a node into the dominating set.

Table 4.2: A table showing how well the different systems coped with four different categories of graphs. An 'x' represents that the dominating set of the majority of the graphs in the category were **not** found. A '∼' represents that the dominating set for **most** of the graphs in the category were found. A '✓' represents that the dominating set for **all** of the graphs in the category were found.

Implementations	Large Instances	Medium Instances	Small Instances
Cypher One Query	x	∼	✓
Cypher With Wrapper	x	✓	✓
Sparsity With Wrapper	✓	✓	✓
C++ Implementation	✓	✓	✓

The graphs were categorised into *large*, *medium* and *small* instances, the instances categorised as large are *barabasi_10000_2*,*barabasi_10000_3*,*barabasi_10000_4*, *pokec_2000* and *gplus_2000*. The instances categorised as *medium* include *barabasi_1000_2*, *barabasi_1000_3*, *barabasi_1000_4*, *pokec_500* and *gplus_500*. And the instances categorised as *small* are *barabasi_100_2*, *barabasi_100_3*, *barabasi_100_4*, *adjnoun*, *anna*, *dolphins*, *polbooks*, *soc52*, *huck* and *lesmis*.

For the small instances, all of the implementations could find a dominating set using the greedy algorithm. For the medium instances, all of the implementations apart from the single Cypher query could find a dominating set. Cypher as a single query could not find the dominating set for the instance *barabasi_1000_2*. For the large graph, the single Cypher query could only find a dominating set for *pokec_2000*.

For the larger *barabasi_10000* instances, a single node for the dominating set could not be found. This was also the case for Cypher with a wrapper.

The results in table 4.6 describe the different slopes of the plots in figures 4.2, 4.3, 4.4, 4.5, 4.6, 4.8 and 4.10 by using the Least Squares Linear Fit Equation. The first four columns are for the raw data, with the final four columns for log-log graphs. The slope columns are the calculation of the full slope, the error is the standard error of the slope, the first slope is the either the first half of the slope, or the point on the slope at which the slope has a significant change and finally the second slope is the rest of the slope after the first slope.

Figure 4.2, gives the plot of all results combined. Almost instantly we can observe that Cypher without a wrapper takes the longest time to complete, with the C++ implementation taking the least amount of time. The instances that cause both Cyphers to crash have a spike in time for both C++ and Sparsity, which implies that Cypher struggles with larger instances.

Following on from this, figure 4.3 shows the plot of the C++ implementations that took less than 1ms to complete. This provides a deeper insight for the smaller instances. From the log-log plot, we can see that all implementations rise in a similar fashion. One interesting detail is that Cypher with a wrapper, and Cypher without a wrapper both have similar starting times. However, once the instances become larger, Cypher without a wrapper starts to take longer. Another interesting feature is for the instance *soc52*, Sparsity takes the same amount of time as the c++ implementation. This could because the instance is relatively small.

Figure 4.4 shows the increase in time for each of of the Cypher queries created

for the barabasi_100 instances. A feature that instantly stands out is that for the instance barabasi_100_3 the final created query takes less time than the previously generated query. It could be that in generating the final query, a function within Cypher isn't called, thus reducing the run-time of the algorithm. This characteristic is also shared with barabasi_100_2, although not to the same extent.

In figure 4.5, we explore in similar fashion, the time taken for each iteration of the generated cypher without a wrapper query to process the barabasi_1000 instances. These instances did find a result. This could be because of the increased search space for edges in comparison to the barabasi_1000_2 instance. Instance barabasi_1000_3 rises at what could be seen as an exponential rate in time taken, whereas for barabasi_1000_4 it is in a more linear fashion.

Figures 4.6 and 4.7 explores how Cypher without a wrapper processes the social network instances. The gplus_200 instance is interesting in that the timing fluctuate with each generation of the cypher query. As the heuristic is not guaranteed to find the same solution in each run, it could be that the generated solutions are causing the time fluctuations. The genral trend for all instances is an increase in time, however there does seem to be a number of fluctuations.

The figures 4.8 and 4.9 explores Newman's data repository and how Cypher without a wrapper interacted with these data sets. The fluctuations identified with the social networks are clearly present here also. Especially instance homer, where the closer the query got to finding a minimum dominating set, the more fluctuation in timings.

Finally, the plot in figure 4.10 shows the networks that caused Cypher without a

wrapper to crash. One item in common with all ofthe instances is that there seems to be a certain point that causes the timings to increase exponentially. For the social media instances, this seems to be after the 100th generated query, and for the other instances this seems to be after 150 generations. This implies the graph engine begins to be effected by memory limitations at these stages.

For all of the instances apart from two, the second slope of the log-log graphs is higher when compared to the first slope, sometimes significantly higher than the first slope as shown for *barabasi_1000_3*. This implies that Cypher does not scale well. However, for two of the instances this was not the case, for *barabasi_1000_4* and *anna*.

In general, the log-log slopes are generally around the 0.3 - 0.4 mark. For *barabasi_1000_2*, the slope is above 1. However the first part of the slope is 0.4, while the second half, where the slope exponentially increases, the slope increases to a factor 8. For the instances *barabasi_100_4*, *barabasi_1000_2*, *barabasi_1000_3*, *lesmis* and *gplus_2000* the second part of the slope is greater than 1.

One item of note is that both graph databases relied on an external "wrapper" to iterate functions within the query methods. One reason for cypher is that due to the declarative nature of the language, introducing a for loop may clash with other internal features that rely on recursion. For sparsity, this is less of an issue, due to the imperative nature of the query language. The introduction of an in-built iterator for functions within the libraries may increase the efficiency of algorithms.

Considering that Cypher is a declarative expressive language, our implementation has only two lines of code less in comparison to the C++ implementation. This is

Table 4.3: The table shows the slope of the plots for the raw data and log-log graphs in figures 4.2, 4.3, 4.4, 4.5, 4.6, 4.8 and 4.10

Graph	Raw Data				log-log			
	Slope	Error	First Slope	Second Slope	Slope	Error	First Slope	Second Slope
barabasi_100_2	0.04	0	0.04	0.06	0.56	0.04	0.24	0.92
barabasi_100_3	0.03	0	0.02	0.05	0.33	0.04	0.25	0.95
barabasi_100_4	0.05	0	0.05	0.08	0.39	0.04	0.33	1.04
barabasi_1000_2	9.27	1.38	1.77	294	1.09	0.08	0.4	8.12
barabasi_1000_3	1.03	0.06	0.5	6.43	0.54	0.03	0.27	2.38
barabasi_1000_4	0.26	0.01	0.7	0.14	0.23	0	0.25	0.19
adjnoun	0.04	0	0.03	0.06	0.23	0.04	0.06	0.58
anna	0.19	0.03	0.4	0.07	0.49	0.04	0.7	0.2
dolphins	0.01	4.07	0.01	0	0.2	0.01	0.13	0.21
polbooks	0.05	0.01	0.08	0.05	0.46	0.04	0.3	0.34
homer	0.39	0.02	0.75	0.19	0.16	0.01	0.09	0.17
huck	0.1	0.01	0.13	0.05	0.53	0.04	0.64	0.47
lesmis	0.03	0.01	0.01	0.17	0.29	0.09	0.15	3.46
pokec_500	0.36	0.02	0.53	0.21	0.33	0.01	0.33	0.33
pokec_2000	1.68	0.03	2.28	1.15	0.3	0.01	0.18	0.38
gplus_200	0.06	0	0.07	0.03	0.31	0.02	0.28	0.37
gplus_500	0.16	0.01	0.24	0.12	0.3	0.01	0.22	0.33
gplus_2000	2.13	0.16	0.79	21.5	0.38	0.03	0.24	7.66

also only the case if the size of the dominating set is 1. If it is greater, then the size of the query itself grows by the size of the dominating set. One might have expected the language to be more concise and expressive.

4.7 Conclusion

In summary, we found that graph databases are not yet reaching their full potential when implementing computational algorithms using the toolsets provided by the databases.

We found the query tools in both client-server and embedded databases are missing some fundamental features which means that there is a reliance on a high-level language to fill in the gaps. For Neo4j in particular, a simple loop procedure that can allow a part of the query to be repeated is a feature that could make

the algorithm more efficient when implemented using Cypher. As well as that, the ability to hold the state of a variable inside of a query seems a feature that is currently missing from the query languages.

One of the major advantages of graph databases over other database families is the graph structure. This should allow an efficient implementation of a variety of graph algorithms. While the current query methods do allow this to an extent, there are some key features , such as looping functions, that are missing. We believe this deficiency represents a significant opportunity for graph database systems to implement and address these issues. From this, we propose that a way of iterating functions within a query would be an useful addition for the query methods provided by graph databases.

In a gentle way, you can shake
the world.

Mohandas Karamchand Gandhi

CHAPTER 5

Further implications of other problem implementations

Chapters 3 and 4, explored Non-deterministic Polynomial-time Hardness (NP-Hard) and Non-deterministic Polynomial-time Completeness (NP-Complete) problems. In this chapter, we explore problems that fall into NP-Complete and Polynomial (P) classification. As shown inthe previous chapter, Graph database systems are found to be inefficient when given a NP-complete problem that relied on looping functions. This gives us a chance to explore the effectiveness of graph database systems when given problems that can be found within graph database query search spaces.

The contribution that can be found in this chapter is a further exploration of graph database systems by having novel implementations of the Betweenness Centrality, Graph Diameter and the Component Labelling algorithms. This allows

an in-depth investigation of built-in functions of the query methods provided by the systems.

The chapter is structured as follows; section 5.1 introduces the topics and problems discussed in the chapter, section 5.1.1 gives a review of the literature (some repetition of chapter 2 may occur here), section 5.2 explains some of the procedures used in the query languages for this chapter and the following section gives some high-level implementations of the problems used. Section 5.4 then gives the implementations of the problems in the graph query languages. The results of the experiments are then given in section 5.5 followed by a discussion and conclusion.

5.1 Introduction

Heuristical algorithms are difficult to implement using the query methods provided by database systems, as shown by the previous chapter. Declarative query languages such as Cypher can potentially hinder complex scenarios, and imperative query languages do not provide enough functionality for the processes used by the Minimum Dominating set problem. We can investigate how well database system can efficiently perform algorithms from different complexity classes that do provide a consistent solution.

Firstly it is worth recapping that finding patterns within a graph is computationally hard (Barceló et al., 2011), thus querying graph databases are computationally hard (Barceló Baeza, 2013). With the addition of Atomicity, Consistency, Isolation and Durability (ACID), there is a danger that the graph database systems cannot

efficiently run algorithms. As well as that, a graph query language has not yet been formalised.

Data is coming from all sorts of places (Hawick, 2014). Different algorithms give different insights to each of these data sets. With all of this extra data, the definition of a "large" database has changed (Waltz et al., 1987). Some graph algorithms can be parallelised (Hawick et al., 2010b; Dineen et al., 2011; Gebremedhin and Manne, 2000) for use on GPUS (Leist and Hawick, 2011; Gallian, 2005). Graph metrics allow complexities within data to be found across mutltiple types of data sets. For example, graph metrics have been used for accessing electrical grids (Hawick, 2012b,a) as well as water networks (Hawick, 2012c) and biology data analysis (Hawick, 2011).

In this chapter we focus on three different problems where common algorithms produce results that do not alter in different runs. The first problem is Betweenness Centrality. It is a highly computational algorithm, with a complexity of $O(V^3)$. Betweenness centrality has been used in electricity networks (Hawick, 2012b,a) and for protein-protein interaction (Hawick, 2014). We will focus on the unoptimised version as a comparison.

The second problem is the Graph Diameter problem. This is described as finding the "longest shortest path" in an instance, or the longest eccentricity. While it is similar to the betweenness centrality algorithm in that every shortest path in a graph can be visited, it requires less computational power. This is because a solution can be found with a more efficient algorithm; the Floyd-Warshall algorithm, which has a complexity of $O(|V|^3)$.

The final problem we explore is Component Labelling. The components of an

instance are labelled, using the one pass method. The label is a unique integer that indicates if the node is part of larger "component" in a graph. This is an interesting problem in that in an undirected graph, should the algorithm run to its end, every node would be labelled the same. Therefore, this problem is best explored in time-steps.

The three problems provide a unique perspective on algorithms, providing unique search spaces and each taking advantage of different procedures inside of graph database systems.

5.1.1 Graph databases

To recap from previous chapters, limitations of relational databases are widely known. This limitation led to the creation of Not Only SQL (NOSQL) database families. NOSQL databases focus on the scalability of data. One of the database types within this family are graph databases, a unique family that provides a graph-based engine for storage of data.

Graph theory has benefited from decades of research. This has allowed it to produce a selection of algorithms which can be used in data. These algorithms could potentially be applied across a variety of graphs and graph models.

As the underlying nature of graph databases are graphs, it allows the possibility of combining graph algorithms and graph databases. Graph Databases use the property graph model to store data. Previous graph database models such as GROOVY, GOOD and others are known and are reviewed in (Angles and Gutierrez, 2008; Angles, 2012). The flexibility of the property graph model allows for a wide range

of data to be imported into a database.

Typically, graph databases can be defined into two separate types; they can be either embedded or client-server.

An embedded database is stored local to a machine. Queries to the database are typically made by in-code library calls supplied by the database systems. At the time of writing, the most popular embedded database is Sparsity (formerly DEX).

A client-server database uses an external server engine which can be queried by a client. An example of a popular client-server database system is Neo4j. Embedded database systems typically do not have a query language, as opposed to client-server database systems, which typically do.

Other graph database systems have been reviewed in Buerli (2012). For experiments in this chapter, representing embedded databases is Sparsity, and representing client-server databases is Neo4j. They have been compared in previous literature (Beis et al., 2015; Vicknair et al., 2010), as well as in the previous chapter.

5.2 Query languages

As stated in previous chapters 2 and 4, client-server databases typically provide a query language to query a database. At the time of writing, there is not a standardised query language for graph databases.

One such query language that has gained protraction is Cypher. Cypher is a declarative scala-based query language created by Neo4j. It heavily uses ASCII-like symbols to represent query expressions. For example, a *()* represents a node

in a query. Two versions of Cypher exist, Cypher and openCypher. openCypher is an attempt to standardise a query language for graph databases, and is essentially Cypher, with some features excluded.

Cypher has been found to have some limitations, such as the inability to hold state or allow parts of a query to be re-cursed (Balaghan et al., 2017), as shown in the previous chapter. This inability effects the implementations of some algorithms within Cypher, such as finding the minimum dominating set in a graph, or others which require state. The query language Cypher has some in-built functions that are some famous graph algorithms. Some of these are the shortest path and the all shortest path algorithms.

In contrast, embedded databases typically provide imperative-based libraries as functions to query the databases. These have some calls within the libraries which act as the functions found within query languages.

5.3 Exploration of problems

Betweenness Centrality, Graph Diameter and the Component Labelling problems have been implemented in Sparsity and Neo4j. The implementations take advantage of built-in functions such as finding the shortest path between two nodes. This allows us to explore how reactive the graph database engines are to pre-built functions when used as part of a bigger algorithm.

To begin with, we recap some features of graph G. In this section, a graph is assumed to be a simple, undirected without self-loops. A digraph is referred to as

$G = (N, E)$ whereby N is a non-empty set of nodes $(n_1, n_2...n_n)$ and E is a set of edges such that $E = NxN$. A directed edge between two nodes is depicted as $[a, b]$, and $[a, b] \in E$. A graph is directed if E is a set of ordered pairs. The digraph may contain multiple edges and self-loops. However they do not contain dangling edges.

A path p is a sequence of nodes that lead from a source node v_1 to a sink node v_n. The path contains a set of nodes $(v_1, v_2...v_n)$ and a set of edges $([v_1, v_2], [v_2, v_3]....[v_{n-1}, v_n])$ subject to $v_n \in N$ and $[v_n, v_{n+1}] \in E$. The immediate nodes in a path refer to the nodes that do not include the source or sink of the path.

Finding the shortest path between two nodes is a well known, and well researched, graph theory problem. The most famous algorithm for finding the shortest path is the one created by Dijkstra (1959). Dijkstra's original algorithm had a complexity of $O(|V^2|)$ however improvements have been made to eventually give the complexity of $O(|E| + |V|log|V|)$ (Fredman and Tarjan, 1987). It finds the shortest possible path between two given nodes $d(x, y)$.

Another method of finding the shortest path between two nodes in a graph is by using the Floyd-Warshall algorithm (Floyd, 1962). A single run of the algorithm returns the lengths between all of the pairs of nodes in a graph.

A metric that build on top of the shortest path is the Graph Diameter problem. The diameter of the graph is the "longest shortest path", i.e. $maxd(x, y)$ also, the max eccentricity. The graph diameter can be found with a complexity of $O(|V^3|)$ using Floyd-Warshall's algorithm (Floyd, 1962).

Another variant of the shortest path is the all shortest-paths algorithm. The all shortest-path algorithm finds all of the shortest paths between two given nodes,

Algorithm 22 The Floyd-Warshall shortest path algorithm (Floyd, 1962)

```
let dist be a N x N array of distances, init set to ∞
for node n in N do
    dist[n][n] ← 0
end for
for edge e in E do
    dist[e.source][e.sink] ← weight(e)          ▷ In an unweighted graph set to 1
end for
for int k to N.size() do
    for int i to N.size() do
        for int j to N.size() do
            if dist[i][j] > dist[i][k] + dist[k][j]  then
                dist[i][j] ← dist[i][k] + dist[k][j]
            end if
        end for
    end for
end for
```

should multiple paths exist. It tends to return a list of possible paths, all of which are the same length.

This variant can be used to find the betweenness centrality of a graph. The Betweenness Centrality problem finds the most "critical node" in a graph. This can be found in equation 5.1 (Freeman, 1977, 1978).

$$g_k = \sum_{i \neq j} \frac{c_k(i,j)}{c(i,j)} \tag{5.1}$$

One particular item of note is that the all shortest path algorithm, rather than the shortest path is used to find all of the shortest paths between two nodes. This allows for multiple short paths to be found between any two nodes.

If a shortest path is found, any immediate nodes are then given a weighting. The

weighting is calculated by dividing 1 by the number of shortest paths found. For example, if a single shortest path is found, then each of the immediate nodes will add one onto their weights. If two paths exist, then 1/2 is added onto the weights. The node with the highest weighting in the graph is deemed the most "critical" node.

Finding the Betweenness Centrality of a graph is a very computationally demanding algorithm, as every shortest path in a graph must be found in order to give the correct weightings to the nodes, as opposed to the other two problems being investigated in this chapter.

The Component Labelling problem find clusters in a graph. The pseudocode can be found in algorithm 23. To begin with, all nodes in the graph are given a unique "label". In this instance, the label is a unique integer value. Beginning with a starting node n , check the neighbourhood of n, if any of the "label"s of the nodes in the neighbourhood are lesser than n, then set the label of n equal to the label of the neighbour.

Algorithm 23 The Component Labelling Algorithm

```
1: Give all nodes a unique label
2: while There are label changes do
3:     for Node n in G do
4:         for Neighbourhood l in n do
5:             if l then.label < n.label
6:                 n.label = l.label
7:             end if
8:         end for
9:     end for
10: end while
```

5.4 Problem implementions

The algorithms in the previous section have been implemented into Sparsity and Neo4j. They have been implemented into Neo4j using Cypher and Sparsity using Java with the libraries supplied by Sparsity. We have also implemented these algorithms into C++ as a comparison.

5.4.1 Implementations in Neo4j

Cypher has an inbuilt function call which provided the shortest path between two given nodes. The query in alg. 24 gives a usage example of the shortest path algorithm. Within the MATCH clause, the shortest path is given as a in-built function.

Algorithm 24 A Cypher query to find the shortest path between two nodes.

```
1: MATCH (n), (m), p = shortestpath((n)-[*]->(m))
2: WHERE ID(n) = i and ID(m) = j
3: RETURN p
```

The shortest path algorithm is an in-built function in Cypher. It can be implemented by using the query in alg. 24. The shortest path query in Cypher differs from Dijkstra's algorithm in that it starts from both the source and destination node. In the *MATCH* clause in alg. 24, p is finding the shortest path between nodes n and m. The result is then returned as a Path.

By using this in-built shortest path function, we can find the Graph Diameter algorithm in Cypher. The query in alg. 25 shows how we can achieve this. Line one of the query essentially states *find every single shortest path possible in graph g.*

Algorithm 25 A cypher query to find the diameter of a graph.

```
1: MATCH p= shortestpath((n)-[*]->(m))
2: RETURN length(p)
3: ORDER BY length(p) DESC
4: LIMIT 1
```

The remainder of the query essentially states *return all of the found paths, and order them descending by length* and then to *reduce this list to the top result.* The result is limited to one as only the first item in the list would give the longest shortest path, and thus the graph diameter. This optimisation also reduces the search space required by the query.

A separate function that builds upon the shortest path is the all shortest-path function. The all shortest-path function finds all of the shortest paths between two nodes. An example of the call can be found in algorithm 26.

Algorithm 26 The all shortest-path function in a Cypher query

```
MATCH p=allShortestPaths((a)-[*]->(b))
WHERE a <> b
RETURN p
```

This can be used to find the Betweenness Centrality. Building upon the query created by Münch (2017) in his blog, algorithm 27 gives the query that can be used in Cypher to find the Betweenness Centrality. It firstly finds all of the shortest paths between a and b. It then collects the paths and puts them into a list. The count of the size of the list is the total count of shortest paths and is saved as allcount. *UNWIND* is used to split the list into separate entities. *UNWIND* is then again used to remove the first and last nodes in each of the paths, giving the immediate nodes a specific weighting. The two *WITH* clauses allow the sum of immediate nodes to

be kept for each individual node in the query. Finally, an ordered list of every node with its weighting is returned.

Algorithm 27 The betweenness centrality algorithm in Cypher. Adapted from (Needham, 2013).

```
1: MATCH p=allShortestPaths((a)-[*]->(b))
2: WHERE a <> b
3: WITH a,b,collect(p) AS allpaths,
4: count(p) AS allcount
5: UNWIND allpaths AS p
6: UNWIND nodes(p)[1..-1] AS n
7: WITH n,a,b,1/tofloat(allcount) AS fraction
8: RETURN ID(n), sum(fraction)
9: AS betweenness_centrality
10: ORDER BY betweenness_centrality DESC
```

The component labelling algorithm can be defined as a query in algorithm 28. The algorithm sets each node to a unique "label" by using its unique database ID. Then every single pair of nodes in the graph are compared, with the label property being compared and the lower label being set.

Algorithm 28 The component labelling algorithm in Cypher.

```
1: MATCH (n)
2: SET n.label = id(n)        ▷ Set the label to the Unique Database ID as a Float
3: WITH n
4: MATCH (m)–>(j)
5: WHERE tofloat(j.label) > tofloat(m.label)
6: SET j.label = m.label
```

An interesting feature of Cypher with the component labelling in algorithm 28 is that the algorithm automatically recurses through until the "final" time-step of the algorithm is complete. For a graph this will mean that the label of each node

in the graph will be the same. While for a digraph, clusters will be found. This is because the directions of the edge will not always allows for the smallest label to be swapped. An item of note is recursion in the queries. Recursion is limited to what can occur within a query statement in Cypher, rather than a function. As shown in algorithm 28, setting the labels of all of the nodes can be complete in a single line of code. However, expanding this into a function is limited by Cypher. While recursion is possible on singular statements, entire functions require more.

5.4.2 Implementations in Sparsity

The three problems were implemented into Sparsity. The first item of note is that there are very few in-built procedures in Sparsity libraries. A lot of the algorithms had to be programmed with the external language.

Algorithm 29 The all shortest-paths problem in Sparsity.

```
1: Objects neightbourOfNode;
2: neighbourOfNode = g.neightbours(currentnode, GenericEdge, EdgeDirection.Any)
3: ObjectsIterator it= neighbourOfNode.iterator();
4: while it.hasnext do
5:     alt = distance[currentverrtex]
6:     neigh_distance = distance[it.next]
7:     if alt <= neigh_distance then
8:             ▷ Then update the distance,and check if it is longer than the previous shortest path, if so, end algorithm
9:     end if
10: end while
```

Algorithm 29 shows how the inner core of the all shortest-path algorithm has been implemented using Sparsity's libraries, as it does not have this function built-

in. Therefore for the implementation of the Betweenness Centrality problem, the all shortest-path algorithm has been manually created. All of the graph logic is done by the libraries inside of Sparsity.

Algorithm 30 shows the implementation of the component labelling algorithm in Sparsity. The in-built neighbours function is used to find the neighbours of the nodes in the graph. The attributes within the graph are set using the library calls as well.

Algorithm 30 The component labelling algorithm in Sparsity.

```
1: AllNodes = getAllNodes(); Graph g; change = false
2: for long j : AllNodes do
3:     g.setAttribute(j, LabelAttrib, new Value.setInteger(id))
4: end for
5: while change do
6:     for long i : AllNodes do
7:         Objects neighbourOfNode
8:         neighbourOfNode ←g.neighbours(i,GenericEdge,Edgedirection.Any)
9:         currentLabel ← g.getAttribute(i,labelAttrib)
10:        original ← currentLabel
11:        ObjectsIterator it ← neighbourOfNode.iterator;
12:        while it.hasnext do
13:            currOID ← it.next
14:            neighbourLabel ← g.getAttribute(currOID, labelAttrib)
15:            if currentLabel >neighbourLabel then
16:                currentLabel ← neighbourLabel
17:            end if
18:        end while
19:        if currentLabel != original then
20:            g.setAttribute(i, labelAttrib,new Value.setInteger(currentLabel))
21:            change = true
22:        end if
23:    end for
24: end while
```

Finally, the graph diameter algorithm uses the same functions as the component labelling algorithm. As the fastest solver of the Graph Diameter problem is the Floyd-Warshall algorithm, most of the solution has again been programmed in the third party language.

5.5 Results

We have taken the various graph algorithms in section 5.4.1 and have implemented them into Neo4j, Sparsity and our own C++ implementations. The algorithms have been implemented in Neo4j using the queries in section 5.4.1, in Sparsity using the code shown in 5.4.2 and as a direct comparison, the same algorithms implemented in C++ by ourselves. We compared the results of the implementations to ensure accuracy.

The data sets used to create the graphs include a karate network (Zachary, 1977), a dolphin social network (Lusseau et al., 2003) , a political book network , a political blog network (Adamic and Glance, 2005), a yeast protein interaction network (Xenarios et al., 2001) and a arxiv general relativity collaboration network. As well as that, some real-world social networks were used (Chalupa, 2017; Chalupa et al., 2017).

We have also created some instances based on the Barabási-Albert model (Barabási and Albert, 1999). For the Barabási-Albert instances we have used the preferential attachment property, and gave the instances either 100, 1000 or 10000 nodes. For each of these instances, they had either a seed number of 2, 3 or 4. The same instance

that was created was used for each of the algorithms.

Sections 5.5.1, 5.5.2 and 5.5.3 give the results for the Component Labelling, Betweenness Centrality and Diameter implementations respectively.

5.5.1 Component Labelling

Table 5.1 gives the raw results for the component labelling implementations. The full raw data plotted results are shown in figure 5.5.1. For a clearer comparison, Sparsity and the C++ implementation have been plotted into figure 5.5.1. For both of the plots, the x-axis has been sorted in the order of time taken for Cypher. This was to make the plot more clear. The order in which the plots are sorted can be found in table 5.2.

The timing results for the component labelling algorithm are very interesting. To begin with, the C++ implementation completely outperforms the other two implementations. For C++, all instances apart from *barabasi_1000_4* and *netscience*, were solved in less than 0.01 seconds. Sparsity also performs quite well, with three instances equalling C++, and for every other instance it is quicker than Cypher. Interestingly, there seems to be a spike for Sparsity and C++ when it comes to the *netscience* instance, something that does not occur for Neo4j. As Neo4j goes to the final state of the instance instantly, it could be that a certain timestep in the algorithm run could cause a spike in time for the other implementations.

For Sparsity and C++, it is clear that the additional complexities that arise with database queries have effected the performance of Sparsity. When the performance results of sparsity are compared with Cypher, it is significantly faster. However,

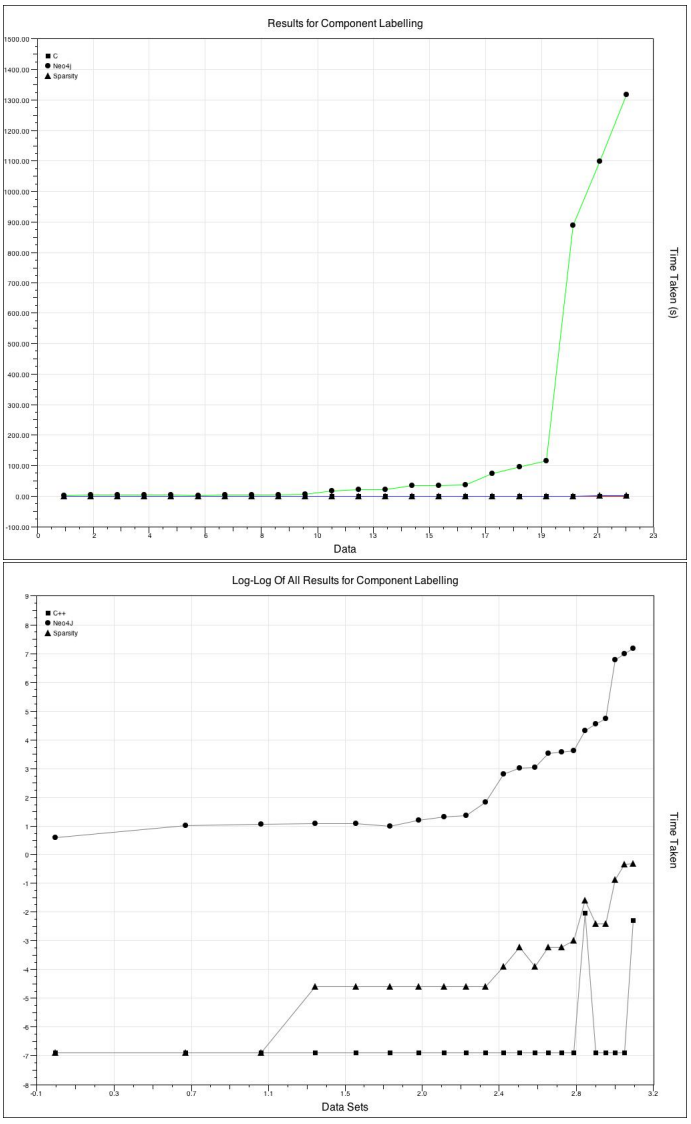

Figure 5.1: Results of the component labelling problem. For both of the plots, the labels for the data sets on the x-axis can be found in table 5.2. The first plot shows the raw data results, and the second shows the log-log results.

when compared to our C++ implementation, it is significantly slower.

In comparison to the other two algorithms, Cypher is significantly slower than the other implementations for all instances. An interesting feature of Cypher's implementation of how the engine handles the component labelling query is that all iterations of the loop are run in one go. This means that for undirected graphs, the "label" of all of the nodes will be the same. This takes away the information that the algorithm could potentially give about the instances.

5.5.2 Betweenness Centrality

The raw results of the Betweenness Centrality problem implementations can be found in table 5.5.2. A N/A signifies that the instance in that implementation did not finish within a reasonable time-frame (>a weeks computation) or it caused an internal database crash. The raw and log-log plot of this data can be found in figure 5.3. The x-axis has been sorted to time taken for Cypher, and then time taken for Sparsity. The order in which the data has been sorted can be found in table 5.4.

For the results given a N/A, this could either be a Java out of memory exception, or crashes Neo4j without giving a response.

The first item of note is that apart from the instance *dolphins* and the instances that did not finish, Neo4j outperforms the Sparsity and the C++ implementations. Neo4j was the only system vendor to have an in-built *allshortestpath* function. This implies that the *allshortestpath* function within Neo4j is efficient.

Another interesting finding from the Betweenness Centrality problem results is that many instances did not finish for Neo4j. Instances *barabasi_1000_4*, *homer* and

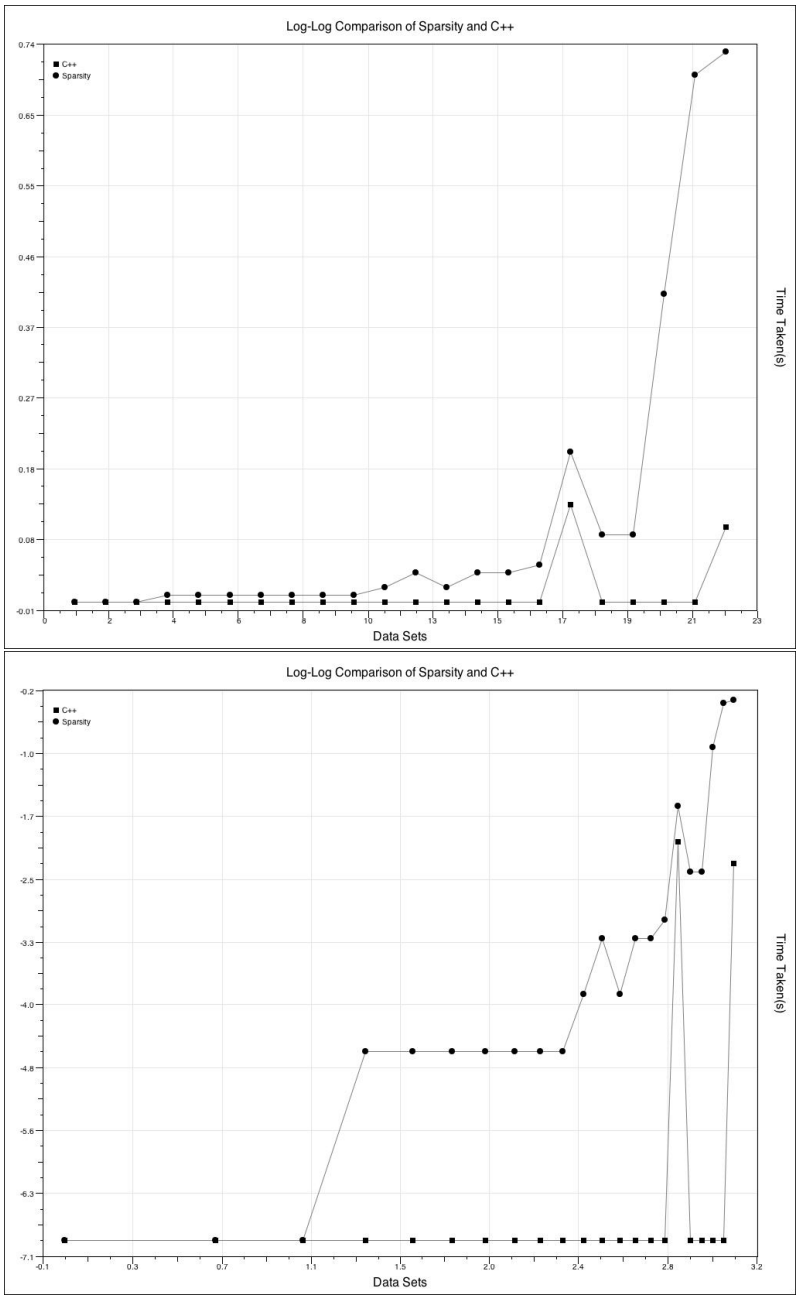

Figure 5.2: A comparison of Sparsity and C++ of the Component Labelling. For both of the plots, the labels for the data sets on the x-axis can be found in table 5.2. The first plot shows the raw data results, and the second shows the log-log results.

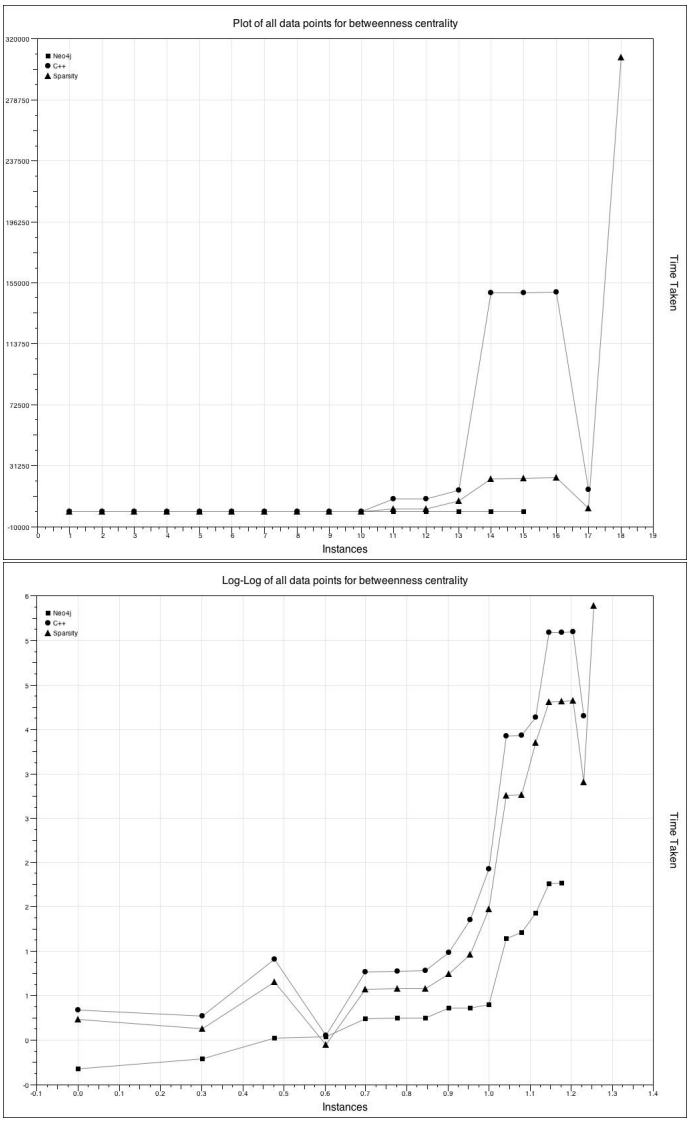

Figure 5.3: Plot results of the betweenness centrality implementations. For both of the plots, the labels for the data sets on the x-axis can be found in table 5.4. The first plots shows the raw data results, and the second shows the log-log results.

pokec_2000 crashed Neo4j, but a result was found for Sparsity. This implies that the procedure used by Neo4j is memory heavy, and would not scale above these instance sizes, or it would scale, with more memory becoming available.

Sparsity's implementations were faster than our own implementations. Sparsity even found a result for instance *pokec_2000*, when neither the C++ nor the Neo4j implementations could find one. This implies that Sparsity can be efficient for larger instances.

5.5.3 Graph Diameter

Table 5.5 has the raw data results for the graph diameter implementations. A N/A in the table signifies an out-of-memory exception. The raw data and the corresponding log-log points have been plotted into figure 5.5.3. For the x-axis of these plots, the data has been sorted by time taken in Neo4j, and the order of the instances can be found in table 5.6.

dolphins, lesmis, huck, barabasi_100_2, barabasi_100_3, barabasi_100_4, polbooks, gplus_200, gplus_500, barabasi_1000_2, barabasi_1000_3 and *barabasi_1000_4* plots indicate that the C++ implementation is more efficient. These instances are generally small in nature. It is worth noting that for two small instances, *adjnoun* and *pokec_500*, Sparsity is faster than the C++ implementation.

As the size of the instances grow, the more efficient Sparsity becomes in comparison to C++.

It is worth noting that for the larger *barabasi_10000* instances, Sparsity crashes with an out-of-memory exception. Something that does not occur in Neo4j or

in the C++ implementation. For these instances, Neo4j outperforms the C++ implementation. This implies that even for larger instances, the shortest path function in Neo4j is efficient for large data sets.

5.6 Additional discussion

From the results found in the previous section, some interesting features appear.

It is clear that the shortest path and the all shortest paths algorithms built into Cypher are highly optimised. The timings of the betweenness centrality problem show this. However, the use of these algorithms rely on having a amount of memory available. Should it be available, it is fair to say that the algorithm would be quicker than an unoptimised betweenness centrality algorithm written in C++.

When the component labelling algorithm is implemented into Cypher, we found that it completed the algorithm to its final step. In an undirected graph, the final step of the algorithm will give all of the nodes in the graph the same label. While in a directed graph this could be useful, in an undirected graph the algorithm would not produce results that are noteworthy.

The diameter algorithm results are very interesting. There are a few different cross-overs when observing the log-log plots. To begin with, Neo4j is slower than the other implementations, however once the instances became larger, it became quicker than the C++ and the Sparsity timings. For the very small instances, the C++ implementations are quicker. As Graph Database systems have additional overheads within queries to maintain ACID, the time increase could be because of this. Once the

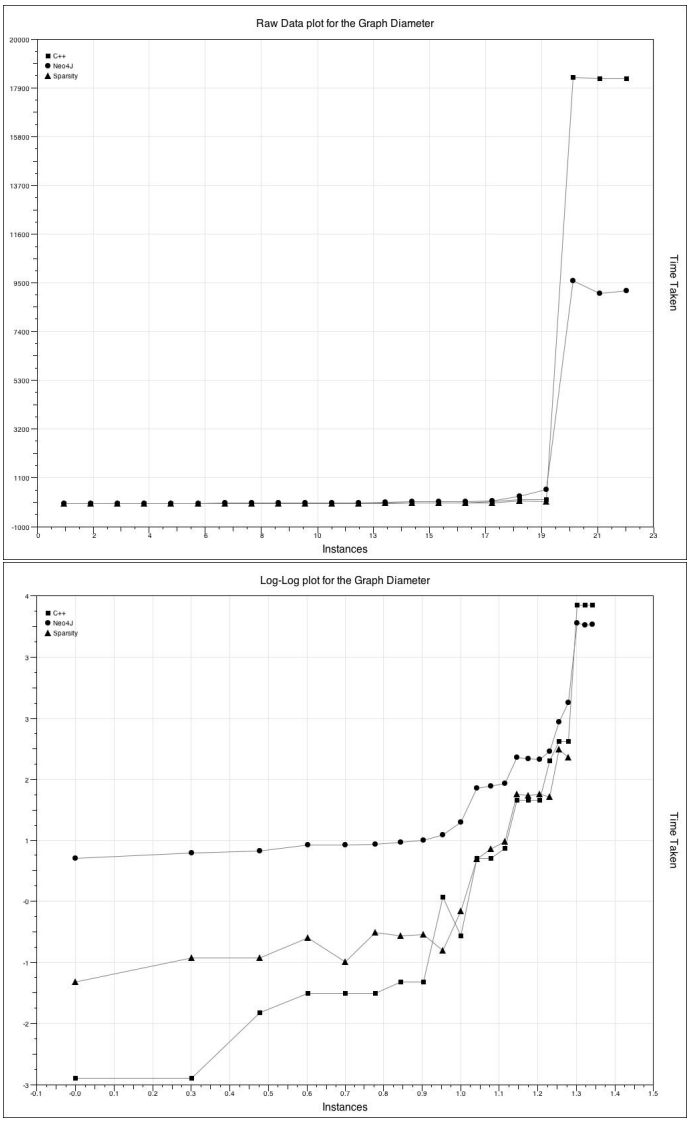

Figure 5.4: Plot results of the graph diameter implementations. For both of the plots, the labels for the data sets on the x-axis can be found in table 5.4. The first plot show the raw data results, and the second shows the log-log results.

instances become larger, the timings of the Sparsity and the C++ implementations become closer to each other. Sparsity crashes once the largest instances were loaded, which is interesting as this does not occur on the other two instances.

A difference that can be seen almost immediately is the difference in amount of lines required to complete each feature. Sparsity uses an imperative query language style, whereas Cypher is a declarative language. For example, completing the single shortest path algorithm in Sparsity uses 13 lines of code in comparison to Cypher, which only uses 3.

The expressiveness of Cypher shines in some of the algorithms explored in this chapter. For example, the graph diameter query only requires 4 lines of code. The declarative style allows Neo4j to choose the most efficient method to run the query. In the cases of betweenness centrality and the graph diameter problems, this is shown.

However, there is an issue. For the component labelling algorithm, it is most beneficial when each loop of the algorithm can be stopped so that the components can be studied. However, Cypher runs the algorithms until it has finished. For a directed graph this would be less of an issue as some components would still occur based on the start node. For an undirected graph, all of the nodes will have the same label once the algorithm has finished its run, thus deeming the use of the algorithm irrelevant.

5.7 Conclusion

Graph databases provide an unique opportunity for graph algorithms to be applied to a database structure. Hence why having highly computational and optimised algorithms is useful, as well as having an efficient graph engine to process said algorithms. In conclusion to this chapter, the in-built functions within the languages tend to be efficient time wise but there there are some features that could be improved.

The in-built algorithms provided by the graph database system, tend to be very optimised speed wise and can even outperform our own code. However, these functions rely on a high amount of memory being present. This indicates that the functions can be unoptimised, which can be seem form the performance results of Betweenness Centrality.

If the amount of memory required is available, Neo4j's performance in finding the betweenness centrality of an instance was impressive. It seemed to be more more efficient than Sparsity and maybe even a vanilla C++ implementation.

There is a function within Cypher that is missing but would be useful. Being able to break within an inbuilt function would allow algorithms such as the component labelling algorithm to be complete. We propose that a looping mechanism with the ability to break out once a condition has been met would be a useful function for query methods.

One item of note is Sparsity's library functions calls were less than expected, and that it was generally impressive in how efficient the basic graph functions were inside of sparsity. The general expressiveness of cypher was also very impressive, with some

algorithms being expressed in a small number of lines of code.

To conclude, the query methods given by the graph database systems currently rely on high memory usage, which implies the engines could be more efficient when given a highly computational problem. As well as that, we propose an additional function for the query methods. This function would build upon the loop function proposed the previous chapter by allowing a state of the iteration to be stopped once a given condition has been met. The instance that has been built at that time is then to be returned.

Table 5.1: A table to show the time taken for Cypher, Sparsity and our own C++ implementations to find a solution for the component labelling algorithm against the networks.

Graph	CYPHER	Sparsity	Our Implementation
barabasi_100_2	2.92	<0.01	<0.01
barabasi_100_3	2.95	0.01	<0.01
barabasi_100_4	2.95	0.01	<0.01
barabasi_1000_2	33.86	0.04	<0.01
barabasi_1000_3	35.38	0.04	<0.01
barabasi_1000_4	37.51	0.05	<0.01
barabasi_10000_2	888.93	0.41	<0.01
barabasi_10000_3	1098.77	0.70	<0.01
barabasi_10000_4	1316.69	0.73	0.01
adjnoun	3.34	0.01	<0.01
anna	3.92	0.01	<0.01
dolphins	1.84	<0.01	<0.01
polbooks	3.74	0.01	<0.01
homer	20.45	0.04	<0.01
huck	2.76	<0.01	<0.01
lesmis	2.72	0.01	<0.01
pokec_500	20.82	0.02	<0.01
pokec_2000	115.32	0.09	<0.01
gplus_200	6.3	0.01	<0.01
gplus_500	16.74	0.02	<0.01
gplus_2000	95.21	0.09	<0.01
netscience	74.42	0.2	0.13

Table 5.2: The order of which data sets are shown in the Component Labelling result graphs. They have been sorted by time taken for the Neo4j results from table 5.1.

Place	Name of Data Set
1	dolphins
2	huck
3	barabasi_100_2
4	barabasi_100_3
5	barabasi_100_4
6	lesmis
7	adjnoun
8	polbooks
9	anna
10	gplus_200
11	gplus_500
12	homer
13	pokec_500
14	barabasi_1000_2
15	barabasi_1000_3
16	barabasi_1000_4
17	netscience
18	gplus_2000
19	pokec_2000
20	barabasi_10000_2
21	barabasi_10000_3
22	barabasi_10000_4

Table 5.3: A table to show the time taken for Cypher, Sparsity and our own C++ implementation to find a solution for the betweenness centrality algorithm against the networks. N/A signifies that the instance did not finish in a reasonable time-frame (> week computation), or caused a database crash.

Graph	Cypher	Sparsity	C++ Implementation
barabasi_100_2	3.83	8.42	13.58
barabasi_100_3	3.85	8.58	13.78
barabasi_100_4	3.9	8.71	14.04
barabasi_1000_2	152.44	22241.86	147981
barabasi_1000_3	155.53	22695.89	147818
barabasi_1000_4	N/A	23024.58	148677
barabasi_10000_2	N/A	N/A	N/A
barabasi_10000_3	N/A	N/A	N/A
barabasi_10000_4	N/A	N/A	N/A
adjnoun	5.05	12.92	22.99
anna	5.1	22.01	57.08
dolphins	2.32	1.87	2.42
polbooks	2.23	10.32	19.38
homer	N/A	2486.17	15026.1
huck	1.28	2.91	4.09
lesmis	0.97	3.73	4.85
pokec_500	33.85	1731.49	8710.89
pokec_2000	N/A	307704.41	N/A
gplus_200	5.61	76.21	226.04
gplus_500	40.08	1752.33	8891.7
gplus_2000	N/A	N/A	N/A
netscience	68.42	7195.34	14546.2

Table 5.4: This tables gives the order of data sets in the x-axis in figure 5.3. They have been sorted by time taken for the Neo4j results from table 5.5.2.

Place	Name of Data Set
1	lesmis
2	huck
3	polbooks
4	dolphins
5	barabasi_100_2
6	barabasi_100_3
7	barabasi_100_4
8	adjnoun
9	anna
10	gplus_200
11	pokec_500
12	gplus_500
13	netscience
14	barabasi_1000_2
15	barabasi_1000_3
16	barabasi_1000_4
17	homer
18	pokec_2000
19	gplus_2000
20	barabasi_10000_2
21	barabasi_10000_3
22	barabasi_10000_4

Table 5.5: A table to show the time taken for Cypher, Sparsity and our own C++ implementation to find a solution for the diameter of the instance. A N/A indicates that an out of memory exception occurred.

Graph	CYPHER	Sparsity	Our Implementation
barabasi_100_2	3.84	0.17	0.02
barabasi_100_3	3.74	0.06	0.02
barabasi_100_4	3.73	0.14	0.02
barabasi_1000_2	78.39	22.8	18.55
barabasi_1000_3	80.25	21.78	18.52
barabasi_1000_4	84.75	22.96	18.56
barabasi_10000_2	9150.55	N/A	18304
barabasi_10000_3	9031.16	N/A	18301.8
barabasi_10000_4	9587.51	N/A	18357.2
adjnoun	4.54	0.16	0.03
anna	5.48	0.09	0.6
dolphins	2.32	0.03	<0.01
polbooks	4.13	0.15	0.03
homer	33.96	4.19	3.32
huck	3.00	0.07	0.01
lesmis	2.83	0.07	<0.01
pokec_500	28.78	2.27	2.34
pokec_2000	586.76	84	148.11
gplus_200	8.54	0.36	0.15
gplus_500	30.8	3.27	2.33
gplus_2000	295.18	111.18	148.24
netscience	104.39	21.19	74.47

Table 5.6: The order of labvels on the x-axis in the Graph Diameter figures 5.5.3. They have been sorted by time taken for the Neo4j results from table 5.5.

Place	Name of Data Set
1	dolphins
2	lesmis
3	huck
5	barabasi_100_4
6	barabasi_100_3
7	barabasi_100_2
8	polbooks
9	adjnoun
9	anna
10	gplus_200
11	pokec_500
12	gplus_500
13	homer
14	barabasi_1000_4
15	barabasi_1000_3
16	barabasi_1000_2
17	netscience
18	gplus_2000
19	pokec_2000
20	barabasi_10000_4
21	barabasi_10000_3
22	barabasi_10000_2

My soul is painted like the
wings of butterflies, Fairy tales
of yesterday will grow but never
die, I can fly, my friends...

Freddie Mercury

CHAPTER 6

Conclusion

The overarching aim of this thesis was to explore a variety of graph problems in the form of graph algorithms and graph algorithms in graph databases. A method for this is by exploring algorithms with a varying complexity, from Non-deterministic Polynomial-time Hardness (NP-Hard) to Polynomial (P). We begin with a NP-Hard problem; the Longest Simple Cycle. We then went into depth with a NP-Complete problems: Minimum Dominating Set and the Betweenness Centrality problem; implementing this into graph database systems. We finished by exploring P problems; the Graph Diameter and the Component Labelling algorithm.

Graph databases allow for a type of storage that should be able to combine the mathematical field of graph theory and database literature. By exploring certain

algorithms that come from a wide variety of complex classes, we were able to understand how graph engines react. In the following sections, we conclude the thesis and explain some of the contributions found in the previous chapters.

6.1 The Longest Simple Cycle problem

The first algorithm that was explored in depth was the Longest Simple Cycle problem. Two novel approaches to finding the longest simple cycle were proposed. One is an exact solver based on a Integer Linear Programming (ILP) formulation, and the second a heuristic that improves a simple Depth-First Search (DFS) search.

The Longest Simple Cycle problem is a known NP-Hard problem. Finding a fast and efficient exact solver was always going to be extremely unlikely. However, an ILP formulation was created. The base of the formulation uses a flow mechanic. It was compared to a previous formulation. It was found that the new formulation was significantly faster than previous formulations. However, in large instances it was found to take too long to find a solution. As well as this it struggled with instances that contained a complex branch structure.

Following on from the exact solver, a heuristic based off a simple DFS search was created. The simple DFS search was improved with 4 perturbation improvement operators. The original simple DFS search finds a cycle c. The first perturbation operator goes through each pair of nodes $((n_1, n_2)...(n_n, n_{n+1})$ in cycle c and checks for a triangle. If a triangle exists, then the third node (n_3) must be part of the longest cycle and is added $(...n_1, n_3, n_2..)$. The second operator follows the same procedure,

but instead finds rectangles. The third operator looks for a different path. For example, if (n_1, n_2, n_3) is part of a long cycle c and the path (n_1, n_4, n_2) also exists, then the original path can be replaced, and the first two perturbation operators can be run against the new pathway. The fourth operator is a repeat of the third, except pathways of length two are found.

We compared the heuristic against the exact solver. The optimum results were found for some of the easy and medium instances. As well as that, the heuristic found long cycles that were close to those found by the exact solver for the larger instances, in a fraction of the time.

We then explored implementing the heuristic and exact approaches into database systems. An exact algorithm was created using the query language Cypher, however even for small instances the amount of computational time required was over a week. We found that creating a heuristic was essentially the same as bringing the instance into memory and running the algorithm using a high level language, which is not different to the method above. This was because the in-built functions of DFS search can not be tinkered with using the query languages.

6.2 The Minimum Dominating Set problem

The Minimum Dominating Set problem is a known Non-deterministic Polynomial-time Completeness (NP-Complete) problem. The dominating set of a graph G is a subset S of nodes such that every node in G is either in S or is adjacent to a node in S. There exists a simple greedy heuristic that can find a solution.

The unoptimised greedy heuristic for the minimum dominating set was implemented into two database systems using the query methods provided by the database systems. We used Neo4j and the query language Cypher to represent client-server database types. For embedded database we used Sparsity and the library calls supplied. Cypher is a declarative query language whereas Sparsity uses an imperative approach. This implementation allowed us to investigate the efficiency of the back engines in the systems, the functions provided by the systems and how well the algorithms can be expressed with the given instruments.

It was found that the system engines were not efficient when it came to implementing the greedy heuristic. Cypher is missing some basic fundamental functions, such as a loop mechanic that finishes when a certain condition has been met. In order to compensate for this, a pipeline that handled missing functionality was created. The pipeline was written using an external high-level language.

The results show a noticeable difference in performance of Neo4j when compared to Sparsity and C++. This is likely due to the missing functionality and the pipeline. Larger instances caused Neo4j to crash with an out of memory exception, which shows a reliability on a large amount of memory being present for the algorithms. The expressiveness of the written algorithm was also disappointing; the amount of lines taken to write the algorithm was close to, or even exceeding (depending on the size of the dominating set) to Sparsity's amount of lines.

In contrast to Neo4j and Cypher, Sparsity's query language uses an imperative model. Therefore any functionality that was not present in the libraries was naturally filled in using a high-level language. For a base, a high-level language implementation

of the greedy heuristic written in c++ was used as a comparison.

The results show that Sparsity's database engines were also inefficient when compared to the high level language implementation. The lack of functionality within Sparsity's library meant that there was a dependency on the high level language. However, this is to be expected due to the imperative nature of the language. The library functions that were tested seemed to struggle with larger instances.

6.3 Exploration of other algorithms

The problems covered above do not have any exact solutions, which is expected due to the classification of the problems themselves. A further route of exploration exists with problems that have known efficient exact solutions. The Graph Diameter, Betweenness Centrality and Component Labelling are problems which have consistent solutions, as well as allowing for different sub-tasks within the problems.

All three problems were implemented into Neo4j using the declarative query language Cypher, Sparsity using the imperative libraries supplied and in C++ as a comparison technique. The implementations of the problems took advantage of any in-built functions supplied by the systems.

The shortest path algorithm is a said in-built function of Cypher and Sparsity. The all shortest-path is another function built into Cypher. The Graph Diameter takes advantage of the shortest path function, and the Betweenness Centrality problem takes advantage of the all shortest-path function.

It was found that for procedures that are in-built into the languages such as

the shortest path algorithms are very efficient. This was shown especially in the betweenness centrality results. Simple queries in graphs can be considered NP-Hard, therefore even for some of the smaller instances in the performance results for the diameter problem, it was slower than the high-level implementation. But as the instances grew in size, the efficiency of the in-built functions are shown as the time decreases in comparison to the high-level implementation.

For the consistent algorithms, the expressiveness of Cypher is impressive, with some of the algorithms being written in significantly less amount of lines of code when compared to the imperative implementations in Sparsity.

One drawback found in the experiments was that the systems generally rely on a large amount of memory being available. For some of the larger instances, both of the database systems crashed as not enough memory was available.

6.4 Additional Discussion

The outcomes found in chapter 3 concluded that a heuristic could be created to find a long cycle. An interesting follow up experiment would be implementing the heuristic into the graph database systems in the same manner as in the previous two chapters.

6.4.1 Finding the Longest Cycle in graph database systems

One of the major drawbacks in declarative languages is that internal procedures cannot be altered. Cypher uses exact algorithms to find the results for queries. As

was shown with the component labelling algorithm, procedures cannot be stopped mid-way.

The heuristic created in chapter 3 modifies an existing DFS found cycle. Due to the declarative nature of Cypher, the in-built DFS function is not modifiable using Cypher alone. The perturbation operators cannot be added into the query.

This is an expected outcome. Heuristics that do not provide a consistent answer to a given question, such as the ones in chapter 3 and 4 are difficult to query. Databases must be consistent in data returned when queried. When assuming the data hasn't changed, database queries must produce the same results each time a query has been sent. Implementing heuristic algorithms into graph databases would require this constraint to not exist.

Rather, a query that find the exact solution can be created in cypher. Algorithm 6.4.1 gives an exact solution to finding the longest cycle in an instance written in Cypher for Neo4j.

The *MATCH* clause is used to find a cycle beginning with node n and ending with n. For the edge, an edge *type* must be given. The instances are given a generic edge type, and thus *LINKS_TO* is used. The two magic numbers *1* and *100* represent the lowest and highest cycle to find, not specifying these numbers will cause a Cypher error to occur. For optimisation purposes the *MATCH* clause is saved into a variable p. This allows the ability to sort and limit the final query result.

In an undirected instance, this algorithm would find the longest cycle up to the maximum given magic number. It would be preferable to use the maximum size of the instance. Logically a previous *MATCH* clause to find the size of the instance

could be combined with a *WITH* statement. However, an issue arises when Cypher only allows an unsigned integer for the second magic number. The number could be given programatically with an external language to provide further optimisation.

It was found that none of the instances tested in chapter 3 produced a query result in a reasonable time. The query produces a large search space even with small instances. Another drawback of the algorithm is with a directed graph. As a directed graph involves having two directed edges between nodes, the algorithm can have repeated nodes in the final result.

Algorithm 31 The exact Cypher query for finding the Longest Cycle in a graph. It should be noted that a magic number has been used after the *LINKS_TO* on the first line. This is because an unsigned integer is expected here, Cypher did not allow any variable names to be used.

```
MATCH p = (n) -[:LINKS_TO:1*100]->(n)
RETURN P
ORDER BY length(p) DESC
```

For Sparsity, it is more complicated. While in spirit it uses an imperative query language, the shortest path algorithm is, in essence, a declarative function call. The shortest path algorithm is built into Sparsity libraries. The implementation of the heuristic in Sparsity would be the same as the one explored in chapter 3. The solution would rely solely on memory, and would not interact with the Sparsity's graph backend.

One outcome of the previous chapters is how graph database systems handle algorithms that do not provide consistent results. For example, the minimum dominating set heuristic can provide two solutions to one problem, as shown in figure 6.1. This would explain why the heuristic could be difficult to implement into the query

Figure 6.1: The minimum dominating set of the same graph. The nodes coloured green are both different solutions of the Minimum Dominating Set problem, in accordance to the greedy heuristic.

language.

6.5 Further avenues of exploration

This thesis explores the beginnings of implementing highly computational algorithms into graph database systems. There is scope to further enhance the research in multiple areas. It is relevant to evaluate other types of other hard algorithms. In the case of the longest simple path, further optimisation to the ant-based heuristic could be provided.

Not only that, the two approaches can be applied to a variety of fields, such as biological networks. This allows real world data to be used in evaluating the

performance impact of the algorithms.

As touched upon briefly in chapter 5, parallelisation of graph algorithms is becoming more popular, especially with the rise of computation on GPUs. Being able to parallelise queries and procedures inside of algorithms could speed up query times.

A significant shortcoming found in this thesis is that graph database query methods lack certain instrumentation or functions. This was the case in both client-server and embedded databases. Further research into the effects of these additional procedures would be interesting.

Parallelising graph algorithms inside of graph databases to make use of multi-core GPU's and CPU's could also be further explored in the field. Another area of exploration would be comparing graph database implementations with existing graph libraries such as GraphML. In the graph libraries, similar implementations of the algorithms studied in this thesis can be compared with the c++ implementations, as well as the graph database systems to provide further performance analysis.

Acronym table

Term	Definition
ACID	Atomicity, Consistency, Isolation and Durability
NOSQL	Not Only SQL
JVM	Java Virtual Machine
SQL	Structured Query Language
ILP	Integer Linear Programming
P	Polynomial
IOT	Internet of Things
NP-Complete	Non-deterministic polynomial-time completeness
NP-Hard	Non-deterministic polynomial-time hardness
NP	Non-deterministic polynomial time
TSP	Travelling Salesman Problem
DFS	Depth-First Search
BFS	Breadth-First Search
DBMS	Database Management System

Bibliography

Abreu, R., D. Archer, E. Chapman, J. Cheney, H. Eldardiry, and A. Gascon (2016). Provenance segmentation. In *8th USENIX Workshop on the Theory and Practice of Provenance (TaPP 16)*, Washington, D.C. USENIX Association.

Adamic, L. A. and N. Glance (2005). The political blogosphere and the 2004 u.s. election: Divided they blog. In *Proceedings of the 3rd International Workshop on Link Discovery*, LinkKDD '05, New York, NY, USA, pp. 36–43. ACM.

Albert, R. and A. L. Barabási (2002). Statistical mechanics of complex networks. *Reviews of modern physics 74*(1), 47.

Amann, B. and M. Scholl (1992). Gram: A graph data model and query languages. In *Proceedings of the ACM Conference on Hypertext*, ECHT '92, New York, NY, USA, pp. 201–211. ACM.

Andrews, T., I. Gershoni, R. Imhof, S. Kaufmann, J. Schaerer, T. Studer, and S. Zumbrunn. Efficient stemmatology: a graph database application in the digital humanities.

Angles, R. (2012). A comparison of current graph database models. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering Workshops*, ICDEW '12, Washington, DC, USA, pp. 171–177. IEEE Computer Society.

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

Angles, R. and C. Gutierrez (2005). *Querying RDF Data from a Graph Database Perspective*, pp. 346–360. Berlin, Heidelberg: Springer Berlin Heidelberg.

Angles, R. and C. Gutierrez (2008, February). Survey of graph database models. *ACM Comput. Surv. 40*(1), 1:1–1:39.

Arnborg, S. and A. Proskurowski (1989). Linear time algorithms for NP-hard problems restricted to partial k-trees. *Discrete Applied Mathematics 23*(1), 11–24.

Aung, M. (1989). Longest cycles in triangle-free graphs. *Journal of Combinatorial Theory, Series B 47*(2), 171–186.

Bachman, M. (2013). Graphaware: Towards online analytical processing in graph databases.

Balaghan, P., K. A. Hawick, D. Chalupa, and C. Maddra (2017, August). Dominating set algorithm implementation in graph databases. Technical Report CSI-0014, Computer Science, University of Hull, Cottingham Road, Hull, HU6 7RX.

Barabási, A.-L. and R. Albert (1999). Emergence of scaling in random networks. *Science 286*(5439), 509–512.

Barceló, P., L. Libkin, and J. L. Reutter (2011). Querying graph patterns. In *Proceedings of the Thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '11, New York, NY, USA, pp. 199–210. ACM.

Barceló Baeza, P. (2013). Querying graph databases. In *Proceedings of the 32Nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '13, New York, NY, USA, pp. 175–188. ACM.

Becker, M. Y. and I. Rojas (2001). A graph layout algorithm for drawing metabolic pathways. *Bioinformatics 17*(5), 461–467.

Beis, S., S. Papadopoulos, and Y. Kompatsiaris (2015). Benchmarking graph databases on the problem of community detection. In N. Bassiliades, M. Ivanovic, M. Kon-Popovska, Y. Manolopoulos, T. Palpanas, G. Trajcevski, and A. Vakali (Eds.), *New Trends in Database and Information Systems II*, Cham, pp. 3–14. Springer International Publishing.

Bianconi, G. and M. Marsili (2005). Loops of any size and Hamilton cycles in random scale-free networks. *Journal of Statistical Mechanics: Theory and Experiment 2005*(06), P06005.

Biggs, N., E. K. Lloyd, and R. J. Wilson (1986). *Graph Theory, 1736-1936*. New York, NY, USA: Clarendon Press.

Björklund, A., T. Husfeldt, and S. Khanna (2004). Approximating longest directed paths and cycles. In *Automata, Languages and Programming*, pp. 222–233. Springer.

Bodlaender, H. L. (1993). On linear time minor tests with depth-first search. *Journal of Algorithms 14*(1), 1–23.

Bonami, P., L. T. Biegler, A. R. Conn, G. Cornuéjols, I. E. Grossmann, J. Laird, C. D.and Lee, A. Lodi, F. Margot, N. Sawaya, et al. (2008). An algorithmic framework for convex mixed integer nonlinear programs. *Discrete Optimization 5*(2), 186–204.

Bronson, N., Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani (2013). Tao: Facebook's distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, San Jose, CA, pp. 49–60. USENIX.

Buerli, M. (2012). The current state of graph databases.

Bursztyn, V. S., M. G. Nunes, and D. R. Figueiredo (2016). How congressmen connect: Analyzing voting and donation networks in the brazilian congress.

Castelltort, A. and A. Laurent (2016). Rogue behavior detection in nosql graph databases. *Journal of Innovation in Digital Ecosystems 3*(2), 70 – 82.

Chalupa, D. (2017, April). An Order-based Algorithm for Minimum Dominating Set with Application in Graph Mining. *ArXiv e-prints*.

Chalupa, D., P. Balaghan, and K. A. Hawick (2018, Jan). A Probabilistic Ant-based Heuristic for the Longest Simple Cycle Problem in Complex Networks. *arXiv e-prints*, arXiv:1801.09227.

Chalupa, D., P. Balaghan, K. A. Hawick, and N. A. Gordon (2017). Computational methods for finding long simple cycles in complex networks. *Knowledge-Based Systems 125*, 96 – 107.

Chen, F. and K. Li (2015). Detecting hierarchical structure of community members in social networks. *Knowledge-Based Systems 87*, 3–15.

Chen, G., Z. Gao, X. Yu, and W. Zang (2005). Approximating the longest cycle problem on graphs with bounded degree. In *Computing and Combinatorics*, pp. 870–884. Springer.

Chvatal, V. (1979). A greedy heuristic for the set-covering problem. *Mathematics of Operations Research 4*(3), 233–235.

Clausen, J. (2003). Branch and bound algorithms - principles and examples.

Codd, E. F. (1970, June). A relational model of data for large shared data banks. *Commun. ACM 13*(6), 377–387.

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, New York, NY, USA, pp. 151–158. ACM.

Corbellini, A., C. Mateos, A. Zunino, D. Godoy, and S. Schiaffino (2017). Persisting big-data: The nosql landscape. *Information Systems 63*, 1 – 23.

Cruz, I. F., A. O. Mendelzon, and P. T. Wood (1987, December). A graphical query language supporting recursion. *SIGMOD Rec. 16*(3), 323–330.

Cruz, I. F., A. O. Mendelzon, and P. T. Wood (1988). G+: Recursive queries without recursion. In *Expert Database Conf.*

Data, C. (1975). *An introduction to database systems*. Addison-Wesley publ.

DBEngine (2017a). Graph database usage.

DBEngine (2017b). Rankings.

DBEngine (2017c). Rankings of graph databases.

DBEngine (2017d). Rankings of graph databases.

Dijkstra, E. W. (1959, December). A note on two problems in connexion with graphs. *Numer. Math. 1*(1), 269–271.

Dineen, M. J., M. Khosravani, and A. Probert (2011). Using opencl for implementing simple parallel graph algorithms. In *Proc. PDPTA'11*, pp. 1–6.

Dinh, T. N., Y. Shen, D. T. Nguyen, and M. T. Thai (2014). On the approximability of positive influence dominating set in social networks. *Journal of Combinatorial Optimization 27*(3), 487–503.

Dixon, E. T. and S. E. Goodman (1976). An algorithm for the longest cycle problem. *Networks 6*(2), 139–149.

Dominguez-Sal, D., P. Urbón-Bayes, A. Giménez-Vañó, S. Gómez-Villamor, N. Martínez-Bazán, and J. L. Larriba-Pey (2010). *Survey of Graph Database Performance on the HPC Scalable Graph Analysis Benchmark*, pp. 37–48. Berlin, Heidelberg: Springer Berlin Heidelberg.

Drakopoulos, G. (2016, July). Tensor fusion of social structural and functional analytics over neo4j. In *2016 7th International Conference on Information, Intelligence, Systems Applications (IISA)*, pp. 1–6.

Ejov, V., J. Filar, and J. Gondzio (2004). An interior point heuristic for the Hamiltonian cycle problem via Markov decision processes. *Journal of Global Optimization 29*(3), 315–334.

Elmasri, R. and S. Navathe (2010). *Fundamentals of Database Systems* (6th ed.). USA: Addison-Wesley Publishing Company.

Euler, L. (1736). Solutio problematis ad geometriam situs pertinentis. *Commentarii Academiae Scientiarum Imperialis Petropolitanae 8*, 128–140.

Feder, T., R. Motwani, and C. Subi (2002). Approximating the longest cycle problem in sparse graphs. *SIAM Journal on Computing 31*(5), 1596–1607.

Floyd, R. W. (1962, June). Algorithm 97: Shortest path. *Commun. ACM 5*(6), 345–.

Francis, N., A. Green, P. Guargliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, M. Schuster, P. Selmer, and A. Taylor (2018, February). Formal Semantics of the Language Cypher. *ArXiv e-prints*.

Fredman, M. L. and R. E. Tarjan (1987, July). Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM 34*(3), 596–615.

Freeman, L. C. (1977). A set of measures of centrality based on betweenness. *Sociometry*, 35–41.

Freeman, L. C. (1978). Centrality in social networks conceptual clarification. *Social networks 1*(3), 215–239.

Fukunaga, K. and P. M. Narendra (1975, July). A branch and bound algorithm for computing k-nearest neighbors. *IEEE Transactions on Computers C-24*(7), 750–753.

Gabow, H. N. (2007). Finding paths and cycles of superpolylogarithmic length. *SIAM Journal on Computing 36*(6), 1648–1671.

Gallian, J. A. (2005, December). A dynamic survey of graph labeling. *The Electronic Journal of Combinatoris 16*, DS6.

Garey, M. R. and D. S. Johnson (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co.

Gavish, B. and S. C. Graves (1978). The traveling salesman problem and related problems.

Gebremedhin, A. H. and F. Manne (2000). Scalable Parallel Graph Coloring Algorithms. *Concurrency: Practice and Experience 12*, 1131–1146.

Gemis, M. and J. Paredaens (1993). *An object-oriented pattern matching language*, pp. 339–355. Berlin, Heidelberg: Springer Berlin Heidelberg.

Girvan, M. and M. E. J. Newman (2002). Community structure in social and biological networks. *Proceedings of the National Academy of Sciences 99*(12), 7821–7826.

Gray, J. (1981). The transaction concept: Virtues and limitations (invited paper). In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81, pp. 144–154. VLDB Endowment.

Gray, J. and A. Reuter (1992). *Transaction Processing: Concepts and Techniques* (1st ed.). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Gremlin (2017). Gremlin home page.

Gross, J. L., J. Yellen, and P. Zhang (2013). *Handbook of Graph Theory, Second Edition* (2nd ed.). Chapman & Hall/CRC.

Gutfraind, A. and M. Genkin (2016). A graph database framework for covert network analysis: An application to the islamic state network in europe. *Social Networks*, –.

Güting, R. H. (1994). Graphdb: Modeling and querying graphs in databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, San Francisco, CA, USA, pp. 297–308. Morgan Kaufmann Publishers Inc.

Gyssens, M., J. Paredaens, J. van den Bussche, and D. van Gucht (1994, Aug). A graph-oriented object database model. *IEEE Transactions on Knowledge and Data Engineering 6*(4), 572–586.

Haerder, T. and A. Reuter (1983, December). Principles of transaction-oriented database recovery. *ACM Comput. Surv. 15*(4), 287–317.

Harrison, W. K. (2016). The role of graph theory in system of systems engineering. *IEEE Access 4*, 1716–1742.

Hawick, K. and H. James (2007). Node importance ranking and scaling properties of some complex road networks.

Hawick, K., H. James, and C. Scogings (2007, 4-6 December). Structural Circuits and Attractors in Kauffman Networks. In H. A. Abbass and M. Randall (Eds.), *Proc. Third Australian Conference on Artificial Life*, Volume 4828 of *LNCS*, Gold Coast, Australia, pp. 189–200. Springer. 978-3-540-76930-9.

Hawick, K. A. (2007, August). Exploring data structures and tools for computations on graphs and networks. Technical Report CSTN-043, Computer Science, Massey University.

Hawick, K. A. (2011, 7-9 November). Applying enumerative, spectral and hybrid graph analyses to biological network data. In *Int. Conf. on Computational Intelligence and Bioinformatics (CIB 2011)*, Pittsburgh, USA, pp. 89–96. IASTED.

Hawick, K. A. (2012a, 25-27 June). Betweenness centrality metrics for assessing electrical power network robustness against fragmentation and node failure. In *Proc. International Conference on Power and Energy Systems (EuroPES 2012)*, Napoli, Italy., pp. 186–193. IASTED.

Hawick, K. A. (2012b, 12-14 November). Node-failure and islanding in national grid scale electricity distribution networks. In *Proc. Int. Conf. Power and Energy Systems and Applications*, Las Vegas, pp. 52–58. IASTED.

Hawick, K. A. (2012c, 3-5 September). Water distribution network robustness and fragmentation using graph metrics. In *Proc. Int. Conf. on Water Resource Management (AfricaWRM 2012)*, Number 762-037, Gabarone, Botswana, pp. 304–310. IASTED. CSTN-158.

Hawick, K. A. (2014, July). Middleware and software architectures for managing irregular and dynamic crowdsourced data and sensor networks. CSI 0005, Department of Computer Science, University of Hull, Robert Blackburn Building, Cottingham Road, Hull, UK.

Hawick, K. A. and H. A. James (2008, 14-17 July). Enumerating circuits and loops in graphs with self-arcs and multiple-arcs. In *Proc. 2008 Int. Conf. on Foundations of Computer Science (FCS'08)*, Las Vegas, USA, pp. 14–20. CSREA.

Hawick, K. A., A. Leist, and D. P. Playne (2010a). Parallel graph component labelling with gpus and cuda. *Parallel Computing 36*, 655–678.

Hawick, K. A., A. Leist, and D. P. Playne (2010b, December). Parallel Graph Component Labelling with GPUs and CUDA. *Parallel Computing 36*(12), 655–678.

He, H. and A. K. Singh (2008). 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*, SIGMOD '08, New York, NY, USA, pp. 405–418. ACM.

Hedar, A.-R. and R. Ismail (2010). *Hybrid Genetic Algorithm for Minimum Dominating Set Problem*, pp. 457–467. Berlin, Heidelberg: Springer Berlin Heidelberg.

Hidders, J. and J. Paredaens (1994). *Goal, a Graph-Based Object and Association Language*, pp. 247–265. Vienna: Springer Vienna.

Hölsch, J. and M. Grossniklaus (2016). An algebra and equivalences to transform graph patterns in neo4j. In *EDBT/ICDT 2016 Workshops: EDBT Workshop on Querying Graph Structured Data (GraphQ)*.

Holzschuher, F. and R. Pein (2016). Querying a graph database – language selection and performance considerations. *Journal of Computer and System Sciences 82*(1, Part A), 45 – 68. Special Issue on Query Answering on Graph-Structured Data.

Holzschuher, F. and R. Peinl (2013). Performance of graph query languages: Comparison of cypher, gremlin and native access in neo4j. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, New York, NY, USA, pp. 195–204. ACM.

Howe, B., D. Halperin, F. Ribalet, S. Chitnis, and E. V. Armbrust (2013). Collaborative science workfwork in sql. *Computing in Science and Engineering 15*, 22–31.

Iordanov, B. (2010). *HyperGraphDB: A Generalized Graph Database*, pp. 25–36. Berlin, Heidelberg: Springer Berlin Heidelberg.

Johnson, D. B. (1975). Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing 4*(1), 77–84.

Johnson, D. S. and M. Trick (1996). *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*. Providence, RI: American Mathematical Society.

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

Karger, D., R. Motwani, and G. D. S. Ramkumar (1997). On approximating the longest path in a graph. *Algorithmica 18*(1), 82–98.

Karp, R. M. (1972). *Reducibility among Combinatorial Problems*, pp. 85–103. Boston, MA: Springer US.

Ke, Y., J. Cheng, and W. Ng (2008, Dec). Efficient correlation search from graph databases. *IEEE Transactions on Knowledge and Data Engineering 20*(12), 1601–1615.

Knuth, D. E. (1993). *The Stanford GraphBase: A Platform for Combinatorial Computing*. Reading, MA: Addison-Wesley.

Kuper, G. M. and M. Y. Vardi (1993). The logical data model. *ACM Transactions on Database Systems (TODS) 18*(3), 379–413.

Land, A. H. and A. G. Doig (2010). *An Automatic Method for Solving Discrete Programming Problems*, pp. 105–132. Berlin, Heidelberg: Springer Berlin Heidelberg.

Lee, C.-H. and C.-W. Chung (2014). Efficient search in graph databases using cross filtering. *Information Sciences 286*(Supplement C), 1 – 18.

Leist, A. and K. A. Hawick (2009, 13-16 July). Circuits as a classifier for small-world network models. In *Proc. WORLDCOMP 2009 International Conference on Foundations of Computer Science (FSC 09) Las Vegas, USA*, Number CSTN-003.

Leist, A. and K. A. Hawick (2011, 18-21 July). Graph generation on gpus using dynamic memory allocation. In *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'11)*, Number PDP3939, Las Vegas, USA, pp. 229–235. CSREA.

Leskovec, J. and A. Krevl (2014, June). SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`.

Levene, M. and A. Poulovanssilis (1991, May). An object-oriented data model formalised through hypergraphs. *Data Knowl. Eng. 6*(3), 205–224.

Levene, M. and A. Poulovassilis (1990, Oct). The hypernode model and its associated query language. In *Information Technology, 1990. 'Next Decade in Information Technology', Proceedings of the 5th Jerusalem Conference on (Cat. No.90TH0326-9)*, pp. 520–530.

Linderoth, J. T. and A. Lodi (2011). MILP software. *Wiley encyclopedia of operations research and management science*.

Lissandrini, M., M. Brugnara, and Y. Velegrakis (2017). An evaluation methodology and experimental comparison of graph databases.

Little, J. D. C., K. G. Murty, D. W. Sweeney, and C. Karel (1963, December). An algorithm for the traveling salesman problem. *Oper. Res. 11*(6), 972–989.

Liu, K., J. Huang, H. Sun, M. Wan, Y. Qi, and H. Li (2015). Label propagation based evolutionary clustering for detecting overlapping and non-overlapping communities in dynamic networks. *Knowledge-Based Systems 89*, 487–496.

Lloyd, J. W. Practical advantages of declarative programming.

Lusseau, D., K. Schneider, O. J. Boisse, P. Haase, E. Slooten, and S. M. Dawson (2003). The bottlenose dolphin community of doubtful sound features a large proportion of long-lasting associations. *Behavioral Ecology and Sociobiology 54*(4), 396–405.

Macko, P., D. Margo, and M. Seltzer (2013). Performance introspection of graph databases. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, New York, NY, USA, pp. 18:1–18:10. ACM.

Malewicz, G., M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski (2010). Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, New York, NY, USA, pp. 135–146. ACM.

Marinari, E., G. Semerjian, and V. Van Kerrebroeck (2007). Finding long cycles in graphs. *Physical Review E 75*(6), 066708.

Marr, B. (2015). 20-mind boggling facts everyone must read.

Martínez-Bazan, N., V. Muntés-Mulero, S. Gómez-Villamor, J. Nin, M.-A. Sánchez-Martínez, and J.-L. Larriba-Pey (2007). Dex: High-performance exploration on large graphs for information retrieval. In *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*, CIKM '07, New York, NY, USA, pp. 573–582. ACM.

Marton, J., G. Szárnyas, and D. Varró (2017). Formalising opencypher graph queries in relational algebra. *arXiv preprint arXiv:1705.02844*.

McColl, R. C., D. Ediger, J. Poovey, D. Campbell, and D. A. Bader (2014). A performance evaluation of open source graph databases. In *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*, PPAA '14, New York, NY, USA, pp. 11–18. ACM.

Mendelzon, A. O. and P. T. Wood (1995, December). Finding regular simple paths in graph databases. *SIAM J. Comput. 24*(6), 1235–1258.

Miller, C. E., A. W. Tucker, and R. A. Zemlin (1960, October). Integer programming formulation of traveling salesman problems. *J. ACM 7*(4), 326–329.

Miller, J. J. (2013). Graph database applications and concepts with neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, Volume 2324.

Münch, F. (2017). graph-processing-betweeness-centrality-neo4js-cypher-vs-graphstream.

Monien, B. (1985). How to find long paths efficiently. *North-Holland Mathematics Studies 109*, 239–254.

Needham, M. (2013). Betweeness centrality cypher vs graphstream.

Neo4j (2017a). Analyzing panama papers.

Neo4j (2017b). Cypher query language.

Neo4j (2017c). Home.

Neo4j (2018). Property graph.

Newman, M. E. J. (2006). Finding community structure in networks using the eigenvectors of matrices. *Physical Review E 74*(036104), 036104–1–036104–19.

Nishizeki, T., T. Asano, and T. Watanabe (1983). An approximation algorithm for the hamiltonian walk problem on maximal planar graphs. *Discrete Applied Mathematics 5*(2), 211–222.

Nowozin, S. and C. H. Lampert (2011, March). Structured learning and prediction in computer vision. *Found. Trends. Comput. Graph. Vis. 6*(3–4), 185–365.

Opatrny, J. (1979). Total ordering problem. *SIAM Journal on Computing 8*(1), 111–114.

OrientDB (2017). Property graph.

Papadimitriou, C. H. and K. Steiglitz (1982). *Combinatorial Optimization: Algorithms and Complexity*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Playne, D. P. and K. Hawick (2018, June). A new algorithm for parallel connected-component labelling on gpus. *IEEE Transactions on Parallel and Distributed Systems 29*(6), 1217–1230.

Potluri, A. and A. Singh (2013). Hybrid metaheuristic algorithms for minimum weight dominating set. *Applied Soft Computing 13*(1), 76 – 88.

Rath, M., D. Akehurst, C. Borowski, and P. Mader (2012, February). Are graph query languages applicable for requirements traceability analysis? In *Joint Proceedings of REFSQ-2017 Workshops, Doctoral Symposium, Research Method Track, and Poster Track (REFSQ-JP 2017)*, pp. 1289–1292.

Robinson, I., J. Webber, and E. Eifrem (2013). *Graph Databases*. O'Reilly Media, Inc.

Rodriguez, M. (2010). The graph traversal programming pattern.

Rodriguez, M. A. and P. Neubauer (2010). The graph traversal pattern. *arXiv preprint arXiv:1004.1001*.

Salwinski, L., C. S. Miller, A. J. Smith, F. K. Pettit, J. U. Bowie, and D. Eisenberg (2004). The database of interacting proteins: 2004 update. *Nucleic acids research 32*(suppl 1), D449–D451.

Sarwat, M., S. Elnikety, Y. He, and G. Kliot (2012, April). Horton: Online query execution engine for large distributed graphs. In *2012 IEEE 28th International Conference on Data Engineering*, pp. 1289–1292.

Seaborne, A. and E. Prud'hommeaux (2008, January). SPARQL query language for RDF. W3C recommendation, W3C. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

Shao, B., H. Wang, and Y. Li (2013). Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, New York, NY, USA, pp. 505–516. ACM.

Sheng, L., Z. M. Ozsoyoglu, and G. Ozsoyoglu (1999, Mar). A graph query language and its query processing. In *Proceedings 15th International Conference on Data Engineering (Cat. No.99CB36337)*, pp. 572–581.

Stonebraker, M. and R. Cattell (2011, June). 10 rules for scalable performance in 'simple operation' datastores. *Commun. ACM 54*(6), 72–80.

Takac, L. and M. Zabovsky (2012). Data analysis in public social networks. In *International Scientific Conference and International Workshop Present Day Trends of Innovations*, pp. 1–6.

Tamassia, R. (2013). *Handbook of graph drawing and visualization*. CRC press.

Tarjan, R. (1973). Enumeration of the elementary circuits of a directed graph. *SIAM Journal on Computing 2*(3), 211–216.

Tarry, G. (1895). Le probleme des labyrinthes. *Nouvelles annales de mathématiques: journal des candidats aux écoles polytechnique et normale 14*, 187–190.

Tausch, N., M. Philippsen, and J. Adersberger (2011). A statically typed query language for property graphs. In *Proceedings of the 15th Symposium on International Database Engineering & Applications*, IDEAS '11, New York, NY, USA, pp. 219–225. ACM.

Tomaszuk, D. (2016). *RDF Data in Property Graph Model*, pp. 104–115. Cham: Springer International Publishing.

Uehara, R. and Y. Uno (2007). On computing longest paths in small graph classes. *International Journal of Foundations of Computer Science 18*(05), 911–930.

Urma, R.-G. and A. Mycroft (2015). Source-code queries with graph databases—with application to programming language usage and evolution. *Science of Computer Programming 97*, 127 – 134.

van Rest, O., S. Hong, J. Kim, X. Meng, and H. Chafi (2016). Pgql: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, pp. 7. ACM.

Vesdapunt, N. and H. Garcia-Molina (2015, April). Identifying users in social networks with limited information. In *2015 IEEE 31st International Conference on Data Engineering*, pp. 627–638.

Vicknair, C., M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins (2010). A comparison of a graph database and a relational database: A data provenance perspective. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, New York, NY, USA, pp. 42:1–42:6. ACM.

Vincent, L. and P. Soille (1991, Jun). Watersheds in digital spaces: an efficient algorithm based on immersion simulations. *IEEE Transactions on Pattern Analysis and Machine Intelligence 13*(6), 583–598.

Wagner, I., A. M. Bruckstein, et al. (1999). Hamiltonian (t)-an ant-inspired heuristic for recognizing Hamiltonian graphs. In *Proceedings of the 1999 Congress on Evolutionary Computation, 1999. CEC 99*, Volume 2. IEEE.

Waltz, D., C. Stanfill, S. Smith, and R. Thau (1987). Very large database applications of the connection machine system. TMC Technical Note DR87-3, Thinking Machines Corporation.

Watts, D. J. (1999). *Small worlds: the dynamics of networks between order and randomness*. Princeton university press.

Watts, D. J. and S. H. Strogatz (1998). Collective dynamics of "small-world" networks. *Nature 393*(6684), 440–442.

White, D. R. and S. P. Borgatti (1994). Betweenness centrality measures for directed graphs. *Social Networks 16*(4), 335 – 346.

Wood, P. T. (2012, April). Query languages for graph databases. *SIGMOD Rec. 41*(1), 50–60.

Wu, J. and H. Li (1999). On calculating connected dominating set for efficient routing in ad hoc wireless networks. In *Proceedings of the 3rd International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, DIALM '99, New York, NY, USA, pp. 7–14. ACM.

Xenarios, I., E. Fernandez, L. Salwinski, X. J. Duan, M. J. Thompson, E. M. Marcotte, and D. Eisenberg (2001). Dip: the database of interacting proteins: 2001 update. *Nucleic acids research 29*(1), 239–241.

Xenarios, I., D. W. Rice, L. Salwinski, M. K. Baron, E. M. Marcotte, and D. Eisenberg (2000). Dip: the database of interacting proteins. *Nucleic acids research 28*(1), 289–291.

Xenarios, I., L. Salwinski, X. J. Duan, P. Higney, S. M. Kim, and D. Eisenberg (2002). Dip, the database of interacting proteins: a research tool for studying cellular networks of protein interactions. *Nucleic acids research 30*(1), 303–305.

Yang, F., R. Zhang, Y. Yao, and Y. Yuan (2016). Locating the propagation source on complex networks with propagation centrality algorithm. *Knowledge-Based Systems 100*, 112–123.

Yang, S., Y. Xie, Y. Wu, T. Wu, H. Sun, J. Wu, and X. Yan (2014). Slq: A user-friendly graph querying system. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, New York, NY, USA, pp. 893–896. ACM.

Yannakakis, M. (1990). Graph-theoretic methods in database theory. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '90, New York, NY, USA, pp. 230–242. ACM.

Zachary, W. W. (1977). An information flow model for conflict and fission in small groups. *Journal of Anthropological Research 33*, 452–473.

Zheng, W., L. Zou, X. Lian, H. Zhang, W. Wang, and D. Zhao (2014). Sqbc: An efficient subgraph matching method over large and dense graphs. *Information Sciences 261*(Supplement C), 116 – 131.

Zhu, Y. and E. Yan (2016). Searching bibliographic data using graphs: A visual graph query interface. *Journal of Informetrics 10*(4), 1092 – 1107.