Maximal Clique Variants

David Steiner

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

A dissertation presented in part fulfilment of the requirements of the Degree of Master of Science at The University of Glasgow

08/09/2014

Abstract

A clique in a graph is a subgraph in which all vertices are adjacent. A maximal clique is a clique that is not contained in any other clique. Maximal clique enumeration is one of the longstanding problems of graph theory which has been extensively researched. There are variants of the problem which have been investigated in less detail. This dissertation addresses two variants, the maximal k-clique and the maximal labelled clique problem. The concept of k-cliques is a relaxation of the original model, allowing a distance of k between the vertices of the clique, rather than requiring direct adjacency. Maximal k-cliques can be enumerated by maximal clique algorithms. Computational results of a notable maximal clique algorithm, adapted for the k-clique variant, are presented to analyse whether this approach is viable in practice. Labelled cliques are a recently introduced variant of cliques, in which labels are assigned to edges. The maximum labelled clique problem is the problem of finding the largest labelled cliques that are within a budget, that is, the number of labels in the clique is limited to be less than or equal to a constant. Based on this, the maximal labelled clique problem is formulated and a new algorithm is proposed for it. A set of benchmark results for real life datasets is presented to show the practical feasibility of the algorithm.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Dr. Patrick Prosser, for his continuous support throughout the project. Also, I would like to extend my gratitude onto Ciaran McCreesh.

Finally, I wish to thank my family for the motivation and encouragement which enabled me to pursue the Masters degree at the University of Glasgow.

Contents

Chapter 1

Introduction

The clique problem is formulated in terms of undirected graphs, graphs in which the connection between vertices are two-way. A simple example of such a graph is a graph representing a social network, in which nodes are people and edges represent acquaintanceship. In this social network, a clique is a set of people who all know each other. Maximal cliques are all such groups that cannot be enlarged by another person, that is, there are no other people who they collectively know. The maximal clique enumeration problem is the problem of finding all these groups in a graph.

Clique enumeration is one of the famous, long standing problems of graph theory. Due to its countless theoretical and practical uses, it has been researched for decades, yet it is still being focussed on today. The number of cliques in a graph is exponential as a function of the graph's size, therefore all algorithms devised for the problem inevitably take exponential time to run. However, for real life data, there exist algorithms that can enumerate cliques in relatively large graphs.

Real world applications of the problem include a wide range of areas. For one, social networks are an obvious area. Boginski et al. used a maximal clique algorithm to find maximal independent sets in a graph (this is an equivalent problem), which in turn allowed them to count the number of completely diversified portfolios in a financial market [4]. Koch used it to solve a related problem, finding maximal common edge graphs and maximal common induced graphs, which are used in computational biology to find structural motifs in protein structures [12]. These are just a few of the notable application areas.

This project deals only in part with the maximal clique enumeration problem. The major topic of it is more recent variations of the problem, for which less literature exists. It is concerned with two problems for which the solution is closely related to the maximal clique problem. The objective was to find out how the changes made to the base model affect the hardness of the problem. For the original problem, we know that many practical graphs are technically feasible to enumerate with state-of-the-art algorithms. Is it also true for its variants as well?

One of these related problems is the maximal k-clique problem. K-cliques are like cliques, but (to stick with the social groups example) instead of only allowing direct acquaintance, we allow a 'distance' of k between people. For instance, a 2-clique is a group of people

who all either know each other, or have a mutual friend. This concept was the first clique relaxation problem [19] originally formulated by Luce and has its roots in sociology [14]. The k-clique problem , in comparison to the plain maximal clique problem, is known to be harder. Despite the definition was proposed over 60 years ago, there is still not nearly as much literature available for it. The project sets out to find out if the problem can be solved solved applying the most recent clique algorithms in a feasible manner.

The other area researched is the maximal labelled clique problem. Labelled cliques were introduced recently by Carrabs et al. [7], who also defined the maximum labelled clique problem. The maximum labelled clique problem assigns labels to edges and looks for maximum cliques in the graph following two rules. First rule being that the clique's cost has to be in the budget, that is, the number of different labels in the clique cannot exceed a given constant. Second rule; if more such maximum cliques are found, the cost is to be minimised. That is, of two maximum cliques (they have the same size), the one using less labels is preferred. In the project, this definition is modified to formulate the maximal labelled clique problem. For this problem no algorithms have been proposed yet. As part of the project, based on a notable maximal clique algorithm, a new algorithm is devised to solve it. Benchmark results will demonstrate whether this algorithm can solve the problem in reasonable time.

The main objective of the project is to demonstrate the hardness of the maximal clique problem variants through vigorous testing. The dissertation consists of 6 main chapters following this introduction. Chapter 2 explains the definitions used throughout the dissertation, formulation of the problems and preexisting solutions. Chapter 3 describes the infrastructure that was developed for the project: components of the core library and the code to carry out the experiments. It also describes the data that was used throughout the testing phase. Chapter 4 highlights the implemented algorithms' correctness by verifying their output. Their efficiency is also demonstrated by benchmarking against real and random datasets. Chapter 5 focusses on the k-clique problem, summarising the observations which have been made about the hardness of the problem. Moreover, Chapter 6 offers a solution to the maximal labelled clique problem, and analyses whether it is efficient for real-life applications. Lastly, Chapter 7 will summarise the results of the project and describes how it could be further improved with potential extension.

Chapter 2

Survey and problem definitions

2.1 Preliminaries

2.1.1 Graphs and notation

Graphs model a set of objects (represented by vertices) and possible connections between them. The connections are represented by edges. There may be directions assigned to the edges or the edges can be undirected. Cliques are defined in undirected graphs, and in the scope of this project graphs mean undirected graphs. Also, all the edges in the considered graphs are proper edges, and the graphs do not contain multi-edges. The set of vertices will usually be denoted by V and the set of edges will be denoted by E:

Definition. *An (undirected)* ***graph*** $G = (V, E)$ *is a mathematical structure consisting of the set V of vertices and the set E of edges, which are unordered pairs of elements of V, its endpoints.*

$N(v)$ denotes the neighbourhood of a $v \subseteq V$, the set of vertices adjacent to v. The degree, denoted by $deg(v)$, is the cardinality of $N(v)$. A complete graph is a graph in which every vertex is joined by an edge.

An often-used term throughout the dissertation is the density of a graph. Since all considered graphs are undirected simple graphs, the density of a graph is defined as $D(G(V, E)) = \frac{2|E|}{|V|(|V|-1)}$.

2.1.2 Cliques and its variants

Maximal and maximum cliques

A clique is a complete subgraph, a subset of V in which all vertices are pairwise connected by an edge. A maximal clique is a clique that cannot be further extended by adding more vertices to it. A maximum clique is the largest clique in a graph. Figure 2.1 shows an

example of cliques in a small graph: {4,6}, {1,2}, {1,5}, {2,3}, {2,5}, {3,4}, {4,5}, {4,6} and {1,2,5} are examples of cliques. Maximal cliques are {4,6}, {2,3}, {3,4}, {4,5}, {4,6} and {1,2,5}, as the other cliques can all be extended by a vertex to form {1,2,5}. {1,2,5} is also a maximum clique, as it is the largest clique with a size of 3.

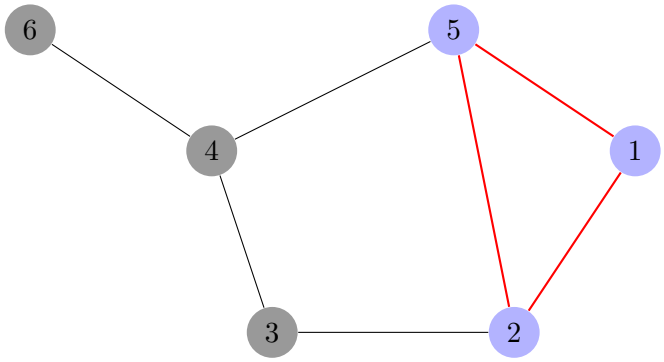

Fig. 2.1: A small graph and its maximum clique. [1]

K-cliques

The following terms are discussed in detail in Chapter 5, but they are also referred to before that, therefore they are briefly described here as well.

Cliques are suitable to model a wide range of real life phenomena. However, it is often too strict a restriction to allow only direct connections between the vertices. Often it is desirable to allow a distance of k between them, where distance means the length of the shortest path between two vertices and is denoted by $dist(v_i, v_j)$. There are two approaches to this relaxation: the distance can be calculated within the clique or in the whole graph. The structures that are generated by the former method are called $k-clubs$, and are not considered further in this project. The latter approach defines $k-cliques$, a major area of research of this dissertation.

Definition. *A* ***k-clique*** *in a graph* $G = (V, E)$ *is a subgraph* $C \subseteq V$*, in which* $dist(v_i, v_j) \leq k : v_i, v_j \in C$.

Furthermore, an important notation closely related to k-cliques is the power graph of a graph G, denoted by G^k, in which an edge between two vertices exists if and only if the length of the shortest path between the two **in the original graph** is less than or equal to k.

Labelled Cliques

Carrabs et al. [7] introduced a variant to the maximum clique problem. They assign a label to each edge in the graph and they seek to find the largest cliques in which the number of different labels of the edges are limited by a given constant 'b' (budget). Among cliques with the same size, the one with the minimum number of different labels is preferred.

Enumerating maximum labelled cliques is named the Maximum Labelled Clique (MLC) problem. An illustration of this concept (as opposed to the maximum clique problem) is shown in Figure 2.2.

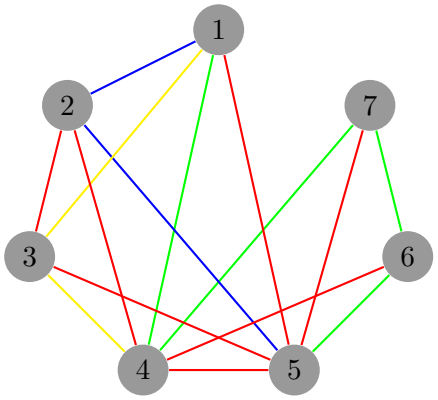

Fig. 2.2: Edge labels are represented by different colours. With unlimited budget, the maximum clique is [1,2,3,4,5]. If the budget is only three, two maximum cliques are [2,3,4,5] and [4,5,6,7] of which the latter is preferred due to using less labels. This figure is the courtesy of Ciaran McCreesh and Patrick Prosser.

Analogously to the simple clique problem, the maximal version of the problem can be formulated. In this case, preferring cliques with the lower cost (number of different labels) makes less sense, as we want to enumerate all maximal cliques. However, similarly to the maximum problem, we can specify a budget, and only look for cliques that are within the budget, and cannot be extended. This can be due to two reasons. Either there are no vertices left which is adjacent to all vertices in the growing clique, or there is one, but adding it would result in exceeding the budget.

2.2 Maximal Clique Enumeration Algorithms

2.2.1 A review of algorithms

Maximal Clique Enumeration is one of the classic problems of graph theory with plenty of literature available on the subject. There have been many algorithms proposed to solve it, however they all take exponential time to run. J. W. Moon and L. Moser have shown that there can exist up to $O(3^{n/3})$ cliques in a graph of size n [17]. Due to this, exponential time is the best one can hope for in general purpose algorithms.

Notable early algorithms dating back to the 70's include the output sensitive algorithm of Tsukiyama et al. [22] and the input sensitive 'Bron-Kerbosch' algorithm devised by Coenraad Bron and Joep Kerbosch [6]. Tsukiyama's algorithm is bounded by $O(nm\mu)$, where n is the number of vertices, m is the number of edges and μ is the number of cliques. It has been shown that the Bron-Kerbosch algorithm can be modified to run in $O(3^{n/3})$ time [21], which is optimal in a sense due to the observation of J. W. Moon and L. Moser.

Many of the modern algorithms are based on either of these two early algorithms. In

practice, the Bron-Kerbosch algorithm has been found to be very effective, and several state-of-the-art algorithms are a modification of it.

2.2.2 The Bron-Kerbosch algorithm

The algorithm traverses the vertices of the graph in a tree-like structure, searching for possible cliques. Once a branch is found to be impossible to result in a maximal clique or a maximal clique is found, it uses backtracking to abandon/return from that branch.

To achieve this, three sets are defined. C is a partial solution, an incrementally grown clique which hopes to eventually extend into a maximal clique. P is the candidate set, which maintains all the vertices that are adjacent to every $v \in C$, meaning that if they are added to C, C still forms a clique. Whenever a vertex is added to C, P shrinks accordingly.

Algorithm 1: The Bron Kerbosch algorithm [6]

```
1 BK(Set C, Set P, Set X)
2 begin
3     if P = ∅ and X = ∅ then report C as maximal clique
4     for v ∈ P do
5         BK(C ∪ {v},P ∩ N(v),X ∩ N(v))
6         P ← P \ {v}
7         X ← X ∪ {v}
```

The candidate set is initially set to V, the vertex set of the input graph. Starting a recursive search in every vertex of P could find all the maximal cliques. However, it is not efficient and it would find the same clique multiple times. To avoid this, Bron and Kerbosch defined another set, denoted by X (also referred to as the NOT set hereafter), that contains all the vertices that at some time have served as an extension to C. These vertices must be excluded from the search.

The cornerstone of the algorithm is the recursive backtracking function. It takes the above sets as parameter, iterates through the vertices in the candidate set – adding them to the partial clique. When such a vertex $v \subseteq P$ is added to C, P and X are shrunk discarding all vertices that are not in the neighbourhood of v. The function is then called recursively for the new sets of C, P and X. Upon return, v is moved from the candidate set to X, excluding it from further calls in the branch.

How does the algorithm find maximal cliques? Clearly, P keeps getting smaller and smaller, eventually becoming empty. At that point, if X is not empty, then the clique we found is a subset of a larger clique we had found in an earlier branch and we must discard it. However, if X is empty, a new maximal clique is found. This shows how introducing X prevents from producing duplicates and decreases the number of recursive calls.

2.2.3 Introducing a pivot

The number of recursive calls can be furher reduced and the running speed enhanced by a heuristic called pivoting. If one picks a vertex $v_p \subseteq P \cup X$, then every maximal clique must contain a vertex not adjacent to v_p or v_p itself. If a pivot is chosen carefully at every call, this observation can greatly reduce the number of recursive calls.

The original article by C. Bron and J. Kerbosch chooses the pivot based on the following observation: if at some stage $P \subseteq N(v)$ for any vertex v in X, then extending the partial clique by vertices from P will never remove v from X and therefore can never lead to a maximal clique. This can be utilised to minimise the number of recursive calls at a stage: the pivot is chosen to minimise the number of recursive call, the vertex from $P \cup X$ that has the least amount of non-neighbours in P.

Tomita et al. propose a nearly identical way to choose the pivot [8, 21], so that $|P \cap N(v_p)|$ is maximised (Algorithm 2). By using a different output format, they show that using this pivot selection ensures an optimal time complexity of $O(3^{n/3})$. [1]

Algorithm 2: Tomita's pivot selection [21]

```
1 BKTomita(Set C, Set P, Set X)
2 begin
3     if P = ∅ and X = ∅ then report C as maximal clique
4     u ← max(|P ∩ N(u)|), u ∈ P ∪ X
5     for v ∈ P \ N(u) do
6         BKTomita(C ∪ {v},P ∩ N(v),X ∩ N(v))
7         P ← P \ {v}
8         X ← X ∪ {v}
```

2.2.4 Initial vertex ordering

Eppstein et al. focused on enumerating cliques in large sparse graphs [9]. They devised an algorithm whose time complexity is bounded by a function of the graph's degeneracy, which is a common measurement of the density of graphs and tends to be small for sparse graphs.

Definition. *The **degeneracy ordering** of a graph is an ordering of its vertices so that each vertex has d or fewer neighbours that come later in the ordering, where d is the degeneracy of the graph.*

They prove that their algorithm's time complexity is $O(dn3^{d/3})$. The algorithm is a modification of Tomita's algorithm, replacing the outer level of the algorithm with a loop that processes the vertices in their degeneracy ordering. The inner levels remain unchanged, using Tomita's pivot selection.

[1] Optimal as a function of the graph's size in the sense that there may be up to $3^{n/3}$ cliques in the graph due to Moon and Moser [17].

Algorithm 3: The algorithm of Eppstein et al. [9]

```
1 BKDegeneracy(Graph G(V, E))
2 begin
3     for v_i ∈ V in a degeneracy ordering do
4         P ← N(v_i) ∩ {vi + 1, .., v_{n−1}}
5         X ← N(v_i) ∩ {v_0, .., v_{i−1}}
6         BKTomita({v_i}, P, X)
```

A major difference between Eppstein's and Tomita's implementation is that Tomita used an adjacency matrix representation of the graph, while Eppstein used adjacency lists. The former is much less efficient for large sparse graphs, as the adjacency matrix representation has a space complexity of $O(n^2)$ regardless of the number of edges. Due to this, their algorithm runs into memory issues for relatively small graphs. The main motivation for Eppstein's algorithm is not to surpass Tomita's runtime efficiency for all types of graphs, but to offer an alternative solution for large sparse graphs that Tomita's algorithm could not handle in terms of memory usage.

Chapter 3

Codebase and data

3.1 Core library

3.1.1 Overview of packages

The core library consists of classes that serve general purposes and are used by multiple executables. It includes implementations of clique algorithms and other algorithms, classes to represent the graphs and utility classes to help reading in data and generating graphs. It is made up of four core packages:

algorithm Has all the code for clique algorithms using different implementations and graph representations, various pivot selections and vertex ordering algorithms.

graph Contains the data structures that represent the data (graphs and vertices).

labelled Everything related to the labelled clique problem.

utility Helper classes and methods.

3.1.2 Implementation of clique algorithms

This section summarises the different implementations of the Bron-Kerbosch algorithm implemented for the project. These algorithms can all be found in the **algorithm** package. While the algorithms could be implemented as static methods, all of them were implemented as a separate Java class. The classes store statistics about the algorithm's results, such as the number of recursive calls (*nodes*) it took to find all maximal cliques and the running time, which are available via accessor methods once the algorithm has been executed. These are all available through the *CliqueAlgorithm* superclass, the algorithm class that all implementations extend.

A straightforward implementation

The first and probably the most simple attempt at implementing the Bron-Kerbosch algorithm was to use the standard collections provided by Java. In this variant, the graph is represented by storing the adjacency list for all vertices in an *ArrayList*. The three sets of the algorithm, C, P and X are also stored in a Java *ArrayList*. As these are actually sets, I also considered using a $HashSet$, but it turned out to be much slower than using an *ArrayList*. Also, throughout the algorithm, we never check if a vertex is in a set which could justify the use of a $HashSet$.

Although this implementation is more readable than other implementations and is quite straightforward, unfortunately it proved to be very slow. Due to its slowness, it was not used for any benchmarking purposes. Nonetheless, it served as a reference against which other implementations could be verified. It is kept in the codebase for the sake of completeness and can be found in the $algorithm.other\_impl.TomitaAdjacencyList$ class.

Bron-Kerbosch style

Bron and Kerbosch used an adjacency matrix representation for the graph and a single array to represent P and X. This array is an integer array of length n that stores all vertices, marked by their index. Since P and X are disjoint sets, two integers can keep track of the starting position of X and P. Moving vertices from or to a set can be achieved by swapping the right labels and moving the pointers to the starting positions accordingly.

They kept track of C in the form of a stack-like global integer array. As in my code this algorithm is implemented as a class, this array is stored as an instance variable. A detailed explanation of this and an Algol60 source code can be found in the original article [6]. The converted Java implementations is in $algorithm.other\_impl.BKPointers$.

Eppstein's algorithm

Eppstein et al. use a very specific implementation for their algorithm which makes their algorithm efficient despite the adjacency list representation of the graph. In order for their algorithm to be compared on fair grounds, a conversion of the algorithm they call *degen* – using the linear space data structure – was implemented in Java (they implemented the algorithm in C). For details of this implementation I refer to the articles of Eppstein et al. [9, 10]. For the Java conversion code, please see the $algorithm.eppstein\_impl$ package.

BitSet algorithms

The major part of clique algorithm code is the BitSet algorithm. This implementation is the most often used one in the project, used for most experiments, for k-clique enumeration and serving as a base for the labelled clique algorithm, too. The Java source code for this algorithm is included in Appendix B.1.

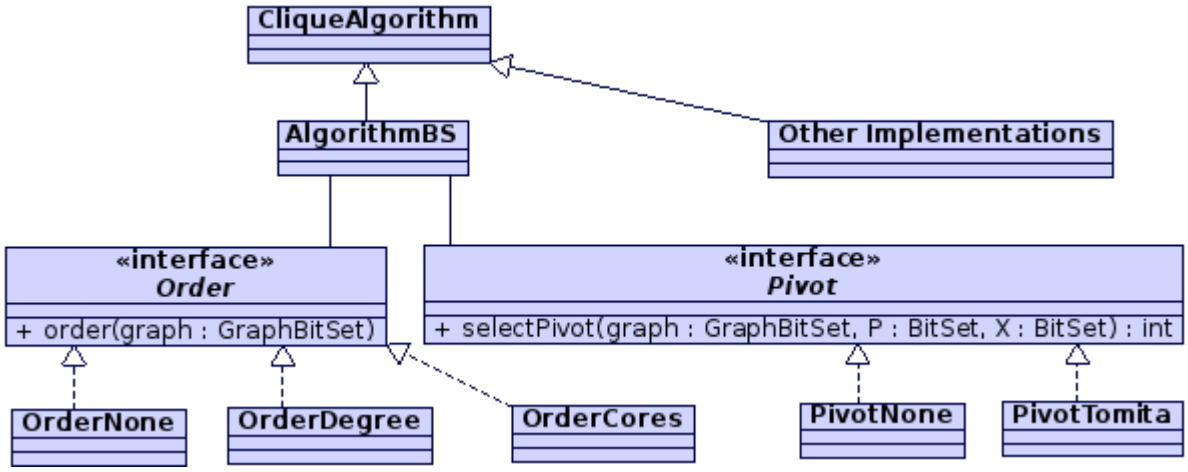

Fig. 3.1: A simplified overview of the implementation of the BitSet algorithm with ordering and pivot selection.

The adjacency matrix and the three sets alike are represented using the standard BitSet implementation of Java. There are certain theoretical limitations of this. For instance, iterating through P takes $O(n)$ time instead of an efficient representation that would require a time proportional to the size of P. However, the BitSet implementation of Java is very fast and it can often make up for such shortcomings in practice. Using BitSet provides convenient solutions with regards to set operations. For instance, getting $X \cap N(v)$ is as simple as writing $X.and(graph.neighbours(v))$. Union operations can be done by the $or()$ operation and complement operations by $andNot()$. As a result, the implementation is rather short and readable.

For this implementation, no specific pivot selection or initial ordering was implemented in the algorithm itself. Rather every instance of the *AlgorithmBS* class has a *Pivot* and an *Order* field. These are interfaces that have a single method which chooses the pivot and orders the vertices, respectively. This allows to separate the pivot selection and ordering implementation from the algorithm itself, allowing for any variations of choice and ordering. Also, this structure is flexible and enables the implementation of new orderings and pivots separately from the main algorithm. The class structure of the algorithm is shown in Figure 3.1. Classes ending in *None* are slightly confusing: *OrderNone* returns the vertices' original order (as they are in the graph), however, *PivotNone* does not mean that there is no pivot selection. Rather, it chooses the first element from P. This takes very little time and helps reduce the number of elements in P and is superior to leaving out pivoting entirely.

The ordering algorithm of *OrderDegree* and *OrderCores* are implementations of the algorithm devised by V. Batagelj and M. Zaversnik [3]. This is a efficient, dedicated algorithm for core decomposition (i.e. degeneracy), but it also orders the vertices by their degree along the way, so it can be modified to find the degree ordering as well.

Both pivoting and ordering take *GraphBitSet* as parameter. This class is the BitSet implementation of a graph. For every vertex $v \in V$, $N(v)$ is a BitSet, with the bits corresponding to neighbours set to 1, non-neighbours set to 0. Finding out if v and w are neigbours takes constant time: we just need to check the bit's value at index w in

the neighbourhood BitSet of v. This is rarely needed though, as in the Bron-Kerbosch algorithm the neighbourhood of a vertex is only used to calculate $N(v) \cap P$ and $N(v) \cap X$. Assuming variable nV stores the neighbours of v, these operations can simply be carried out by calling $nV.and(P)$; and $nV.and(X)$; respectively. These operations are very fast, much faster than iterating through the bits in a loop for the same effect.

The BitSet based classes are in the *algorithm.bitset_impl* package.

3.1.3 The Code for K-Clique Enumeration

To enumerate k-cliques, the above algorithms work without any modification, given that we have an algorithm to construct a power graph for the graph (see Chapter 5). Solutions to build the power graph can be found in the **utility** package, in class *PowerGraph*. The class is an abstract class containing static methods to construct the graphs.

As I did not have any results against which the correctness of the algorithms could have been verified, five different algorithms were implemented to achieve the same output. They do produce the same power graphs, which supports the claim that they work well. Also, having various algorithms offers the possibility to choose the quickest one.

For the list of algorithms and details about their implementation, see Section 5.3.

3.1.4 Labelled cliques

A similar approach to k-cliques does not work for labelled cliques. For labelled clique enumeration one needs a new graph representation for *labelled graphs* and new algorithms to find maximal cliques, too. Classes for both are in package **labelled**, namely *LabelledGraph* and classes starting with BKL, along with other helper classes that are used in carrying out verification, tests and experiments. Labelled clique enumeration is not discussed further at this point. Chapter 6 deals with the problem in details.

3.1.5 Helper code

In addition to the algorithms and data structures, the library has classes that help with reading in graphs, generate random graphs and carry out minor operations. For random graph generation, it provides means to generate graphs specified by the size and edge probability or by the size and the exact number of edges to have. The later will had significance when performing the experiments for k-cliques.

3.2 Experiments

For this project, many experiments had to be conducted. Much of the code was developed to run these experiments specifically and does not belong to the core library. These include executable classes that produce job files, which, in turn, provide a way to produce many

results at once, in an automatised and reproducable fashion. They also include executable classes that aggregate and output the results in a way that is usable for human analysis, and readable by applications that were used to produce this dissertation (i.e. Latex and gnuplot). Please see Appendix A for details about the experimentation process.

Since some experiments took many days (even week, on some occasions) to finish, it would have been very inconvenient to run them on my personal laptop. Instead, they were carried out on a university server machine. The machine was accessible from the school network via ssh and I was granted exclusive access, therefore the experiments could run without any interference. The CPU of the machine has sixteen cores, which allowed many experiments to be run at the same time.

3.3 Graph instances used in the project and the format of graph files

The following list contains the list of datasets used for benchmarks with their description and references:

BioGRID [20] Biological General Repository for Interaction Datasets. It is a collection of datasets that "archives and disseminates genetic and protein interaction data from model organisms and humans".

DIMACS instances [11] A collection of computationally challenging graphs from the second DIMACS challenge, commonly used for benchmarking.

Mark Newman dataset [18] A compilation of free datasets gathered from various scientific sources. It has graphs from a wide range of application areas, such as social networks, collaborations of scientists, books and blogs.

Moon-Moser graphs [17] Moon-Moser graph instances used by Tomita et al. [21] and Eppstein et al. [10] for benchmarking.

SNAP [13] Stanford Large Network Dataset Collection. A collection of data about large social and information networks. These graphs are larger and sparser than other datasets.

The data from these sources is stored in plain text files, each text file containing a single graph. The format of the files is the following. The first line is the number of vertices, the second line is twice the number of edges. Subsequent rows contain the edges, each row having two integers separated by a comma. These two numbers are the indices of the vertices that are connected by this edge. Edges are duplicated, so if there is an edge connecting v_i and v_j, then there will be two lines showing this: 'i,j' and 'j,i'. To avoid having duplicate code for the same function, random graphs generated by my code follow the same format.

Labelled graphs are stored similarly to unlabelled graphs with slight differences. The second line in the text file stores the number of labels, instead of the number of edges. Edges are stored in the same way, with the label appended to each row as a third integer. In this case, edges are not duplicated.

3.4 Conclusion

This chapter gave a high level view and structure of the packages that were programmed for the project. These include clique algorithms, helper algorithms, data structures (such as the BitSet graph) and file I/O functions. There are also separate executable (that is, classes with main methods) applications that provide means to run the experiments in an automatised fashion.

The benefits of carrying out experiments automatically in the aforementioned way were demonstrated. It reduces the probability of introducing human errors in the processing, stores every subresult generated during the experiment and makes it possible for any substep to be rerun, reproduced. This is very important as it makes the result traceable, followable step by step.

The code contains methods to generate random graphs, but we also want to use real life data from external data sources. Three data sources are used throughout the project, which are described and referenced in Section 3.3.

Chapter 4

Maximal Clique Experiments

4.1 Overview

Even though the overall focus of the project is variations of the maximal clique problem, the first step is to test the algorithms on the classic maximal clique enumeration problem. Since the same algorithm is used for k-clique enumeration as for the classic problem, and the same algorithm is modified for the labelled clique problem, it is crucial that the base algorithms work properly. To ensure this, a number of experiments were dedicated to verifying their correctness.

The verification process consisted of two steps. First step was to generate a set of random graphs, and check that all algorithms (of which 3 are completely unrelated implementations) yield the same number of cliques. The number of maximal cliques were counted by the algorithms over 50 random graphs at 5 different settings (varying edge probability, size fixed at a hundred). This gives a total number of 250 graphs, for which all algorithms gave the same results. Second step was to test real life datasets. For the classic maximal clique problem published results exist for many datasets. For example, Eppstein et al. [10] published results for the BioGRID and Mark Newman datasets. The number of cliques is shown in Table 4.2 and 4.1. Second step was to check whether the algorithms' output match such published results. It was found that most results are the same as the ones published by Eppstein et al. However, there were some difference, my algorithms found:

- 39,288 in *internet*, in contrast to their results, 39,275
- 1,386 in *celegens*, they reported 856

In these instances, the number of edges they listed for their graph also differed, so they may have used different data from mine. No differences were found in the BioGRID dataset. Overall, the results gave enough confidence to accept the hypothesis that the algorithms indeed are correct.

Once the algorithms were verified, I could move on to research their behaviour and show that my BitSet implementation is efficient and a reasonable choice for further usage in the

project. (that is, for the k-clique problem and the labelled clique problem). Experiments were carried out to observe the effects of pivot selection and initial ordering. The algorithms were benchmarked to show that the BitSet algorithm is viable for all different kinds of data used.

4.2 Real-world data

Results will be shown for 6 algorithms:

BSCT BitSet algorithm with core ordering for the outer call and Tomita's pivot selection.

BSDT BitSet algorithm with degree ordering and Tomita's pivot selection.

BSNN BitSet algorithm with arbitrary pivot selection (no ordering of vertices).

BSNT BitSet algorithm with Tomita's pivot (no ordering of vertices).

EPPS Eppstein's implementation

PT Bron-Kerbosch style implementation with the original pivot selection (no ordering).

These algorithms cover the most important options. First observation with regards to runtimes is that all algorithms are very much viable. However, there are differences, which, in some cases can be rather significant. While extremely fast for small graphs, **PT** struggled with some of the larger graphs. For instance, it took it 40 times longer to enumerate cliques in *condmat* – which is apparently the hardest of the Mark Newman graphs – than Eppstein's implementation (Table 4.1). Interestingly, for the other algorithms, *yeast* from BioGRID was the hardest. For **PT** this was much easier than *condmat*, taking only a little over two seconds, which made it the fastest one for *yeast*. This may be due to the fact that *yeast* has many less vertices than *condmat* for roughly the same number of edges. Again, **PT** appears to perform much better on smaller graphs.

Data	n	m	cliques	BSCT	BSDT	BSNN	BSNT	EPPS	PT
marknewman-internet	22,963	48,436	39,288	0.986	0.907	0.753	0.716	0.542	6.783
marknewman-adjnoun	112	425	303	0.014	0.013	0.009	0.010	0.016	0.007
marknewman-astro	16,706	121,251	15,794	1.103	1.056	8.182	1.048	0.621	6.673
marknewman-dolphins	62	159	84	0.005	0.004	0.003	0.004	0.009	0.003
marknewman-condmat	40,421	175,693	34,274	2.443	2.713	4.164	2.682	0.935	38.931
marknewman-polblogs	1,490	16,715	49,884	0.764	0.824	1.379	0.456	0.332	0.156
marknewman-celegens	297	2,148	1,386	0.047	0.049	0.048	0.047	0.080	0.039
marknewman-lesmis	77	254	59	0.008	0.008	0.008	0.006	0.011	0.004
marknewman-football	115	613	281	0.020	0.019	0.017	0.018	0.023	0.010
marknewman-netscience	1,589	2,742	741	0.044	0.051	0.040	0.044	0.063	0.157
marknewman-power	4,941	6,594	5,687	0.088	0.076	0.090	0.096	0.169	0.638
marknewman-karate	34	78	36	0.002	0.002	0.001	0.001	0.006	0.000
marknewman-polbooks	105	441	199	0.009	0.015	0.007	0.007	0.017	0.005

Table 4.1: Algorithm runtimes in seconds (Mark Newman dataset)

Data	n	m	cliques	BSCT	BSDT	BSNN	BSNT	EPPS	PT
biogrid-fission-yeast	2,031	12,637	28,520	262	262	500	295	198	178
biogrid-fruitfly	7,282	24,894	21,995	0.328	316	268	320	377	1.415
biogrid-yeast	6,008	156,945	738,613	6.312	4.833	37.670	5.507	2.155	2.005
biogrid-human	9,527	31,182	23,863	0.285	285	260	274	266	1.314
biogrid-worm	3,518	6,531	5,652	0.077	0.061	0.052	0.061	0.112	0.191
biogrid-plant	1,745	3,098	2,302	0.048	0.042	0.069	0.062	0.070	0.131
biogrid-mouse	1,455	1,636	1,523	0.034	0.035	0.018	0.029	0.038	0.066

Table 4.2: Algorithm runtimes in seconds (BioGRID dataset)

The choice of initial ordering seems to have very little effect both on runtimes and the number of nodes (Table 4.3). Eppstein's algorithm being the exception, which relies heavily on the degeneracy ordering in its implementation. But for the BitSet implementation it does not seem to make a major difference. This may be due to the fact, that in the BitSet implementation, the initial ordering only affects the outer call, but at the inner calls, vertices are still processed in their original order. It appears that pivot selection has a much greater impact. While, with regards to speed, Tomita's pivot selection is not always better, it does consistently reduce the number of nodes visited (Table 4.3). The reason why this does not always result in improved speed too, is that finding the optimal pivot using the BitSet representation is an expensive operation. In many cases the benefits of reducing the number of calls does not make up for the increased time per call.

The benefits of Tomita's pivoting is the most apparent in *marknewman−astro* where it reduces the nodes from around 2.6 million to only about 85 thousand, but also *marknewman−polblogs* (2.16 million to around 122 thousand) and *marknewman−condmat* (720 thousand to around 160 thousand). For these graphs, this also shows in runtimes: **BSNT** is about 8 times, 3 times and 2 times faster than **BSNN** for these graphs, respectively. But these numbers also show that the performance gain is not as good as the decrease in recursive calls due to the extra effort of finding the pivot.

Data	n	m	cliques	BSCT	BSDT	BSNN	BSNT	EPPS	PT
marknewman-internet	22,963	48,436	39,288	82,786	83,331	163,625	80,074	82,846	69,672
marknewman-adjnoun	112	425	303	561	558	755	481	562	463
marknewman-astro	16,706	121,251	15,794	83,533	84,338	2,633,377	91,694	82,786	79,333
marknewman-dolphins	62	159	84	203	195	248	185	204	170
marknewman-condmat	40,421	175,693	34,274	160,659	162,298	720,345	178,813	159,694	144,986
marknewman-polblogs	1,490	16,715	49,884	122,206	125,255	2,162,171	124,474	122,920	122,262
marknewman-celegens	297	2,148	1,386	3,129	3,146	6,234	3,038	3,123	2,796
marknewman-lesmis	77	254	59	230	237	588	192	235	188
marknewman-football	115	613	281	655	659	1,284	645	652	610
marknewman-netscience	1,589	2,742	741	2,982	2,991	5,147	3,016	2,982	2,428
marknewman-power	4,941	6,594	5687	11,156	11,149	11,886	11,181	11,158	9,391
marknewman-karate	34	78	36	100	100	103	73	100	69
marknewman-polbooks	105	441	199	548	547	999	512	551	486

Table 4.3: Algorithm nodes (Mark Newman dataset)

It may seem to be an error that the number of nodes is not the exact same for **BSCT** and **EPPS**, which are implementations of the exact same algorithms. But it is not, there are only slight differences, which are the result of the different data structures, because of which vertices are processed in a slightly different order. Both **EPPS** and **PT** use a representation where vertices are often swapped when moved between sets. Due to this, the very same processing order cannot be achieved with BitSets in an efficient implementation.

It is worth to note the speed of Eppstein's implementation. It is very efficient over all data sets, and usually by far the fastest for the harder graphs. The results reproduce their claims [10] that it is indeed as efficient as Tomita's algorithm for real life data, despite using much less memory for sparse graphs due to the adjacency list representation.

4.3 Random data

Random graphs can behave differently from real life data. The objective of this section is to analyse the algorithms' efficiency on random graphs and compare it to the results observed in the previous section. The two data sets analysed are graphs of size 80 (as for this size,

graphs with any density are feasible to enumerate) and larger sparse graphs of size 5000 and 7500.

Graphs with 80 vertices are small, smaller than most real life data in our data sets. However, a very dense small graph can be just as challenging as a large sparse graph. Figure 4.1 shows the runtimes of the algorithms for densities from 0.05 to 0.95, the runtimes are in milliseconds. While the runtime for real life data rarely exceeded a couple of seconds (slowest being 39 seconds for *condmat*), enumerating dense graphs of size 80 took more than 100 minutes on many occasions.

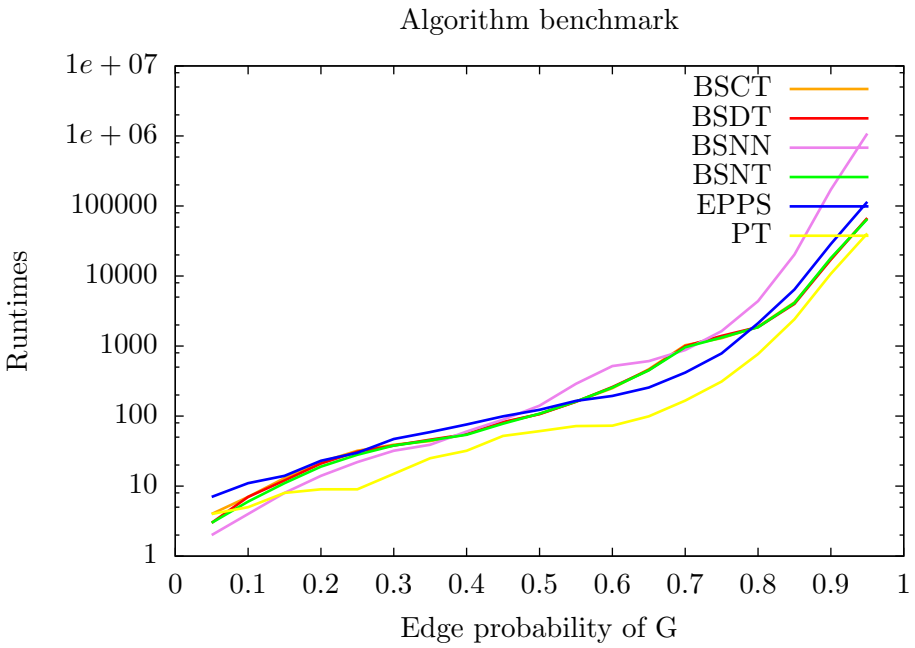

Fig. 4.1: Algorithm runtimes for random graphs of size 80.

Generally, **PT** performed much better on random small graphs than on real life graphs. This is not entirely surprising; it was the fastest among the algorithm for small graphs of the Mark Newman data sets too. As the density is increased, the inefficiency of the random pivot selection becomes more obvious. For very dense graphs, it is more than ten times slower than the same algorithm with Tomita's pivot selection. The other algorithms' performance is similar to what was observed for real life data. **EPPS** and the BitSet algorithms all perform well with Tomita's pivot. Again, the initial vertex ordering does not make a major difference.

Size	p	BSCT	BSDT	BSNN	BSNT	EPPS	PT
5000	0.005	0.351	0.331	0.267	0.332	0.408	0.514
	0.01	0.667	0.668	0.519	0.708	0.582	0.644
	0.015	1.104	1.042	0.664	1.034	0.793	0.835
7500	0.005	0.851	0.813	0.574	0.788	0.765	1.139
	0.01	1.339	1.414	0.822	1.579	1.301	1.689
	0.015	2.360	2.395	1.353	2.404	2.116	2.178

Table 4.4: Algorithm times in seconds for sparse random graphs.

For sparse graphs, the results are slightly different. Table 4.4 shows that for the analysed

graphs, **BSNN** beats **BSNT** in every case, even if only by a small margin. This is unlike the case of dense graphs, where the reduced number of nodes could make up for the extra cost of choosing the pivot. In sparse graphs, the well chosen pivot does not result in a much reduced number of recursive calls (see Table 4.5). Now, finding the pivot is not very efficient with the BitSet implementation. Calculating $P \cup X$ takes $O(n)$ time, then we iterate through every vertex in the union BitSet which takes $O(n)$ time again. For every bit set to 1 (the number of 1's can be any value from 0 to n), we then calculate $P \cap N(v)$, taking linear time again. Finally, the cardinality of the resulting set is counted in linear time. Therefore, the runtime complexity of this operation is $2n + n(n + n) \sim O(n^2)$. Apparently, the reduced number of calls is not enough in this case to even out the cost of this calculation.

Size	p	BSNN	BSNT
5000	0.005	69,771	66,898
	0.01	149,391	142,114
	0.015	260,090	248,371
7500	0.005	156,160	149,591
	0.01	355,975	341,506
	0.015	660,430	635,863

Table 4.5: In sparse graphs, Tomita's pivot selection does not reduce the number of nodes by a great extent.

4.4 Memory limitations of the BitSet algorithm

The SNAP dataset is a collection of very large, sparse graphs and it would have been nice to be able to test the algorithms on them as well. However, most of the graphs were too large to enumerate in terms of memory usage (with 6 gigabytes of RAM allocated for the heap). This rendered the experiment described in 4.2 useless, as most of the time the programme exited with raising an out of memory exception. Therefore, instead of running benchmark tests, I just measured which graphs fit in memory from the SNAP graphs. Table 4.6 summarises the results. Note, that the memory usage of the BitSet representation of

Data	n	m	cliques	Could be enumerated?
amazon0601	403,394	2,443,408	*	No
berkstan	685,231	6,649,470	*	No
cit-Patents	3,774,768	16,518,947	*	No
email-Eu	265,214	364,481	*	No
email-enron	36,692	183,831	226,859	Yes
epinions1	75,888	405,740	1,775,074	Yes
google	875,713	4,322,051	*	No
roadNet-CA	1,965,206	2,766,607	*	No
roadNet-PA	1,088,092	1,541,898	*	No
roadNet-TX	1,379,917	1,921,660	*	No
slashdot0902	82,168	504,230	890,041	Yes
wiki-Talk	2,394,385	4,659,565	*	No
wiki-Vote	7,115	100,762	459,002	Yes

Table 4.6: Memory issues with the SNAP dataset.

graphs does not depend on the number of edges, only the size of the graph. It is very

inefficient for such sparse graphs, like the ones in the SNAP dataset. It appears that there are no issues below a hundred thousand vertices. Over that, every attempt failed to run the algorithm (in fact, the exception is raised when the graph is being loaded – running the algorithm itself would not take much extra memory). In conclusion, memory usage is a significant drawback of the adjacency matrix representation when the data we wish to model is very sparse.

4.5 Conclusion

The correctness of the algorithms was verified through their output: they all match, and they match the results found in published articles. In addition to their correctness, it had to be decided whether they are reasonably fast too. All 6 algorithms (of which 4 use the same base algorithm, the BitSet implementation) were found to be viable.

For small graphs, the code converted from the original Bron-Kerbosch algorithm source code was very efficient, slightly faster on average than any other algorithm. For larger graphs of over a thousand vertices, it often struggled to keep up with the other algorithms. The other algorithm that was significantly slower on some data sets, was the algorithm with random pivoting instead of Tomita's pivot selection. In some cases it was 10 times slower than the same algorithms with the better pivot selection.

The initial vertex ordering did not have a similar significant impact on their efficiency. For the BitSet implementation with Tomita's pivot, various orders were tested. Although they resulted in small changes in the number of calls and runtimes, they did not change the outcome to a great extent. The pivot selection does change the outcome. For dense graphs, choosing a good pivot vastly reduces the number of calls. However, finding the pivot with the BitSet implementation takes $O(n^2)$ which is fine for small dense graphs. But for sparse graphs, where the pivot selection has a smaller effect on the recursive calls, and for which n is greater, a fast arbitrary pivot selection was found to be faster.

In conclusion, choosing the pivot as described in the original Bron-Kerbosch article [6] or Tomita's article [21] is well worth it even for BitSet algorithms, despite finding the pivot being a costly operation. Both the BitSet algorithm and Eppstein's algorithm are consistently fast over various data sets. For further benchmarks, **BSNT** algorithm was used, as it was found to perform well and as it is very flexible.

Chapter 5

K-cliques

5.1 Description

In a clique, vertices are joined pairwise by an edge, i.e. the distance between vertices is one. This requirement could be relaxed to allow a distance of larger than one. There are two ways this can be done: the distance can be calculated within the subgraph (the diameter of the subgraph has to be less than or equal to k), in which case we are talking about k-clubs, or the distance is calculated using all edges of the graph, resulting in k-cliques.

Definition. *A **k-clique** in a graph $G = (V, E)$ is a subgraph $C \subseteq V$, in which $dist(v_i, v_j) \leq k : v_i, v_j \in C$.*

Definition. *The **diameter** of a graph $G = (V, E)$ is $diam(G) = \max_{v_i, v_j \in G} dist(v_i, v_j)$.*

Definition. *A **k-club** in a graph $G = (V, E)$ is a subgraph $C \subseteq V$, such that $diam(C) \leq k$.*

The maximum k-clique problem is possible to solve with algorithms used for solving the maximum clique problem due to a simple observation: the problem of finding maximum k-cliques in a given G graph can be reduced to finding maximum cliques in G^k; where G^k is defined as $G^k = (V, E^k)$, $E^k = \{(v_1, v_2) : 0 \leq d_G(v_1, v_2) \leq k\}$ [2]. Therefore, one can use for example Tomita's algorithm to find maximum k-cliques.

We are faced with more difficulties regarding the maximum k-club problem. The root of the difficulties lies in the non-hereditary nature of k-clubs: if we remove a vertex from a k-club, the remaining subgraph is not necessarily a k-club [19], which renders an approach similar to the one used for k-cliques useless. Nonetheless, exact algorithms for the problem exist, notably the algorithm of Bourjolly et al. [5], but they are less efficient than maximum clique algorithms. Significantly less literature is available on the maximum k-club problem than on the maximum clique problem, and there is even less literature to be found on maximal k-club enumeration.

For simplicity, the focus of the dissertation is problems that can be reduced to the maximal clique problem, and existing maximal clique algorithms can be used or modified for their enumeration. As the maximal k-club problem is not such a problem, it is out of this project's

scope. However, using the aforementioned technique, the maximum k-clique problem can easily be reduced to the maximum clique problem. It will be shown that the same is true for the maximal cliques.

5.2 Enumerating maximal k-cliques

First, it must be shown that analogously to maximum cliques, the maximal k-clique problem can indeed be reduced to finding the maximal cliques in the power graph. This ensures that maximal clique algorithms can solve the maximal k-clique problem too.

Statement 1. *Every maximal k-clique in $G = (V, E)$ is a maximal clique in G^k, and every maximal clique in G^k is a maximal k-clique in G.*

Proof. The first statement can easily be shown. Let $C \subseteq V$ be a maximal k-clique in G. By definition, $d_G(v_i, v_j) \leq k$ for all $v_i, v_j \in C$, therefore, every vertices in C are pairwise joined by an edge in G^k, so they form a clique. If this clique wasn't maximal, that would mean that there is a $v \in V$ which is not in C and $C \subseteq N(v)$. However, this would mean that there is a $v \notin C$ for which $d_G(v, v_i) \leq k : \forall v_i \in C$, contradicting with the maximality of C in G.

The other direction can be shown as follows. Let $C \subseteq V$ be a maximal clique in G^k, meaning that $d_G(v_i, v_j) \leq k$ for all $v_i, v_j \in C$, therefore by definition, C is a k-clique in G. C is a maximal clique in G^k, therefore, there is no vertex $v \notin C$ for which $d_G(v, v_i) \leq k : \forall v_i \in C$, so C is a maximal k-clique in G. □

Theoretically, this means that if a graph can be efficiently raised to k, then maximal k-clique enumeration is reduced to a maximal clique problem. In practice the maximal k-clique problem is much harder than the maximal clique problem for the same graph[16]. The reason for this is that the running time of the clique algorithms heavily depends on the sparsity of the graph and other characteristics. G^k is usually much denser than the original G graph for any $k > 1$. The following sections in this chapter will analyse the characteristics of G^k as a function of G and k. Benchmark results will also be presented to support the claim that finding can k-cliques can indeed require orders of magnitude more time. Even for relatively small graphs ($n \sim 200$), finding the maximal cliques in G^2 can provide significant technical challenges.

5.3 Constructing the power graph

There are numerous ways to build G^k. A common approach is initiating a breadth-first search in every vertex and stopping at depth k, adding visited vertices to the adjacency list of the vertex. Another viable solution – if the graph is represented as an adjacency matrix – is to raise the adjacency matrix to the power of k using a matrix multiplication algorithm. None of these algorithms require exponential time, so any should be feasible, taking less time than the actual clique algorithm. In reality, constructing G^k can be surprisingly time consuming if the wrong algorithm or implementation is chosen. As it was stated in Chapter

3, five different algorithms were implemented for constructing the power graph. Since the power graph tends to be dense, these algorithms focus on BitSet adjacency matrix graphs. Memory usage turned out to be less of an issue as the bottleneck is the running time of clique algorithms for such graphs.

raiseBBS Straightforward breadth-first search implementation for BitSet adjacency matrix graphs.

raiseBAL Straightforward breadth-first search implementation for graphs using adjacency lists.

raiseMM This algorithm works on adjacency matrix representations, calculating G^k by raising the adjacency matrix to the power k.

raiseGMA and raiseGMB Alternatives proposed to the above algorithms, described in more details below.

The latter 2 algorithms use a helper method *multiply* which takes G_1 and G_2 graphs of the same size as parameter and produces another graph G – if v_i and v_j are neighbours in G_1 and v_j and v_k are neighbours in G_2, then v_i and v_k are neighbours in G. This means that $G^{k+l} = multiply(G^k, G^l)$. The algorithm is implemented for BitSet adjacency matrices.

Algorithm 4: Helper method for *raiseGMA* and *raiseGMB*

```
1 multiply(Graph G_1(V, E_1), Graph G_2(V, E_2))
2 begin
3     Graph G ← new Graph
4     for v_i ∈ V do
5         N_G(v_i) ← N_{G_1}(v_i)
6         for v_j ∈ N_{G_1}(v_i) do
7             N_G(v_i) ← N_G(v_i) ∪ N_{G_2}(v_j)
8         N_G(v_i) ← N_G(v_i) ∪ N_{G_2}(v_i)
```

Given the helper method, *raiseGMB* and *raiseGMA* produce the final result in two different ways. The former uses a simple **for** loop to call *multiply* k-1 times, while $raiseGMA$ attempts to save time by reducing the number of calls to *multiply*. It splits the power k to the sum of powers of 2, e.g. $11 = 8 + 2 + 1$. Calculating G^8 requires 3 calls to *multiply*; $G^2 = multiply(G, G)$, $G^4 = multiply(G^2, G^2)$ and $G^8 = multiply(G^4, G^4)$. The lower powers are calculated on the way to get G^8, therefore all is given to calculate G^{11}: $G^3 = multiply(G, G^2)$ and $G^{11} = multiply(G^3, G^8)$. The helper function is called 5 times in comparison to 10, required by $raiseGMB$.

The five algorithms gave the same results on all tested instances. For small graphs the correctness of the output can also be verified by visual inspection, however, the fact that they all result in the same output graph gives extra confidence of their correctness. Although the major motivation for five different implementations is to test the correctness of them, the running time of the algorithms is also interesting, which is analysed in the next section.

The algorithms were benchmarked on various random graphs of interest for the purposes of this project. All algorithms perform reasonably well for such data and for the purposes

of the project any would suffice. To stretch their performance, tests were also performed for large sparse graphs. This showed a clear

The algorithms were benchmarked on various random graphs and using a k values ranging from 2 to 16. Selected results are shown in Table 5.1.

Data			Algorithm				
k	n	p	raiseAL	raiseBBS	raiseMM	raiseGMA	raiseGMB
2	2500	0.01	0.672	0.012	0.610	0.012	**0.005**
		0.05	7.396	**0.010**	0.587	0.026	0.016
		0.1	15.204	**0.013**	0.556	0.062	0.032
	10000	0.005	18.029	**0.050**	28.265	0.160	0.126
		0.05	*	**0.052**	27.568	1.170	1.018
	25000	0.001	7.928	**0.219**	293.182	0.323	0.346
		0.01	*	**0.220**	*	2.934	2.680
5	2500	0.01	2.071	0.310	1.453	0.094	**0.015**
		0.05	7.649	0.301	1.518	0.245	**0.054**
		0.1	16.031	0.262	1.424	0.268	**0.110**
	10000	0.005	57.611	18.908	101.373	3.887	**0.485**
		0.05	*	17.272	97.983	12.379	**3.928**
	25000	0.001	*	90.297	*	7.479	**1.427**
		0.01	*	179.444	*	178.040	**10.306**

Table 5.1: Benchmark results for power graph algorithms (random data) measured in seconds. Best times in bold.

There is no clear winner among the algorithms that would perform the best on all graph types. However, *raiseGMB* consistently performs the best on critical graphs (large graphs, large k) and its results are not far from the best on all other inputs as well. On the other hand, *raiseBBS* performed very well on denser, smaller graphs. Interestingly, *raiseGMA* takes significantly more time to run than *raiseGMB* for large k's. It has a simple explanation: even though the number of calls to *multiply* is reduced, the running time of *multiply* depends on the sparsity of its first input graph. When using *raiseGMB*, this first parameter is always the input graph G itself, while for *raiseGMA* it is a power of G. These graphs are much denser then the original G, therefore *multiply* takes much longer time.

Due to its consistent performance, *raiseGMB* was used for all purposes to build G^k (it had decent runtimes for small dense graphs too). There may be more efficient algorithms to achieve the same, but it performs at a desired level. Any case where building the power graph takes more than a few seconds results in a graph of whose maximal cliques are technically infeasible to enumerate. In the following sections, the time to construct G^k is **not included** in the indicated runtimes.

5.4 Experimental results

5.4.1 The density of the power graph

Clearly, the power graph must be at least as dense as the plain graph. Usually, it is much denser. Experiments were run to measure the relationship between the density of the power graph as a function of the original graph's edge probability. In a random graph, generated by specifying its edge probability, the expected density equals the edge probability, because the expected number of edges is $\mu[|E|] = p * 1/2 * |V| * (|V| - 1)$, substituted into the definition of the density yields $\mu[D] = p$.

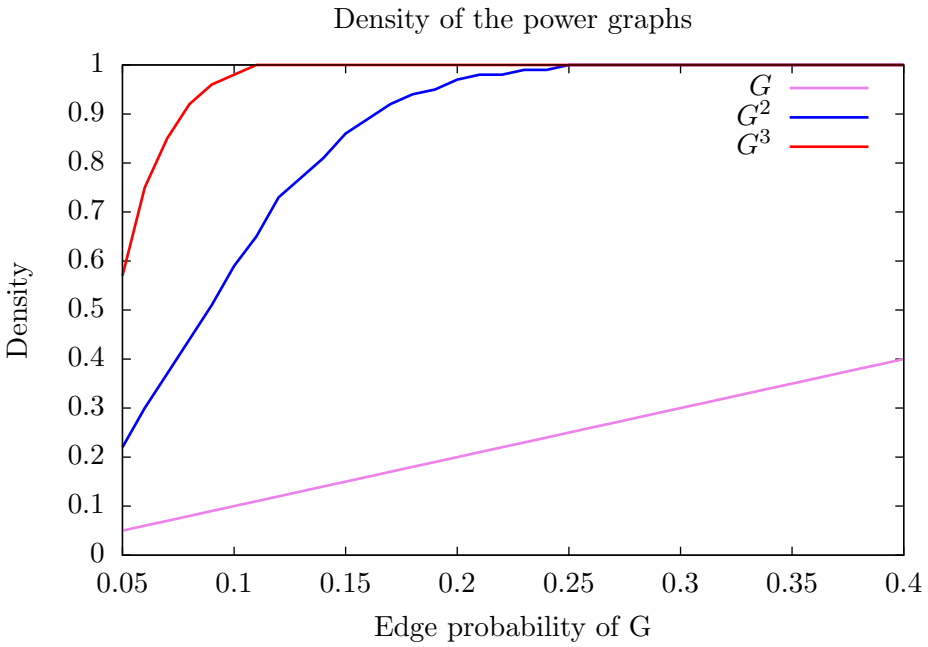

Fig. 5.1: Density of the power graph as a function of the original graph's density ($n = 80$).

Therefore, for the plain graph a linear relationship is expected between the edge probability. This, and the densities of G^2 and G^3 are shown in Figure 5.1. Of course, $D(G^3) \geq D(G^2)$ and $D(G^2) \geq D(G)$.

Generally, finding cliques in denser graphs are harder. Figure 5.1 supports the claim that the maximal k-clique problem is a much harder problem than the maximal clique problem for the same graph. However, there are exceptions to this. For example, in dense graphs, if $diam(G) = k$, then G^k will be a complete graph, which makes finding the maximal clique obvious and Figure 5.1 also suggests that the this happens for relatively low p and k values.

5.4.2 The turning point

Due to Figure 5.1 I became interested in the density of G for which G^k is a complete graph, as that makes k-clique enumeration trivial and pointless. In the first experiment, random graphs were tested. For this, a Java program called *TurningPoint* was coded,

which takes a *size*, a k value, a starting probability and a increment value as parameter. Starting at the starting probability, it produces a random graph with that probability and checks if G^k is complete. If not, the probability is incremented by the given value and the calculation is repeated. It is repeated until a complete graph is constructed, at which point the probability is stored.

This process is repeated a number of times, say 50 (the case of the actual experiment carried out for the project). The average of these runs gives an idea of the density for which G^k for a G of given size will have a diameter of k. By no means is this an accurate statistical measurement. For instance, doing the same, but starting from a high probability and decrementing it until a non-complete graph is found would give higher average results. Yet, it gives an idea of the trends.

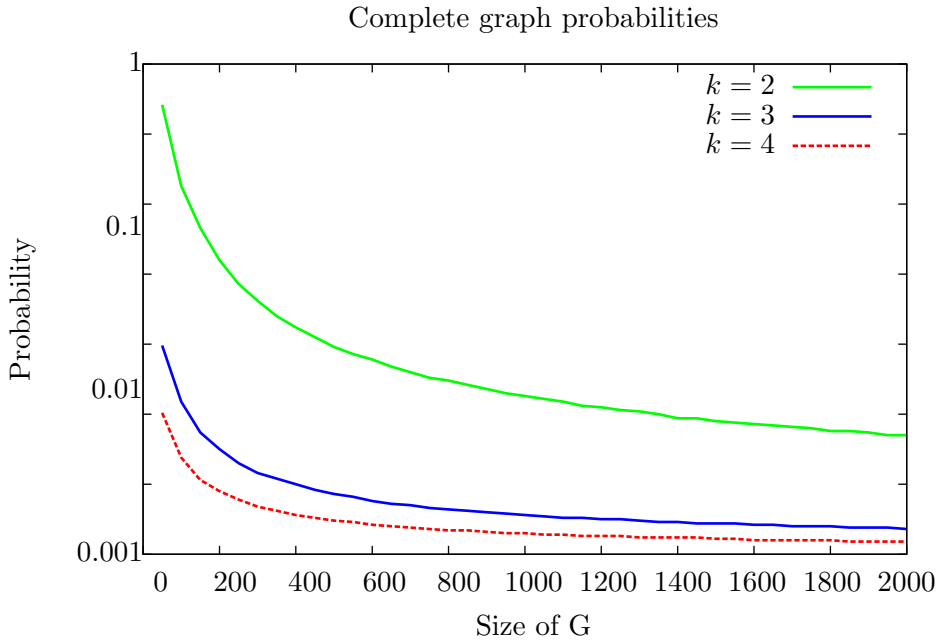

Fig. 5.2: A rough (under)estimation for the density of G for which G^k will be a complete graph.

Figure 5.2 shows the results for $k = 1, 2, 3$ for sizes 50 to 2000. This shows that the larger the graph, the less density it takes for the power graph to be complete. For example, the values for $n = 2000$ are as low as 0.085 (k=2), 0.018 (k=3) and 0.009 (k=4). The results lead to two observations: for a wide range of graphs and values of k, the k-clique problem is actually trivial and easier than the corresponding maximal clique problem and for many graphs the problem only makes sense for relatively low values of k.

Motivation is typically real life applications. Does the above observation hold true for real life datasets? Tests were run to calculate the diameter of real life data (Table 5.2 and Table 5.3).

It is quite apparent, that the real world graphs are sparse and their diameters tend to be larger than what was observed in the case of random graphs. Therefore, finding k-cliques in this graph generally makes sense for a range of k values. For all instances the diameter is larger than 3, which is the largest k this project deals with. Please note, that my algorithm

Data	Density	Diameter
marknewman-internet	0.000	>9
marknewman-astro	0.001	>9
marknewman-lesmis	0.087	5
marknewman-adjnoun	0.068	5
marknewman-power	0.001	>9
marknewman-polblogs	0.015	>9
marknewman-karate	0.139	5
marknewman-polbooks	0.081	7
marknewman-celegens	0.049	5
marknewman-netscience	0.002	>9
marknewman-football	0.094	4
marknewman-dolphins	0.084	8
marknewman-condmat	0.000	>9

Table 5.2: Density and diameter of Mark Newman graphs.

Data	Density	Diameter
biogrid-worm	0.001	>9
biogrid-yeast	0.009	5
biogrid-human	0.001	>9
biogrid-fission-yeast	0.006	>9
biogrid-plant	0.002	>9
biogrid-mouse	0.002	>9
biogrid-fruitfly	0.001	>9

Table 5.3: Density and diameter of BioGrid graphs.

did not check if the graphs are connected in the first place. Probably, some of the real life graphs are not connected, therefore the diameter is infinity.

5.4.3 K-cliques in practice

Arguably, benchmark results of real data is more interesting than k-clique enumeration for random graphs. Social networks, for example, are a natural application area, where it is easy to interpret the structures enumerated by 2-clique and 3-clique algorithms. It would show all people who have an acquaintance in common, or know someone who knows someone who knows the other person. But is it technically feasible to find such structures? Can these cliques be enumerated reasonably fast for the real world datasets?

Data	Cliques	Time	Calls
biogrid-worm	189,353	5.551	1,011,623
biogrid-yeast	*	*	*
biogrid-human	*	*	*
biogrid-fission-yeast	*	*	*
biogrid-plant	1,018	0.079	7,426
biogrid-mouse	906	0.092	5,658
biogrid-fruitfly	167,888,720	2,298.622	530,375,379
marknewman-internet	*	*	*
marknewman-astro	*	*	*
marknewman-lesmis	29	0.017	276
marknewman-adjnoun	1,679	0.090	5,904
marknewman-power	3,942	0.279	20,466
marknewman-polblogs	*	*	*
marknewman-karate	12	0.005	92
marknewman-polbooks	275	0.051	1,393
marknewman-celegens	13,341,191	49.693	60,354,157
marknewman-netscience	519	0.066	3,472
marknewman-football	5,435	0.149	20,083
marknewman-dolphins	198	0.024	717
marknewman-condmat	*	*	*

Table 5.4: K-clique enumeration for the Mark Newman and BioGRID datasets ($k = 2$).

The short answer is no. While most of the Mark Newman and BioGRID graphs took only a couple of seconds at most to enumerate maximal cliques, k-clique enumeration took too long for several graphs. The program was stopped if it had not finished within an hour. Table 5.4 shows the overall results of the experiment. The cases where the algorithm did not finish are marked by an asterix. Many of the harder graphs did not finish, and this

is only for 2-cliques, 3-cliques were not even attempted. Larger k values are even harder. As a result, I concluded that k-clique enumeration is too hard a problem for a lot of real applications.

5.4.4 The hardness of the k-clique problem

As it was seen with real data, the k-clique problem is much harder than the clique problem. The same is true for random graphs. When enumerating maximal cliques takes only a couple of milliseconds, the corresponding 2-clique and 3-clique problem may take minutes or even hours. Figure 5.3 shows, for instance, that for a graph with 80 vertices and an edge probability of 0.08, finding maximal cliques takes less than 10 milliseconds, whereas finding 3-cliques takes almost a hundred times longer. However, what I was interested in is does its hardness depend solely on the density? Or do power graphs possess other characteristics that make it an easier or even harder problem than what their density would indicate?

To find the answer for the questions, a number of random power graphs were generated. Then, for every power graph an *equivalent graph* was generated. Equivalency simply means that they have the same number of vertices and edges. As a result, we have a number of power graphs and the same amount of equivalent graphs which have the exact same density. The purpose of the experiment is to see which set of graphs is harder to enumerate. Is it the random set or power graph set? The following experiments provide the answer. For all runs, the BitSet algorithm was used with no ordering and Tomita pivot selection. At each setting, 50 random graphs were generated, for which all power graphs were constructed and for each power graph, an equivalent random graph was generated.

The first experiment used random $G(V, E)$ graphs, where $|V| = 80$. For this size, the full range of edge probabilities could be covered. However, from $p = 0.4$ onwards, every power graph became a complete graph, so the $p > 0.4$ range was excluded from the experiment. Figure 5.3 clearly shows that on the most critical settings (that is, around $p = 0.2$ for G^2 and $p = 0.08$ for G^3), power graphs are significantly easier to enumerate than random graphs of the same density. Please note that the y axis uses a log scale: runtimes for the power graphs are about a hundred times better.

Figures 5.4 and 5.5 show the number of nodes (recursive calls) it took the algorithm to find all maximal cliques and the number of cliques found. They roughly follow the same shape as the runtimes. Apparently, there is a strong correlation between the three values (as it could be expected).

While based on these graphs, it would seem that finding maximal k-cliques is easier than what the density of the power graph would indicate, this is not entirely true. For one, what we really see from this experiment is the hardness of the hardest case for small graphs: when the power graph is very dense but not complete yet. However, the very hard cases for even slightly larger graphs are infeasible to enumerate.

This is often not an issue, as real world data is often sparse. Using this setting, it cannot be told what happens for low probabilities in terms of runtimes; the differences are very minor and it takes very little time so the measurement is possibly inaccurate too. Nonetheless, it seems that at some point, the lines cross each other, and for low probabilities, the maximal

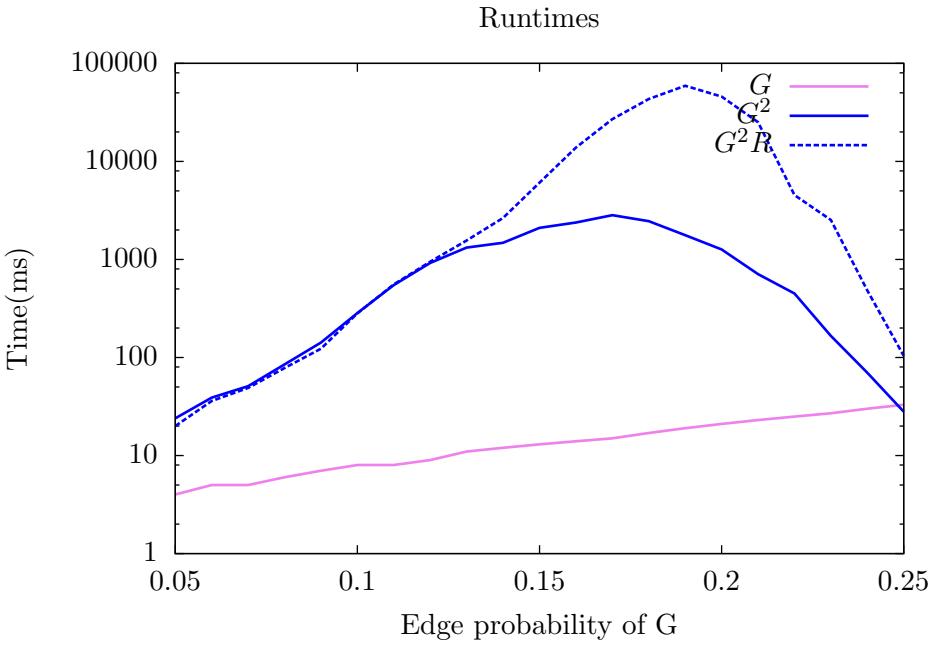

Fig. 5.3: Runtimes of enumeration for graphs of size 80.

k-clique problem is actually harder than enumeration in the equivalent graph. To investigate this further, experiments were run using other settings:

- $n = 500$, probabilities ranging from $p = 0.001$ to $p = 0.01$, $k = \{2, 3\}$
- $n = 1000$, probabilities ranging from $p = 0.005$ to $p = 0.02$, $k = 2$

Unfortunately, a wider range of probabilities proved to be infeasible to enumerate. The experiment backs up the hypothesis formulated in the experiment for dense graphs. For sparse graphs the k-clique problem is not easier than the corresponding random graphs at all, rather, it is harder at a number of settings. For example, where $n = 500$ and $p = 0.01$, it took less than 1,000 seconds to find cliques in the equivalent graph, while finding cliques in G^3 took nearly 10,000 seconds (Figure 5.6). For $k = 2$ the differences are less apparent (Figure 5.7), but the power graph is steadily harder.

This suggests that for many important applications, the k-clique problem is even harder than a maximal clique problem of a graph with the same density as the power graph.

5.5 Summary

It was shown that the maximal k-clique problem can be solved using a standard maximal clique algorithm, if we feed the power graph to it as input. It is possible to construct the power graph with relative ease, although, doing this efficiently was not a major focus of the project. Because of this, the time to build the power graph was not included in the results of the experiments.

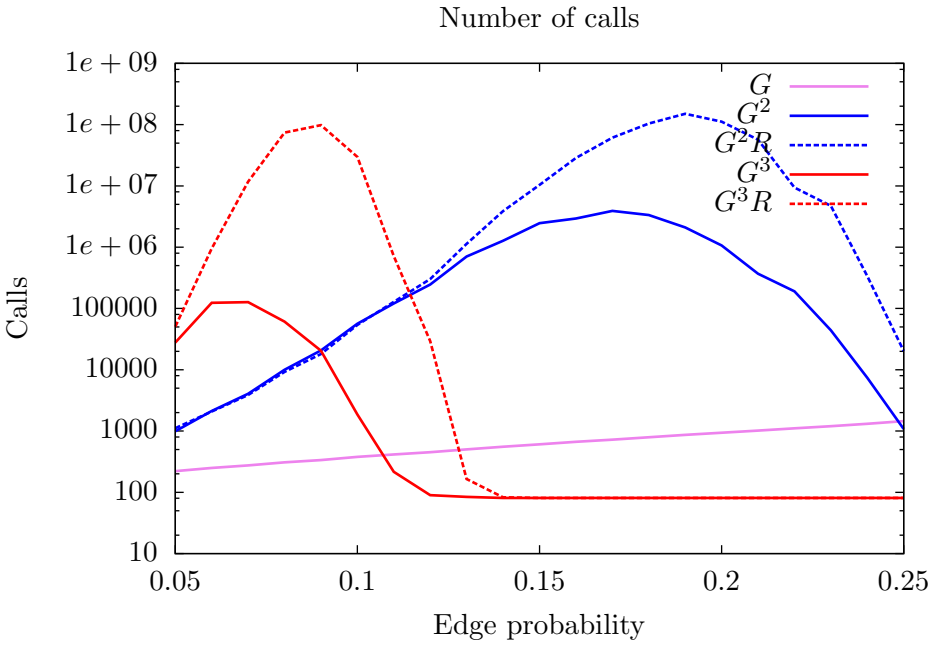

Fig. 5.4: Number of calls it took to enumerate all maximal cliques in the power graphs and their equivalent graphs. (n = 80)

The power graph turned out to be much denser than the original graph. This makes the problem at least as hard, and typically much harder, than the classic maximal clique problem. The exception to this is when the power graph is a complete graph, which happens when $k \geq diam(G)$. It is not unusual for dense random graphs to have a relatively small diameter. In real life data, the diameter was typically larger, and the k-clique problem made sense for $k = 2$ and $k = 3$ in all observed graphs.

Even though the problem is solved by constructing the power graph and enumerating maximal cliques in them, it is not clearly a feasible solution in practice. For the experiments, only the Mark Newman and the BioGRID dataset were used, which contain relatively small graphs. Processing data from the SNAP dataset was not even attempted. Results were only obtained for $k = 2$. Even with these strong restrictions, some runs took an extremely long time.

In the literature, no faster alternative was found to the approach used in this project. As it is now, maximal k-clique enumeration is a hard problem, even too hard for a number of practical applications. It would be worth researching if different viable approach existed, a special purpose algorithm that suits k-clique enumeration more.

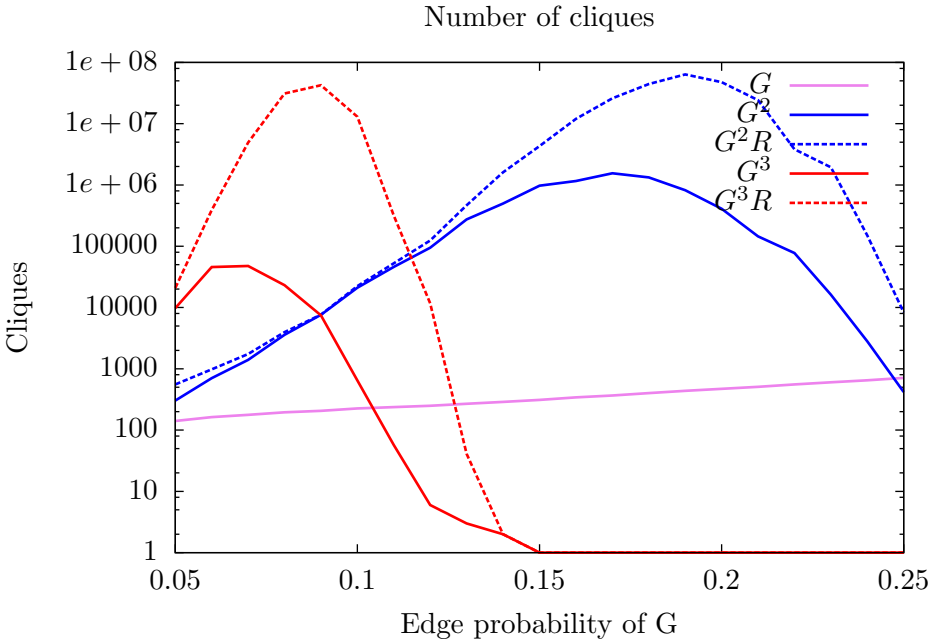

Fig. 5.5: Number of cliques in the power graphs and their equivalent graphs. (n = 80)

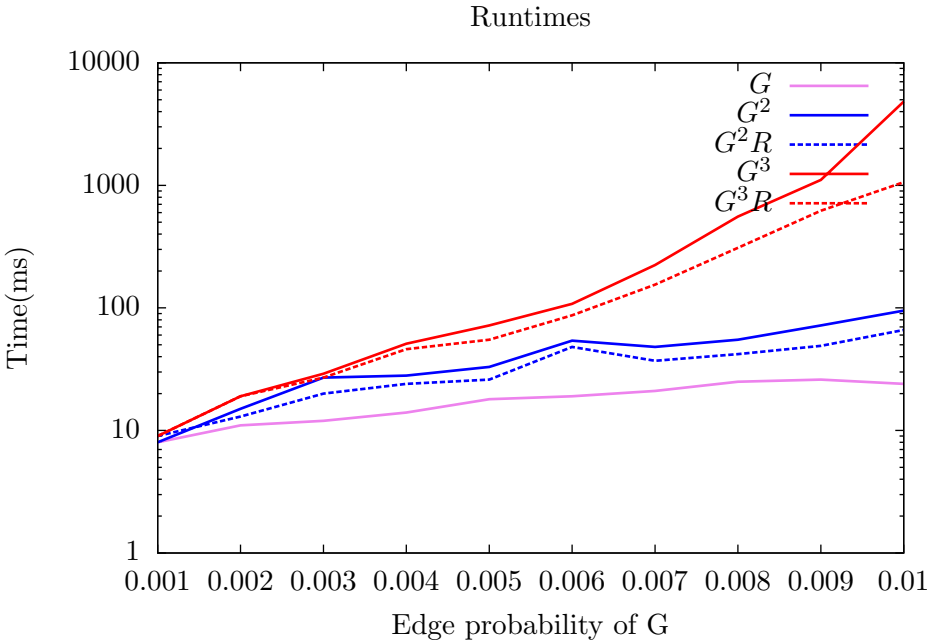

Fig. 5.6: Runtimes of the k-clique problem and equivalent graphs problem for graphs of size 500.

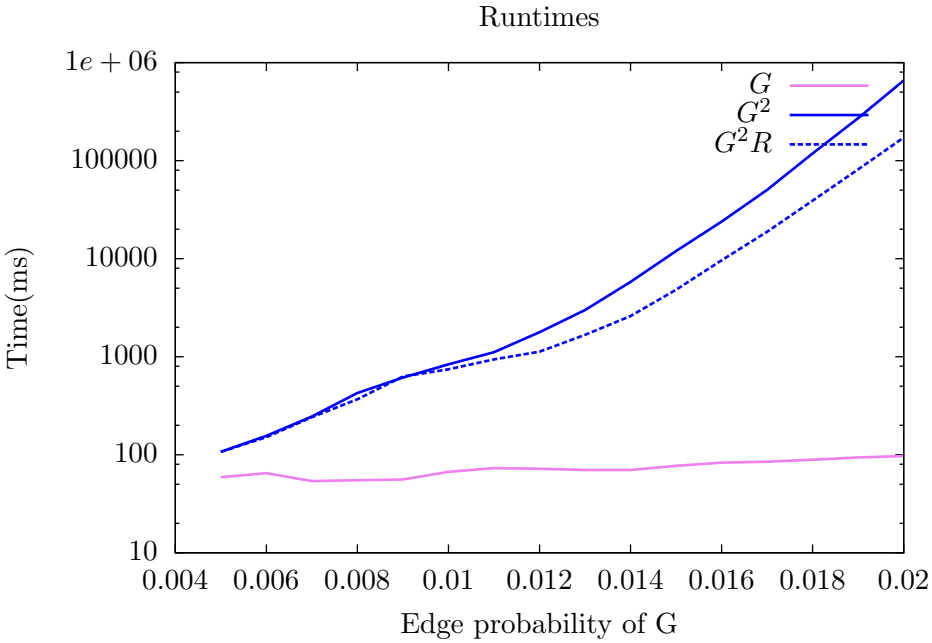

Fig. 5.7: Runtimes of the k-clique problem and equivalent graphs problem for graphs of size 1000.

Chapter 6

Labelled cliques

6.1 A maximal labelled clique algorithm

6.1.1 The problem

Carrabs et al. introduced a variant of the maximum clique problem for labelled graphs, $G = (V, E, L)$ [7]. The set of possible labels is denoted by L. They defined a function $l : E \to L$ that represents the label assignments to the edges, and $\lambda : V \to L$ that returns all the different labels of the edges which connect vertices in a $V' \subseteq V$ subset of vertices given as input. A constraint is introduced which limits the number of different labels that are allowed in the cliques. This number is the budget b. For budgets $b \geq |L|$, the problem does not differ from the basic maximum clique problem, therefore, the interesting cases are $|L| > b > 0$. The cliques that satisfy this constraint are named *feasible cliques*. The problem is finding the maximum feasible cliques. If more than one such maximum cliques exist, the one with a smaller cost is preferred, that is, the one using less different labels.

Analogously, the maximal labelled clique problem can be formulated: we wish to enumerate all maximal labelled cliques that are within a budget. A maximal labelled clique $C \subseteq V$ is a feasible clique (within the budget) that cannot be enlarged either because it is a maximal clique (there is no vertex that is adjacent to all $v \in C$) or there exist vertices that could be added to it, but doing so would result in an infeasible clique. To my knowledge, no algorithm has been proposed for this problem.

6.1.2 The algorithm

P. Prosser and C. McCreesh devised an alternative [15] to the mathematical programming model proposed by Carrabs et al. Their algorithm is similar to other clique algorithms in the sense that it is a branch and bound, recursive algorithm. However, it also takes into account that the clique has to be feasible by keeping track of the labels used in the growing clique in a variable. When a vertex is added to the growing clique, a new label set is calculated, and if its size is over the budget, the branching is stopped. Inspired by this, I adapted the Bron-Kerbosch algorithm to suit the maximal labelled clique problem. The

core idea is that at each recursive call, when P and X are recalculated, we also consider the budget. For every vertex in P and X, at each call, we calculate what would be the size of the label set if the vertex was added to the growing clique. If this exceeds the budget, the vertex is removed from the set containing it. A maximal labelled clique is found, when P and X are empty. Hereafter, I call this algorithm BKL.

Statement 2. *Algorithm BKL finds all and only the maximal labelled cliques in a labelled graph.*

Proof. At each recursive call, when a new vertex v is added to the growing clique, P and X are recreated. First, the non-neighbours of v are removed from P and X. This means that the vertices that remain in them are all adjacent to C, the growing clique, therefore, if they are added to C, the result is a clique as well. Then, for all $v \in P \cup X$, $c(v) = |\lambda(C \cup v)|$ is calculated. This is the cost of the clique, if v was added to C. If $c(v) > b$, v is removed from P or X. This ensures, that all remaining vertices, if added to C, would result not only in a clique, but a feasible clique too.

Clearly, the algorithm branches into feasible cliques, as every vertex that is in the candidate set is adjacent to all vertices in C, and adding it to C does not exceed the budget, by the definition of how P is maintained. We must only show that, when P and X become empty, C is a maximal labelled clique. Since P is empty, the clique cannot be further extended. However, if X is not empty at this point, that means that there exist a vertex, which has been visited before, that could be added to C and the result would be a larger feasible clique. On the other hand, if X is empty, C is a maximal labelled clique, as if it wasn't, then the vertex that could extend it would still be in either P or X. □

Algorithm 5: The BKL algorithm.

```
1  BKL(Set C, Set P, Set X, int b)
2  begin
3      if P ≠ ∅ then
4          for v ∈ P do
5              Set C' ← C ∪ v
6              Set P' ← P ∩ N(v)
7              Set X' ← X ∩ N(v)
8
9              for v ∈ P' do
10                 if c(v) > b then  P' ← P' \ {v}
11
12             BKL(C', P', X', b)
13
14             P ← P \ {v}
15             X ← X ∪ {v}
16     else
17         for v ∈ X do
18             if c(v) > b then  X ← X \ {v}
19         if X = ∅ then  report C as maximal labelled clique
```

Calculating $c(v) = |\lambda(C \cup v)|$ is a rather expensive operation to perform at each recursive call for all v in P and X. Fortunately, the NOT set does not have to be maintained continuously, as we only need to check its emptiness once there are no candidates left. I found that if we keep removing vertices that are not adjacent to the next candidate, but we only remove infeasible vertices once P is empty, the runtime of the algorithm is better in every case (and the algorithm still gives the same results). Sometimes the benefits are marginal, in other cases it can be significant, depending on the type of input. For the complete algorithm with this modification, see Algorithm 5, and Appendix B.2 for the Java implementation. Similarly to the algorithm of P. Prosser and C. McCreesh, my implementation passes another argument to the recursive function, the label set of the growing clique. This speeds up the calculation of $c(v)$, because the labels within C do not need to be calculated. Instead, we only need to add the labels of the edges joining the candidate and the vertices of C. The cardinality of the resulting label set equals $c(v)$.

6.2 The correctness of the algorithm

To ensure that the algorithm is indeed correct, and also that my code implements it correctly, it had to be tested. The difficulty was that there are no published results for the problem against which the output could have been easily verified. Yet, there were ways to confirm it. The first step was to run the algorithm on small graphs, for which the results can be calculated on paper. This is an error-prone and inefficient process but it worked well as a first filter. The second – still informal – test was to modify the algorithm to enumerate maximum labelled cliques. The maximum labelled clique problem is a subset of the maximal labelled clique problem. Once we have the maximal solutions, we can then discard the non-maximum ones, thus solving the maximum problem, although, it is a a much slower technique than using a dedicated algorithm. As I have written, P. Prosser and C. McCreesh have devised an efficient maximum labelled clique algorithm [15]. C. McCreesh kindly sent me the results for some of the data I had generated. My modified algorithm found the same maximum cliques as him.

The third and most formal test, was to solve the problem using only the classic Bron-Kerbosch algorithm whose correctness had been verified. I calculated all $b-$*combinations* of the labels, where b is the budget, and constructed unlabelled graphs for every combination from the labelled graph (that is, an unlabelled graph that has all the edges from the labelled graph which have a label that is in the combination). In the resulting graphs, classic maximal cliques can then be enumerated. However, this will find duplicates and even non-maximal labelled cliques, too. To find the correct number of maximal labelled cliques, all the results need to be merged, and then the duplicates, and sets of vertices that are a subset of another solution need to be discarded. This is a very inefficient way to solving the problem. The number of combinations is $\binom{|L|}{b}$, which grows at a factorial rate for fixed budgets as a function of the cardinality of the label set. Then, cleaning the results can be very slow, too. Nonetheless, it is still fast enough to enumerate graphs of a hundred vertices, if there are not many labels (i.e. it worked for $|L| = 5$). And it only uses tested algorithms. Again, the output of BLK matched the results produced this way.

6.3 The experiments

I could not find appropriate real life data of labelled graphs. Instead, labels were generated uniformly at random for unlabelled graphs to construct labelled graphs. There is a wide range of possible settings that could be used for the experiments, random or real life data, different number of labels and varying budgets. The focus of the project is to find out whether the maximal clique variants can feasibly be solved in practice, therefore, I narrowed down the data set to real life graphs. The number of labels was also set to 5 for all tests, which seemed a reasonable value for real life uses. The budget varied from 2 to 4.

Data	budget = 2			budget = 3		
	Cliques	Runtime	Nodes	Cliques	Runtime	Nodes
marknewman-internet	49,355	0.815	100,634	57,784	1.246	180,027
marknewman-astro	220,866	7.376	792,527	578,577	68.016	6,797,798
marknewman-lesmis	224	0.021	602	258	0.029	1,087
marknewman-adjnoun	334	0.015	687	354	0.016	843
marknewman-power	6,098	0.104	11,876	5,795	0.100	12,221
marknewman-polblogs	35,684	0.560	89,075	77,486	1.309	382,163
marknewman-karate	54	0.003	136	47	0.003	161
marknewman-polbooks	335	0.021	849	374	0.025	1,234
marknewman-celegens	1,850	0.056	4,215	2,287	0.072	6,508
marknewman-netscience	2,242	0.075	6,596	2,393	0.097	12,033
marknewman-football	496	0.030	1,180	543	0.036	1,854
marknewman-dolphins	113	0.006	273	109	0.006	328
marknewman-condmat	166,443	4.492	444,559	205,999	10.173	997,484

Table 6.1: Labelled Clique Enumeration results for the Mark Newman datasets.

Data	budget = 2			budget = 3			budget = 4		
	Cliques	Runtime	Nodes	Cliques	Runtime	Nodes	Cliques	Runtime	Nodes
biogrid-worm	6,190	0.097	11,187	6,267	0.093	12,678	5,954	0.103	13,623
biogrid-yeast	353,809	3.446	822,573	804,010	14.272	3,705,854	1,343,894	127.453	31,528,745
biogrid-human	27,986	0.328	50,186	28,599	0.421	63,359	26,097	0.522	75,711
biogrid-fission-yeast	20,121	0.277	43,722	34,099	0.543	116,189	40,595	0.869	274,785
biogrid-plant	2,903	0.061	6,264	3,099	0.075	8,602	2,799	0.099	11,374
biogrid-mouse	1,578	0.040	3,157	1,553	0.042	3,231	1,534	0.042	3,263
biogrid-fruitfly	23,212	0.245	34,872	22,954	0.249	37,962	22,363	0.262	39,109

Table 6.2: Labelled Clique Enumeration results for the BioGRID datasets.

Table 6.1 shows the results for the Mark Newman dataset. All graphs were could be enumerated within reasonable time. However, we can observe that the higher the budget, the harder the problem gets. While the algorithm finished reasonably quickly for $b = 2$, larger budgets caused issues, especially for $marknewman - astro$. This corresponds to the results we got for the classic clique enumeration problem in Chapter 4. In that case, on average, $marknewman - condmat$ was the hardest, **BSNN** struggled with enumerating $marknewman - astro$. This is probably due to the structure of the graph, which makes it much harder to enumerate if Tomita's pivot selection is not applied to it. Unfortunately, pivoting does not work with the labelled clique algorithm, as the observation that enabled it is no longer true for the labelled case. This may be the cause why the runtime is more for *astro* than for *condmat*. The results for the BioGRID dataset are very similar (Table 6.2. For $b = 2$, the runtimes are low, around the same as for the unlabelled problem. For larger budgets, the runtimes are higher, but they are still in a reasonable range.

In general, the lower the budget, the more restrictive the constraint is which makes the problem easier. The problem also becomes easier if there are more possible labels. If the cardinality of the label set would have been increased to, say, 10, then the runtimes for $b = 3$ and $b = 4$ would be much lower as well.

One could think that the number of cliques should decrease with the introduction of labels, as it is a restriction after all. While this is true in some cases, it is not the case in general.

The above tables show that the number of cliques is usually more for larger budgets. On the other hand, if we compare the results to those for the unlabelled graphs, we see that there are consistently less maximal cliques than labelled cliques in most graphs. Overall, the introduction of labels in fact increases the number of maximal cliques. A simple example of this is 3 vertices that are all connected. These form a maximal clique if the edges are unlabelled. If we assign three different labels to the edges, then, for $b = 2$ and $b = 3$, the number of maximal labelled cliques is 3, as all three vertices cannot be in a single clique due to the budget.

Extremes of this phenomenon can be observed in the results I got for the test data used by Tomita et al. [21] (DIMACS graphs and Moon-Moser graphs). There are 7 and 8 maximal cliques in $c-fat-200$ and $c-fat-500$ and these can be enumerated in roughly a hundredth second. However, the labelled clique problem is much harder – for a $b = 3$, in $c-fat-500$, there exist over 18 million maximal labelled cliques on average and the algorithm took well over an hour to find these (Table 6.3 and 6.4. Similar

Data	budget = 2			budget = 3		
	Cliques	Runtime	Nodes	Cliques	Runtime	Nodes
dimacs-brock200-2	46,974	0.708	133,543	164,493	1.674	674,624
dimacs-c-fat200-5	47,692	1.447	243,132	262,004	21.096	4,628,462
dimacs-c-fat500-10	933,360	55.728	6,463,984	18,054,637	4949.231	5,978,614
dimacs-johnson8-4-4	6,538	0.218	21,512	22,602	0.707	128,433
dimacs-johnson16-2-4	35,892	0.801	130,254	208,502	3.547	1,181,333
dimacs-hamming6-2	7,838	0.320	34,411	35,346	1.719	406,541
dimacs-hamming6-4	701	0.035	1,279	903	0.037	1,822
dimacs-keller4	66,038	1.094	223,963	353,045	4.542	1,844,077
dimacs-MANN-a9	2,801	0.140	11,183	9,249	0.633	92,497
dimacs-p-hat300-1	30,906	0.394	60,629	56,075	0.673	161,656
dimacs-p-hat300-2	216,432	4.046	861,849	1,495,837	44.389	12,904,256
m-m-30	865	0.078	2,901	1,977	0.159	14,804
m-m-45	3,003	0.159	12,605	10,533	0.745	118,530
m-m-48	3,689	0.185	16,035	13,995	0.931	167,318
m-m-51	4,505	0.218	20,178	18,349	1.203	234,293

Table 6.3: Labelled Clique Enumeration results for Moon-Moser and DIMACS graphs.

Data	n	m	Cliques	Runtime	Nodes
dimacs-brock200-2	200	9,876	431,586	1.028	1,103,895
dimacs-c-fat200-5	200	8,473	7	0.011	588
dimacs-c-fat500-10	500	46,627	8	0.020	1,548
dimacs-hamming6-2	64	1,824	1,281,402	1.674	3,094,420
dimacs-hamming6-4	64	704	464	0.011	1,362
dimacs-johnson8-4-4	70	1,855	2,027,025	5.927	14,158,473
dimacs-johnson16-2-4	120	5,460	114,690	0.281	273,062
dimacs-keller4	171	9,435	10,284,321	6.205	14,319,049
dimacs-MANN-a9	45	918	590,887	0.530	964,208
dimacs-p-hat300-1	300	10,933	58,176	0.192	133,744
dimacs-p-hat300-2	300	21,928	79,917,408	122.328	196,461,994
m-m-30	30	405	59,049	0.154	88,573
m-m-45	45	945	14,348,907	6.432	21,523,360
m-m-48	48	1,080	43,046,721	24.458	64,570,081
m-m-51	51	1,224	129,140,163	64.632	193,710,244

Table 6.4: Labelled Clique Enumeration results for Moon-Moser and DIMACS graphs.

This is not true for other data, though. Interestingly, in most of the DIMACS graphs, and especially in the Moon-Moser graphs (data starting with m-m), the opposite can be observed. There are less maximal labelled cliques than simple maximal cliques and the algorithm run much faster in the labelled case. For instance, in $m - m - 51$, there are around 129 million maximal cliques. With a budget of 2, there are only 4,505 cliques and the runtime also drops from over a minute to 0.2 seconds. Great difference can be seen in the case of many other graphs, e.g. $m - m - 45$, $m - m - 48$, $p - hat - 300 - 2$ and $johnson - 8 - 4 - 4$. Typically, the assignment of labels seems to have a greater impact on these graphs than on the Mark Newman and BioGRID graphs. I suspect this is due to their density and special characteristics that make them hard for classic clique enumeration.

In conclusion, it remains unclear whether the labelled clique problem is harder or easier than the classic maximal clique problem. It depends on the input – in some cases it has very different characteristics, but depending on the graph, this can mean that it is much easier or that it is much harder a problem. Admittedly, my maximal labelled clique algorithm has shortcomings. For example, pivot selection greatly improves the original Bron-Kerbosch algorithm, and it is not implemented for the labelled algorithm. Also, there may exist more efficient approaches to implement the algorithm with regards to data structures and how the cost of adding a vertex to the growing clique is calculated.

Chapter 7

Conclusion

The aim of the project was to find out the extent to which maximal k-cliques and maximal labelled cliques are feasible to be enumerated by classic maximal clique algorithms. While there is no definite answer to this question, the project has been successful in producing valuable results.

Overall, k-clique enumeration was found to be a hard problem. With the implemented Bron-Kerbosch algorithm, many of the real life data sets proved to be too hard. Larger graphs, such as SNAP graphs, were not even attempted due to memory limitations. I assume that they would have turned out to be even harder. The hardness of the problem comes from the technique: we enumerate maximal cliques in the power graph using general purpose maximal clique algorithms. The power graph is very dense, which makes the problem much harder than the original problem. For instance, for random graphs with $n = 80$ and $p = 0.05$ the density of G^3 is as high as 60% on average. Power graphs built from random graphs cannot be considered random, though. Therefore, enumerating maximal cliques in power graphs is not the same as doing so in randomly generated graphs with the same density. I analysed whether power graphs or equivalent graphs are the harder. For small graphs, in cases where the power graph got very dense, finding maximal cliques in the power graph was easier. Unfortunately, these results are less interesting, as real life graphs tend to be larger and sparser. Experiments were also run using larger ($n = 500$ and $n = 1000$), sparser graphs. It was shown that with these settings, the power graphs are equally hard (even harder) as the equivalent graphs.

Although labelled cliques had been defined due to Carrabs et al. [7], the maximal labelled clique problem was formulated in this project. An adaptation of the Bron-Kerbosch algorithm was proposed to enumerate maximal clique. This method was shown to be viable for all used real life graphs. However, for some graphs, with the settings I used, the labelled clique variant was still harder than the unlabelled problem. This is highly setting dependent. Ultimately, the hardness of the labelled clique problem depends on the quotient of the cardinality of the label set and the budget. Using five labels and $b = 2$ was usually enough restriction for the problem to become easier than the unlabelled one. Whether introducing labels increases or decreases the hardness of the problem is also much input dependent. This could be seen in the results for Moon-Moser and DIMACS instances. The same setting resulted in a great increase in the number of cliques and runtimes in some graphs, while in some other graphs it reduced them.

In general, the labelled clique problem is much harder than the k-clique problem, and is generally feasible for all input for which the classic maximal clique algorithms are feasible. It is somewhat disappointing that it is not clearly easier, as it is an additional constraint. It is not only harder in some cases with regards to runtimes, which could be an effect of the extra operations, but often the number of recursive calls is greater for the labelled problem.

The experimental results are not the only achievements of the project. The code contains several different implementations of the Bron-Kerbosch algorithm. Altogether, the code makes up a library for whoever needs a tested implementation in Java. The BitSet implementation is both faster than a straightforward implementation and more efficient memory-wise, than, for example, using a 2D boolean array for the adjacency matrix.

The project has also some shortcomings and it could be improved and extended in a number of ways. The labelled clique algorithm could possibly be further optimised. As it is now, it does not implement any pivoting, from which the Bron-Kerbosch algorithm has been shown to greatly benefit. I feel that given enough time, it could be implemented in a more efficient way. It would be worthwhile to try the algorithm's speed using other data structures, like the one used in the original Bron-Kerbosch article – maybe these would allow to maintain the candidate and the NOT set with less effort. Also, the vertices could be initially ordered based on the labels of their edges. Unfortunately, these modifications were out of scope for this project.

A possible extension of the project would be to implement code for the maximal common subgraphs problem. Similarly to k-cliques, it is a problem that can be solved by an adapted maximal clique algorithm, by transforming to input (instead of power graphs, here we need to produce the product graph from the input, which is 2 undirected graph). The problem has practical uses with applications in theoretical biology, for example [12]. It was planned for this project, but was never implemented.

Regardless of the above shortcomings, the main goals of the project were to gain insight into the hardness of the clique variants and providing a library to whomever needs a solution in Java. These objectives have been fulfilled.

Appendix A

How experiments are conducted

A.1 Overview

The core library has all the reusable classes – data structures, algorithms, often used utility methods – that are needed throughout the project. It also has several executables which serve as an entry point to obtaining certain results. One could go about doing the experiments by calling these methods, storing and analysing them manually. However, some experiments take weeks to execute, process thousands of graphs and result in thousands of result files. To automatise the process, for each experiment type, several additional Java classes were implemented. The exact process varies from experiment to experiment, but they all share certain characteristics. The Java applications of the experiments do not do any of the real work, they just print out a list of calls to applications in the core library. These calls can be printed to a text file, which can be made executable to run them. I call such a batch of Java calls a job file.

This method of conducting the experiments have several benefits. The major objective is to make the results repeatable and traceable. This way, this objective is satisfied, because even though the experiments themselves are automatic, they consist of individual Java calls. For example, when we need the average runtime of an algorithm over 50 random graphs, an experiment can do that automatically. However, it also saves each generated graph and results for each graph to text files. Therefore, not only the overall result is saved, but individual results too; and if any of the results is suspicious, it can be recalculated by running the algorithm just for that graph again. Other benefits include the separation of the core functionality from the experiment code. That is, the experiment code does not use any of the classes from the core library, it just prints calls to the job files. This is not even Java specific, the same job files could be written in any language or even manually.

The main steps of the experiments are also separated. For instance, generating the data and running the algorithms are separate jobs. Analysing the results is another. An example when this is beneficial is when we want different statistics for the same results. Let us assume that I calculated the average, the maximum and the minimum of runtimes over a set of data. Eventually I may wish to find the median as well. The experiment needs not be redone wholly, the individual results are available. Only a new analyser job needs to be coded that also calculates the median.

The following sections give a quick overview how three of the important experiments work to give a general idea how the experiments were conducted.

A.2 Maximal clique experiments on real life data

Every experiment has a separate folder that contains the executables and the folders used in the experiment. For simple maximal clique experiments, this folder is called $experiment_clique$. Since the same experiment had to be run on different data and settings, the folder was first copied and given a custom name, for instance $experiment_clique_marknewman$, when the clique experiment was run on the Mark Newman dataset.

The first step is to put the graphs to be enumerated in the *data* folder. Second, the MakeExperiment Java application needs to be run. This takes a single argument, the number of times the algorithms are to be run on each graph. The runtime is the average of these runs. The application prints out all the Java calls to the core library that need to be run. Executing the job file puts the results in the *results* folder. This experiment enumerates cliques using 6 different algorithms. The results are save in text files in a way that every result can be easily found. For instance, the result file called $marknewman - karate_epps$ contains the results for *karate* from the Mark Newman dataset produced by Eppstein's algorithm.

Once the results are available, they need to be aggregated. The experiment offers a Java application called *LatexTable* which constructs and prints a Latex table from the results. This way, I could simply print it to a .tex file and include it directly in the dissertation.

A.3 K-clique experiments for random graphs

For the experiments that use random graphs, numerous graphs need to be generated at different settings (for instance various size and edge probability). The *RandomGraph* class in the core library provides means to generate a single random graph taking two arguments: the size and the edge probability. The *MakeGenerator* class in the experiment folder takes parameters such as the starting edge probability, ending edge probability, the step size between them, the number of vertices and the number of graphs to be constructed at each setting. It prints out the calls to *RandomGraph* which produce the graphs specified in the arguments. The printed output can be directed to a text file, which can in turn be made executable by chmod. The graphs are saved in a folder of a specified name. Each experiment folder contains a *data* folder in which this folder is placed. The graphs are saved under names which use a specific format: $nnnn - ppp - cccc$. The first block shows the number of vertices in the graph, the second block shows the edge probability (per thousand) and the last block is the number of the graph generated at the setting: e.g. $0100-015-003$ stores the third graph generated at the setting $n = 100$, $p = 1.5\%$. This allows to identify the graphs stored in the files.

Once the graphs have been generated, their corresponding power graphs need to be constructed. The program *MakeRaise* creates a list of calls to *Raise* – which builds G^k for a given G and k – to build the desired power graphs for all graphs in a given folder. These

resulting graphs are stored in another folder by the name of the folder which contains the input files with ˆk appended to it. For instance, if the original graphs are stored in folder */data/GRAPH/* and $k = 2$ then the output graphs will be stored *GRAPHˆ2*. Each power graph is stored under the same file name as the original graph, therefore, it is very easy to look it up: if G is stored in *data/GRAPH/0100-015-003* then *data/GRAPHˆ2/0100-015-003* contains G^2. In the experiment this data is generated for, I was interested in whether the hardness of k-clique enumeration comes solely from the density of the power graphs, or the power graphs possess unique characteristics that make it a harder or easier than a random graph with the same density. For this, equivalent graphs are generated for the power graphs. An equivalent graph is a graph that has equal number of vertices and edges, but is randomly generated. These graphs are stored in a separate folder under with the same name as the power graph folder with R appended to it. For instance, the equivalent graphs for the graphs stored in *data/GRAPHˆ2/* are stored in *data/GRAPHˆ2R/*.

Obtaining the results is nearly identical to how it was done with regular cliques, but in this case only one algorithm was used. However, instead of building a Latex table from the results, in this case I wanted to plot the results. The code that aggregates the results does it so that the format of the output is directly usable by *gnuplot*, an application used in the project to plot graphs of the results.

A.4 Other experiments

Other experiments are either carried out in a very similar manner, or they even reuse steps from other experiments. Take enumerating k-cliques in the Mark Newman graphs for example. The k-clique experiment for random graphs provides a step that constructs the power graphs for all graphs in a specified folder. We can put the Mark Newman graphs in a folder and perform this step to build the power graphs. Then by moving these graphs to the data folder of the plain clique experiment, we can enumerate cliques in the power graphs and by doing so, enumerate k-cliques in the original graphs.

Appendix B

Important code snippets

B.1 BitSet Clique Algorithm

```
package algorithm.bitset_impl;

import algorithm.CliqueAlgorithm;
import graph.GraphBitSet;

import java.util.BitSet;

/**
 * BitSet implementation of the Bron-Kerbosch algorithm with pivot selection and
     (optionally) initial vertex ordering.
 */
public class AlgorithmBS extends CliqueAlgorithm {

        private final GraphBitSet graph;
        private final int size;
        private Order order;
        private Pivot selector;

        /**
         *
         * @param graph a BitSet graph for which maximal cliques are to be
             enumerated
         * @param order vertex ordering for outer loop
         * @param selector    pivot selector
         */
        public AlgorithmBS(GraphBitSet graph, Order order, Pivot selector) {
                this(graph, order, selector, false);
        }

        /**
         *
         * @param graph a BitSet graph for which maximal cliques are to be
             enumerated
         * @param order vertex ordering for outer loop
         * @param selector pivot selector
```

```
 * @param verbose if set to true, every maximal clique is printed to
     stdout (used mainly for testing)
 */
public AlgorithmBS(GraphBitSet graph, Order order, Pivot selector, boolean
    verbose) {
        super(verbose);
        this.graph = graph;
        this.size = graph.size();
        this.order = order;
        this.selector = selector;
}

/**
 * Runs the algorithm.
 */
@Override
public void run() {
        int[] order = this.order.order(graph);

        BitSet P = new BitSet(size);
        P.set(0, size);

        BitSet R = new BitSet(size);
        BitSet X = new BitSet(size);

        if (order == null) {
                extend(R, P, X);
                return;
        }

        for(int v : order) {
                // Add v to R
                BitSet newR = (BitSet)R.clone();
                newR.set(v);

                // intersection of P and the neighbourhood of v
                BitSet newP = (BitSet)P.clone();
                newP.and(graph.neighbours(v));

                // intersection of X and the neighbourhood of v
                BitSet newX = (BitSet)X.clone();
                newX.and(graph.neighbours(v));

                // call recursion (Tomita's recursive call, implemented in
                    its superclass)
                extend(newR, newP, newX);

                // remove v from candidates
                P.flip(v);

                // add v to X
                X.set(v);
        }
}
```

```
        private void extend(BitSet R, BitSet P, BitSet X) {
                inc();
                // If the candidate set is not empty, continue the algorithm
                if (P.nextSetBit(0) > -1) {

                        // Tomita pivot selection: maximise the size of the
                            intersection of P and N(pivot)
                        int pivot = selector.selectPivot(graph, P, X);

                        // Create P - N(pivot)
                        BitSet S = (BitSet) P.clone();
                        S.andNot(graph.neighbours(pivot));

                        for (int v = S.nextSetBit(0); v > -1; v = S.nextSetBit(v +
                            1)) {
                                // Add v to R
                                BitSet newR = (BitSet) R.clone();
                                newR.set(v);

                                // intersection of P and the neighbourhood of v
                                BitSet newP = (BitSet) P.clone();
                                newP.and(graph.neighbours(v));

                                // intersection of X and the neighbourhood of v
                                BitSet newX = (BitSet) X.clone();
                                newX.and(graph.neighbours(v));

                                // call recursion
                                extend(newR, newP, newX);

                                // remove v from candidates
                                P.flip(v);

                                // add v to X
                                X.set(v);
                        }
                }
                // If maximal clique is found, print it and return
                else if (X.nextSetBit(0) == -1) {
                        reportClique(R);
                }
        }
}
```

B.2 Labelled Algorithm

```
package labelled;

import algorithm.CliqueAlgorithm;

import java.util.BitSet;

/**
 * Labelled clique algorithm final version. Takes a labelled graph and a budget
    as arguments in the constructor.
 * Should be started by calling {@link algorithm.CliqueAlgorithm#execute()}
    instead of calling run directly.
 */
public class BKL_PostX extends CliqueAlgorithm {

	LabelledGraph graph;
	int budget;

	/**
	 * Constructor.
	 * @param graph the input graph.
	 * @param budget a maximum allowed number of different labels in the
	    cliques.
	 * @param verbose set this to true if all cliques should be printed.
	 */
	public BKL_PostX(LabelledGraph graph, int budget, boolean verbose) {
		super(verbose);
		this.graph = graph;
		this.budget = budget;
	}

	/**
	 * Alternative constructor with verbosity set to false.
	 * @param graph the input graph.
	 * @param budget a maximum allowed number of different labels in the
	    cliques.
	 */
	public BKL_PostX(LabelledGraph graph, int budget) {
		this(graph, budget, false);
	}

	@Override
	public void run() {
		BitSet labels = new BitSet(graph.l);
		BitSet P = new BitSet(graph.n);
		P.set(0, graph.n);

		BitSet R = new BitSet(graph.n);
		BitSet X = new BitSet(graph.n);

		extend(R, P, X, labels);
	}

	private void extend(BitSet R, BitSet P, BitSet X, BitSet labels) {
```

```
        inc();
        int firstSet = P.nextSetBit(0);
        // If the candidate set is not empty, continue the algorithm
        if (firstSet != -1) {
                for (int v = P.nextSetBit(firstSet); v > -1; v =
                    P.nextSetBit(v + 1)) {
                        BitSet newLabels = (BitSet)labels.clone();
                        updateLabels(R, v, newLabels);

                        // Add v to R
                        BitSet newR = (BitSet) R.clone();
                        newR.set(v);

                        // intersection of P and the neighbourhood of v
                        BitSet newP = (BitSet) P.clone();
                        newP.and(graph.neighbourhood(v));

                        // intersection of X and the neighbourhood of v
                        BitSet newX = (BitSet) X.clone();
                        newX.and(graph.neighbourhood(v));

                        // fill in labels for vertices in the candidate set
                            and remove those over the budget
                        updateSet(newR, newP, newLabels);

                        // call recursion
                        extend(newR, newP, newX, newLabels);

                        // remove v from candidates
                        P.flip(v);

                        // add v to X
                        X.set(v);
                }
        }
        // If maximal clique is found, report it and return
        else {
                updateSet(R, X, labels);
                if (X.nextSetBit(0) == -1) {
                        reportClique(R);
                }
        }
}

private void updateSet(BitSet R, BitSet set, BitSet labels) {
        for (int i = set.nextSetBit(0); i > -1; i = set.nextSetBit(i + 1))
            {
                BitSet bs = (BitSet)labels.clone();
                for (int j = R.nextSetBit(0); j > -1; j = R.nextSetBit(j +
                    1)) {
                        if (i != j)    { bs.set(graph.label(i, j) - 1); }
                }
                if (bs.cardinality() > budget)
                        set.clear(i);
        }
```

```
        }

        private void updateLabels(BitSet R, int v, BitSet labels) {
                for (int i = R.nextSetBit(0); i > -1; i = R.nextSetBit(i + 1)) {
                        labels.set(graph.label(i, v) - 1);
                }
        }

}
```

Bibliography

[1] Clique problem - Wikipedia. `http://en.wikipedia.org/wiki/Clique_problem`. Accessed: 23/03/2014.

[2] B. Balasundaram, S. Butenko, and S. Trukhanov. Novel approaches for analyzing biological networks. *Journal of Combinatorial Optimization*, 10(1):23–39, 2005.

[3] V. Batagelj and M. Zaversnik. An o(m) algorithm for cores decomposition of networks, 2002.

[4] Vladimir Boginski, Sergiy Butenko, and Panos M. Pardalos. Statistical analysis of financial networks, 2005.

[5] J. Bourjolly, G. Laporte, and G. Pesant. An exact algorithm for the maximum k-club problem in an undirected graph. *European Journal of Operational Research*, 138(1):21 – 28, 2002.

[6] C. Bron and J. Kerbosch. Algorithm 457: Finding all cliques of an undirected graph [h]. *Communications of the ACM*, 16(9):575–579, 1973.

[7] F. Carrabs, R. Cerulli, and P. Dell'Olmo. A mathematical programming approach for the maximum labeled clique problem. *Procedia - Social and Behavioral Sciences*, 108(0):69 – 78, 2014. Operational Research for Development, Sustainability and Local Economies.

[8] F. Cazals and C. Karande. A note on the problem of reporting maximal cliques. *Theoretical Computer Science*, 407(1-3):564–568, 2008.

[9] D. Eppstein, M. Löffler, and D. Strash. Listing all maximal cliques in sparse graphs in near-optimal time. In Otfried Cheong, Kyung-Yong Chwa, and Kunsoo Park, editors, *ISAAC (1)*, volume 6506 of *Lecture Notes in Computer Science*, pages 403–414. Springer, 2010.

[10] D. Eppstein and D. Strash. Listing all maximal cliques in large sparse real-world graphs. In *SEA*, volume 6630 of *Lecture Notes in Computer Science*, pages 364–375. Springer, 2011.

[11] David J. Johnson and Michael A. Trick, editors. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11-13, 1993*. American Mathematical Society, 1996.

[12] I. Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theoretical Computer Science*, 250(1-2):1–30, 2001.

[13] J. Leskovec. Stanford large network dataset collection. http://snap.stanford.edu/data/index.html. Accessed: 29/06/2014.

[14] R. Luce and Albert Perry. A method of matrix analysis of group structure. *Psychometrika*, 14(2):95–116, June 1949.

[15] C. McCreesh and P. Prosser. A Branch and Bound Algorithm for the Maximum Labelled Clique Problem. 2014.

[16] C. McCreesh and P. Prosser. Maximum cliques, k-cliques and k-clubs: Computational experiments. 2014.

[17] J. W. Moon and L. Moser. On cliques in graphs. *Israel J. Math.*, 3(1):23–28, 1965.

[18] M. E. J. Newman. Mark Newman data sets. http://www-personal.umich.edu/~mejn/netdata/. Accessed: 27/06/2014.

[19] S. Shahinpour and S. Butenko. Distance-based clique relaxations in networks: s-clique and s-club. In Boris I. Goldengorin, Valery A. Kalyagin, and Panos M. Pardalos, editors, *Models, Algorithms, and Technologies for Network Analysis*, volume 59 of *Springer Proceedings in Mathematics & Statistics*, pages 149–174. 2013.

[20] C. Stark, BJ Breitkreutz, T. Reguly, L. Boucher, A. Breitkreutz, and M. Tyers. A General Repository for Interaction Datasets. http://thebiogrid.org/download.php. Accessed: 24/08/2014.

[21] E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science*, 363(1):28–42, 2006.

[22] S. Tsukiyama, M. Ide, and I. Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM J. Comput.*, 6(3):505–517, 1977.