QATAR UNIVERSITY

COLLEGE OF ENGINEERING

A CONCEPTUAL HEURISTIC FOR SOLVING THE MAXIMUM CLIQUE PROBLEM

BY

ZEINEB SAFI

A Thesis Submitted to the Faculty of
the College of Engineering
in Partial Fulfillment
of the Requirements
for the Degree of
Master of Science Computing

June 2017

Committee

The members of the committee approve the Thesis of Zeineb Safi defended on May 14:

Prof. Ali Jaoua

Thesis Supervisor

Prof. Jihad Jaam

Committee Member

Prof. Yaochu Jin

Committee Member

Approved:

Khalifa Nasser Al-Khalifa, Dean, COLLEGE OF ENGINEERING

Abstract

Safi, Zeineb, Masters:

June: 2017, Master of Science Computing

Title: A CONCEPTUAL HEURISTIC FOR SOLVING THE MAXIMUM CLIQUE PROBLEM

Supervisor of Thesis: Prof. Ali Jaoua

The maximum clique problem (MCP) is the problem of finding the clique with maximum cardinality in a graph. It has been intensively studied for years by computer scientists and mathematicians. It has many practical applications and it is usually the computational bottleneck. Due to the complexity of the problem, exact solutions can be very computationally expensive. In the scope of this thesis, a polynomial time heuristic that is based on Formal Concept Analysis has been developed. The developed approach has three variations that use different algorithm design approaches to solve the problem, a greedy algorithm, a backtracking algorithm and a branch and bound algorithm. The parameters of the branch and bound algorithm are tuned in a training phase and the tuned parameters are tested on the BHOSLIB benchmark graphs. The developed approach is tested on all the instances of the DIMACS benchmark graphs, and the results show that the maximum clique is obtained for 70% of the graph instances. The developed approach is compared to several of the most effective recent algorithms.

Dedication

For my parents.

Acknowledgements

First and foremost, I would like to thank my parents. I thank my mother for her unconditional love and support. My father for being such a great idol and setting an example for me to follow. I thank them both for believing in me and pushing me to always give my best. My deepest thanks and gratitude to my supervisor Prof. Ali Jaoua, who I learned a lot from, and who's door was open for me whenever I needed. I would also like to thank my team, Eman Rizk for her valuable advice, Dr. Abdelaali Hassaine, Fahad Islam and Abubakr Aqle for their feedback and support.

This contribution was made possible by NPRP grant #06-1220-1-233 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 Overview

Optimization is very important in our human life. We constantly strive to optimize our productivity, our time consumption and even the roads we take. Factories seek optimization of their production by optimizing their production chains. Governments make great efforts to optimize the allocation of their resources. As possibilities grow larger and the number of choices increases, optimization becomes harder and harder to achieve. All of these problems are related to what is known in information theory as "Combinatorial Optimization", which is a set of problems in which an optimal object is sought from a set of objects.

The maximum clique problem has been researched intensively for decades as an important combinatorial optimization problems. The k-clique problem, checking if a clique of size k exists in a graph, is one of the 21 original NP-Complete problems introduced by Karp [28]. The maximum clique problem is not just hard to solve but also approximate solutions for it are hard to find [6]. It has many generalizations and relaxations. In the maximum weight clique problem, graph edges have weights associated to them and the clique with the highest weight needs to be found. The k-plex, quasi-clique, k-subgraph problems, are all relaxations of the constraints on the density and the number of adjacent vertices of the maximum clique to be found. All of these have

important applications in different domains, and because of their difficulty, efficient exact and heuristic solutions need to be developed.

In applied mathematics, Formal Concept Analysis is the field that is concerned with deriving a conceptual hierarchy from a collection of objects and their corresponding attributes. The survey conducted by Singh et. al. [47], illustrates the usefulness of formal concept analysis in solving problems in various fields of applications including Software Engineering, Social Network Analysis, Information Retrieval, Security Analysis etc. The sound mathematical foundation of conceptual methods, can be applied to devise solution to problems like the maximum clique, and that is what we attempt to explore in the scope of this thesis.

1.2 Maximum Clique Applications

The maximum clique problem is motivated by its significance in solving practical problems in various fields, that are both interesting and important. In bioinformatics, the comparison of protein structures is an important task that can help in developing medical treatments that are based on proteins. Some methods that compare protein structures can be represented as a maximum clique problem as in the work done by Malod-Dognin et. al. [34], Ravetti and Moscato [40] and Strickland et. al. [50].

Determining the maximum size of a code satisfying a minimum Hamming distance is a challenging coding theory problem. One of the solution approaches is to construct a Hamming graph such that two sequences of length n are vertices and they have an edge conncting them only if their hamming distance is larger than or equal to d. Sloane [48] and Etzion and Östergård [14] formulated the bound on the hamming distance as a maximum independent set problem.

The maximum clique and its equivalent problems also have applications in economics. One of which is analyzing the stock market and drawing conclusions from it. The stock market is represented by a market graph, where stocks are vertices and a correlation between pairs of stocks is marked by an edge. An example of this is the work done by Boginski et. al. [8].

In the field of wireless networks and telecommunication, Chen et. al. [13] modeled the task of estimating path available bandwidth in multirate and multihop wireless network as a weighted maximum clique problem. Similarly, Jain et. al. [26] modeled inference between neighboring nodes in a network as a conflict graph and modeled the problem of finding a lower bound on the optimal throughput as a maximum independent set problem.

Social network analysis is an emerging field that is gaining a lot of popularity. A social network graph has actors or users as nodes and ties between them as edges. The maximum k-plex problem is one of the relaxations of the maximum clique problem, that entails finding subgraphs with high density and not necessarily complete. Many social network analysis problems can be modeled as the maximum k-plex problem, such as the work done by Balasundaram et. al. [4]. k-plexes have also been used for data mining and graph-based clustering applications as in the work done by Balasundaram [3].

1.3 Thesis Objective

The focus of this thesis is the extraction of maximum cliques from a simple unweighted graph using conceptual methods. The following are the objectives of the thesis:

- Design a heuristic solution, that uses formal concept analysis as a foundation, for solving the maximum clique problem.

  - Design a new efficient greedy algorithm that is based on formal concept analysis.
  - Escape the local optimality of the greedy algorithm using a backtracking design.
  - Explore a larger portion of the hyper concept tree using a branch and bound design.

- Investigate the use of different algorithm design approaches in developing an efficient and effective solution.
- Compare the performance of the developed method to other published results using benchmark graphs.

1.4 Thesis Outline

In Chapter 2 we present the necessary background information and formal definitions for the maximum clique and related problems, and conceptual methods, which we base our solution on. In Chapter 3 a review of developed exact and approximate solutions of the maximum clique and other NP-Hard problems is presented. Our developed solution and its different variations are presented in Chapter 4, and the obtained results are discussed in Chapter 5. Chapter 6 concludes the thesis and presents some future work.

Chapter 2: Background

Here we presents the background needed for the reminder of the thesis. In Section 2.1 we present the maximum clique problem and other problems that are equivalent to it with examples. Section 2.2 presents a few of the main complexity classes and the one under which the maximum clique problem falls. Section 2.3 presents the formal concept analysis background and Section 2.4 contains a brief description of the different algorithm design methods.

2.1 Maximum Clique and Related Problems

Graph theory is a very diverse and rich field. Many different types of graphs exist and are useful for different applications. For the purpose of this work we are interested in studying a specific type of graphs that has the following characteristics [10]:

- **Finite:** Contains finite vertex and edge sets.
- **Simple:** An edge always has unique ends. Two vertices can be connected by only one edge.
- **Connected:** Every vertex is reachable from any other vertex.
- **Undirected:** Edges have no directions.
- **Unweighted:** Edges have no weights assigned to them.

Definition 1 *"A **graph** is defined by $G = (V, E)$ where $V = \{1, ..., n\}$ is a set of vertices and $E = \{e_1, ..., e_m\}$ is a set of edges. An edge $e = (u, v)$ where $u, v \in V$. If $e \in E$ u and v are said to be adjacent. [10]"*

Figure 2.1 shows a finite, simple, connected, undirected and unweighted graph.

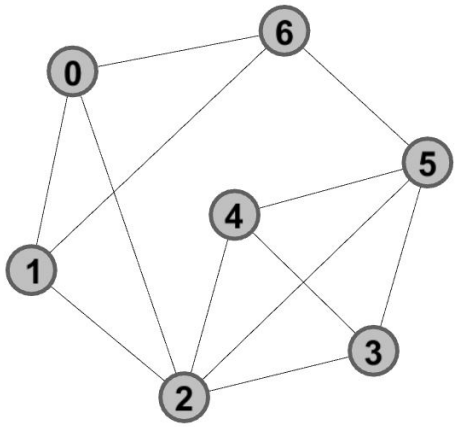


Figure 2.1: Finite Simple Connected Undirected and Unweighted Graph

Definition 2 *"$\bar{G}$ denotes the **complement** of the graph $G = (V, \bar{E})$, where $\bar{E}$ is the complement of E. If $e \in E$ then $e \notin \bar{E}$ similarly, if $e \notin E$ then $e \in \bar{E}$."*

Figure 2.2 shows the complement of the graph in Figure 2.1

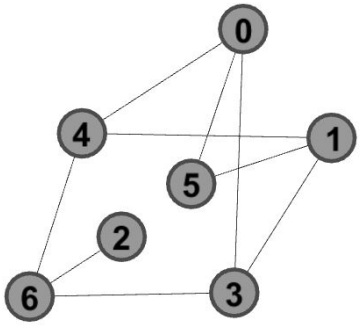


Figure 2.2: Graph Complement

Definition 3 *"A **subgraph** of G is the graph $G' = \{V', E'\}$ where $E' \subseteq E$ and $V' \subseteq V$. "*

Definition 4 *"The **cardinality** of a graph or a subgraph is the number of its vertices."*

Definition 5 *"A subgraph C of G is a **clique** if it is complete (all its vertices are pairwise adjacent). Figure 2.1 contains a few cliques, the vertices {2, 4, 5} for example form a clique of cardinality 3."*

It is important to distinguish between maximal and maximum cliques.

Definition 6 *"A clique is **maximal** if it cannot be enlarged by adding more vertices to it. A clique is **maximum** if it has the maximum cardinality."*

In Figure 2.1 vertices {0, 1, 6} form a *maximal* clique, while vertices {2, 3, 4, 5} form the *maximum* clique of the graph.

Definition 7 *"The **clique number** $\omega(G)$ is the cardinality of the maximum clique in a graph G."*

The clique number in Figure 2.1 is $\omega(G) = 4$.

Definition 8 *"An **independent set** of the graph G is a subset of the vertices of the graph $I \subseteq V$ where for any two vertices $u, v \in I$, $(u, v) \notin E$."*

If C is a clique in G, then C is an independent set in $\bar{G}$. Similarly, if C is a maximum clique in G, C is a maximum independent set in $\bar{G}$.

Definition 9 *"The **independence number** $\alpha(G)$ is the cardinality of the maximum independent set in the graph G."*

Definition 10 *"If V' is a subset of V and every edge $(u, v) \in E$ has at least one end point in V', V' is called a **vertex cover**. "*

In Figure 2.3 graph (c) to the right of shows a maximum clique colored in blue. Graph (b) is the complement of (c) and the corresponding maximum independent set is highlighted in blue, the vertex cover on the complement is highlighted in graph (a).

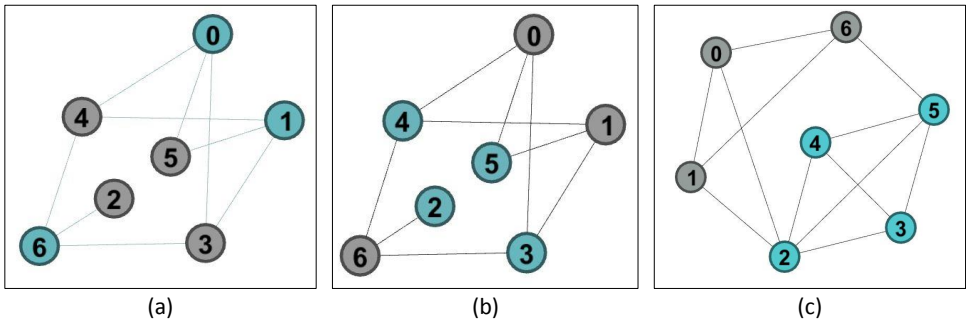


Figure 2.3: (a) Vertex Cover (b) Independent Set (c) Clique

More information and theoretical background about these problems can be found in [9].

2.2 Complexity Classes

We here present a brief overview of some complexity classes and the complexity class under which the maximum clique problem falls.

Definition 11 *"For any problem* Π*, if a polynomial time algorithm exits to solve* Π *with a time complexity of* $O(n^k)$*, where* n *is the size of the input and* k *is a non-negative integer then* Π *is called* ***tractable****, otherwise* Π *is called* ***intractable****."*

Definition 12 *"A* ***decision problem*** *can only have one of two outcomes yes or no, while in an* ***optimization problem*** *the concern is to minimize or maximize a certain quantity."*

The k-clique problem is a decision problem that takes a graph G and a non-negative integer k as input and returns a decision of whether or not G contains a clique of size k. The optimization version of it is the maximum clique problem, it requires determining the maximum value of k in G such that the graph contains a clique of size k and no cliques of size $k+1$.

Figure 2.4 shows the relationship between the 4 complexity classes.

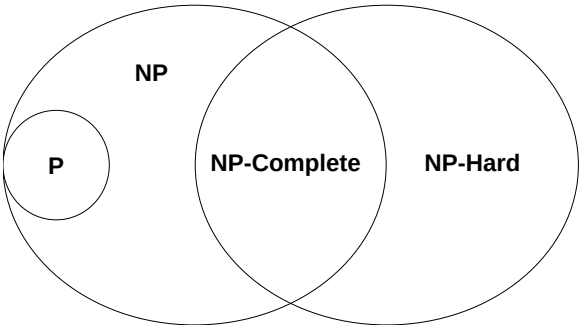


Figure 2.4: Complexity Classes

The class of problems NP (Nondeterministic Polynomial) is the set of problems Π the solution of which is verifiable in polynomial time using a deterministic algorithm. A deterministic algorithm is presented with only one choice at each step of its execution. The algorithms used to solve problems in the class NP are nondeterministic. There are two phases to a deterministic algorithm. The guessing phase, where an arbitrary candidate solution is generated in polynomial time. The verification phase, where the generated solution is verified using a deterministic algorithm[17].

The class P (Polynomial) is a subclass of NP. It consists of decision problems that can be solved using a polynomial deterministic algorithm.

NP-Complete problems are a subclass of intractable problems. Hundreds of problems fall under the class of NP-Complete, and if one of these problems has a polynomial time solution, then a polynomial time solution for all the other problems also exists.

Formally, NP-Complete problems can be defined as:

Definition 13 *"An **NP-Complete** decision problem* Π *must satisfy two conditions:"*

1. $\Pi \in NP$

2. $\forall\ \Pi' \in NP,\ \ \Pi' \propto_{poly} \Pi$

$\Pi' \propto_{poly} \Pi$ symbolizes that the problem Π is reducible to Π' in polynomial time. An instance of the problem Π can be transformed into and instance of the problem Π' using an algorithm that runs in polynomial time.

Definition 14 *"A problem* Π *is in the class **NP-Hard** if* $\forall\ \Pi' \in NP,\ \Pi' \propto_{poly} \Pi$*"*

A problem is NP-Hard, if it is at least as hard as the NP-Complete problems. The k-clique problem is an NP-Complete problem because:

1. k-clique $\in$ NP since a solution to the problem can be verified using a deterministic algorithm.

2. SAT $\propto_{poly}$ k-clique

The optimization version, the maximum clique problem is NP-Hard [17].

2.3 Formal Concept Analysis

Formal Concept Analysis (FCA) is the field of applied mathematics that mathematizes the underlying philosophical notion of "concepts". Traditionally used for knowledge processing and knowledge discovery, we here present another application of FCA as basis for solving some NP-Complete and NP-Hard problems. In this section, we present the mathematical foundations of Formal Concept Analysis, and hyper concepts in particular, necessary for the remainder of this thesis. The FCA background is in the most part is taken from the first chapter of Ganter and Wille's book [16].

2.3.1 Context, Concept and Concept Lattice

Definition 15 *"A **formal context** is a triplet $K = (G, M, R)$ where G is a set of objects, M is a set of attributes, and R is a relation between G and M. $(g, m) \in I$ indicates that object $g \in G$ is in a relation with attribute $m \in M$ and it is read as object g has the attribute m. "*

A context may be represented by a matrix. The matrix rows are objects and the columns are attributes. If $(g, m) \in I$, the value '1' is placed in the cell corresponding to row g and attribute m, '0' otherwise.

Figure 2.5 shows an example of a formal context where the set of objects $G = \{0, 2, 3, 4, 5, 6, 7\}$ and the set of attributes $M = \{a, b, c, d, e, f, g, h\}$.

	a	b	c	d	e	f	g	h
0	1	0	0	0	0	1	0	0
1	1	0	0	0	0	1	1	0
2	1	1	0	0	0	1	1	0
3	0	1	0	0	0	1	1	1
4	1	0	0	1	1	0	0	0
5	1	1	1	0	1	0	0	0
6	0	1	1	1	0	0	0	0
7	0	1	1	0	1	0	0	0

Figure 2.5: Formal Context Example

Definition 16 *"A **formal concept** is a pair (A, B) with $A \subseteq G$, $B \subseteq M$, $A' = B$ and $B' = A$. A' is the set of attributes common to all objects in A. B' is the set of objects common to all attributes in B. "*

$A' := \{m \in M(g, m) \in I \ \forall g \in A\}$

$B' := \{g \in G(g, m) \in I \ \forall m \in B\}$

A formal concept is also referred to as a maximal rectangle or a non-enlargeable rectangle.

Figure 2.6 shows the formal context of the previous example with a highlighted formal concept where the set of objects $A = \{1, 2\}$ share the set of attributes $B = \{a, f, g\}$.

	a	b	c	d	e	f	g	h
0	1	0	0	0	0	1	0	0
1	1	0	0	0	0	1	1	0
2	1	1	0	0	0	1	1	0
3	0	1	0	0	0	1	1	1
4	1	0	0	1	1	0	0	0
5	1	1	1	0	1	0	0	0
6	0	1	1	1	0	0	0	0
7	0	1	1	0	1	0	0	0

Figure 2.6: Formal Concept Example

Definition 17 *"The **concept lattice** is a hierarchical organization of all concepts extracted from a context (G, M, I) in subconcept, superconcept relation. Given two concepts (A_1, B_1) and (A_2, B_2), (A_1, B_1) is a **subconcept** of (A_2, B_2) if $A_1 \subseteq A_2$. Consequently, (A_2, B_2) is a **superconcept** of (A_1, B_1). This is written as $(A_1, B_1) \leq (A_2, B_2)$. The relation $\leq$ is the **hierarchical order** of concepts."*

2.3.2 Hyper Concepts

Extracting all concepts from a formal context is a very time consuming process. In addition, the concept lattice is a large structure that requires a lot of storage space, and that grows exponentially as the size of the formal context grows. In practice, most applications do not require the full set of concepts. Minimal conceptual coverage aims to cover the given context by a smaller number of concepts while preserving the knowledge. The hyper concept algorithm developed by Hassaine et. al. [22] is one such attempt where the context is covered by optimal "hyper concepts". In [23, 22] hyper concepts are used as textual

features for classification of Islamic advisory opinions [22] and classifying news articles [23]. In [38] Otaibi el. al. used the hyper concept algorithm for feature extraction combined with conceptual reasoning for detecting inconsistencies in text.

Definition 18 *"A relation R is a subset of the Cartesian product of two sets X and Y"*

Definition 19 *"$R^{-1} = \{(e, e^{'})(e^{'}, e) \in R\}$ is the converse of the relation R."*

Definition 20 *"$m.R = \{x \in G(x, m) \in R\}$ is the image of m by the relation R."*

Definition 21 *"$I(A)$ is called the identity relation. It is a subset of the Cartesian product $A \times A$ such that $\forall m \in M, m.I(M) = \{m\}$"*

Definition 22 *"$R \circ R^{'} = \{(e, e^{'})\exists t((e, t) \in R)\&((t, e^{'}) \in R^{'})\}$ is the relative product or composition of two binary relations. "*

Definition 23 *"Given a formal context $K = (G, M, R)$ and $m \in M$ an arbitrary attribute. A **hyper concept** denoted as $H_m(R) = I(m.R^{-1}) \circ R$."*

A hyper concept corresponding to an attribute $m \in M$ is the union of all concepts containing the attribute m.

Figure 2.7 shows the formal context and the hyper concept corresponding to attribute a. The image of attribute a by the relation R^{-1} is $a.R^{-1} = \{0, 1, 2, 4, 5\}$. The hyper concept shown in the figure correspond to $I(a.R^{-1}) \circ R$.

Definition 24 *"The "importance" of a hyper concept $H_m(R)$ is measured by its weight $W(H_m(R))$."*

The calculation of the different weights depends on the following attributes:

	a	b	c	d	e	f	g	h
0	1	0	0	0	0	1	0	0
1	1	0	0	0	0	1	1	0
2	1	1	0	0	0	1	1	0
3	0	1	0	0	0	1	1	1
4	1	0	0	1	1	0	0	0
5	1	1	1	0	1	0	0	0
6	0	1	1	1	0	0	0	0
7	0	1	1	0	1	0	0	0

	a	b	c	d	e	f	g
0	1	0	0	0	0	1	0
1	1	0	0	0	0	1	1
2	1	1	0	0	0	1	1
4	1	0	0	1	1	0	0
5	1	1	1	0	1	0	0

Figure 2.7: Hyper Concept Corresponding to Attribute a

- d: The domain cardinality of $H_m(R)$
- c: The codomain cardinality of $H_m(R)$
- r: The cardinality of $H_m(R)$"

Definition 25 *"The economy of a binary relation (gain in storage space) is given by the formula:* $W(R) = \frac{r}{d \times c} \times (r - (d + c))$"

The quantity $\frac{r}{d \times c}$ is the density of the relation, while the quantity $(r-(d+c))$ is the economy of information [21].

The hyper concept tree is a hierarchical organization of hyper concepts. The hyper concepts at each level are extracted from the hyper concept of the previous level. The hyper concepts at each level are organized by weight in a non-increasing order. Assuming that the weight of the hyper concept is just "d", the cardinality of the domain, Figure 2.8 shows the first level of the hyper concept tree constructed from the formal context of Figure 2.5.

The hyper concept with the highest weight in the first level is the one corresponding to attribute a, since a has the highest domain cardinality. Figure 2.9 shows the second level hyper concepts extracted from the hyper concept corresponding to attribute a.

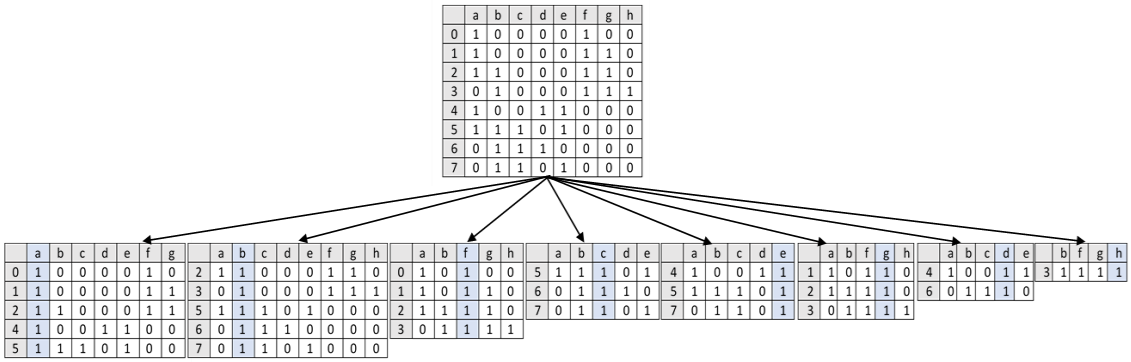


Figure 2.8: Hyper Concept Tree First Level

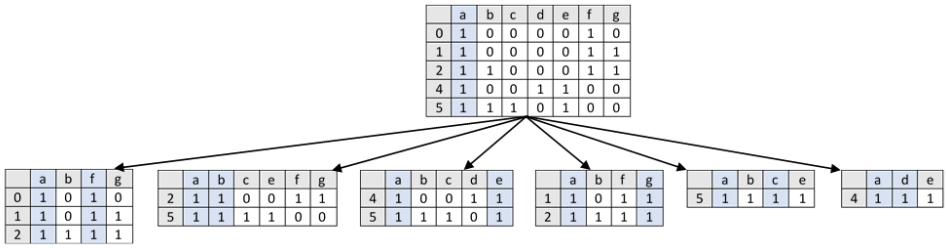


Figure 2.9: Hyper Concept Tree Second Level

2.4 Algorithm Design Methods

Different variations of our proposed solution use different algorithmic design methods. Here we introduce a few basic exact and approximate design methods.

2.4.1 Exact Design Methods

Divide and Conquer

A powerful algorithm design method that operates by dividing the problem instance into p subinstances and recursively solves each instance separately. Merging, or combining the solutions of the sub-instances results in a solution to the problem [1].

Dynamic Programming

Dynamic programming is similar to divide and conquer in that algorithms that employ it are usually stated in the form of recursive functions. Unlike divide and conquer, dynamic programming algorithms operate in a bottom-up manner. Smaller problem instances are solved first, and their intermediate solutions are saved to be used in computing the final result [1].

Backtracking

A systematic searching technique in which the search space is organized as a tree and the tree is traversed in a depth first manner. The search stops when a certain criterion is met [1].

Branch and Bound

A very similar design strategy to backtracking that is typically used for designing solutions to optimization problems. Like backtracking, the search space is orgranized in the form of a tree, but it does not impose any constraints on how the tree is traversed. In a branch and bound algorithm, each node in the tree has a bound to determine if the node is promising or not [1].

2.4.2 Approximate Design Methods

Greedy Method

Greedy algorithms are usually iterative algorithms that are also used for solving optimization problems. Greedy algorithms make decisions that will result in the maximum immediate gain without worrying about the future. The resulting solution is a local optimum, which might be translate to a global optimum [1].

Evolutionary Algorithms

Evolutionary Algorithms are inspired by the process of natural selection (increasing the existence of favorable traits in future generations) and evolution (changing the genetic makeup of populations). One type of Evolutionary algorithms is genetic algorithms. It operates by first randomly initializing a population that consists of individuals, where each one is an encoding of a possible solution to the problem instance. Crossover (exchanging substrings of the solution between pairs of individuals) and Mutation (adding, deleting or replacing a part of the individual by another) are performed recursively. The solution is evaluated in each iteration and the algorithms stop when a certain criteria is met [37].

Chapter 3: Literature Review

The maximum clique problem has many exact and approximate solutions. A few comprehensive surveys of these approaches were conducted. The first dates back to 1994 by Pardalos and Xue [39], followed by the survey of Bomze et. al. [9]. The most recent survey for the maximum clique algorithms is the work done by Wu and Hao [56], which is a comprehensive review of mathematical formulations, enumerative algorithms, exact solution approaches, and heuristic approaches for the maximum clique problem and a few other generalizations and relaxations of it. In our review, we focus on recent exact and approximate solution approaches of the unweighted maximum clique and related problems.

3.1 Exact Algorithms for the Maximum Clique Problem

Most of the exact solutions of the maximum clique and the related problems are based on a branch and bound algorithms. The algorithm enlarges an initial small clique until one with a maximum cardinality is found. Given a graph $G = (V, E)$, the algorithm maintains a list Q for the vertices of the current clique and a list Q_{max} for the largest clique already found, both of which are intially empty, and a candidate list $R \subseteq V$ that is initially set to V. The algorithm operates by adding a vertex $p \in R$ to Q, p is then removed from R along with all the vertices that are not connected to it. When $R = \emptyset$, if

$Q > Q_{max}$, Q_{max} is replaced by Q. The algorithm then backtracks and another vertex p is selected.

Most algorithms start from the general framework above and add improvements to the branching and pruning strategies, some approaches use approximate coloring to color the candidate set of vertices. $\chi(G)$ is the chromatic number of a graph. A clique in any given graph can never be larger than the chromatic number. This set of solutions uses approximate graph coloring of the candidate set of vertices as an upper bound.

Tomita and Seki [53] presented MCQ, an improvement to the basic algorithm that an approximation of $\chi(G)$ to set an upper bound and prune the search space. Colors are greedily assigned to vertices of the candidate set R. Each color class C_k is represented by an integer k. Colors are assigned to vertices as follows. A vertex v_1 is assigned a color $k = 1$, if vertex v_2 is adjacent to v_1 then v_2 is assigned a color $k = 2$, otherwise v_2 is assigned $k = 1$. Vertices in R are sorted based on their assigned colors in ascending order. The vertex selected from R to be added to Q is the one with the maximum color value. If adding the vertex of maximum value to Q will result in a clique that is smaller in size than Q_{max} ($Q + Max\{k\} \leq Q_{max}$) then the branch is disregarded.

The vertices in MCQ [53] before coloring are sorted in descending order of their degree in the graph G. MCR [52] developed by Tomita and Kameda, improves the initial ordering of vertices. Vertices are ordered into a list $L = \{v_1, v_2, ..., v_n\}$ such that v_n has the lowest degree of G, v_{n-1} has the lowest degree of $G\backslash\{v_n\}$, v_{n-2} has the minimum degree in $G\backslash\{v_n, v_{n-1}\}$ and so on. The remaining steps operate as in MCQ. In MCR, vertices chosen to expand the current clique are those that have the maximum color. As stated earlier, vertices that have colors that are smaller than a threshold will not produce a maximum clique eventually and the corresponding branches are pruned.

To further improve MCR, Tomita et. al. introduced MCS [54] that further reduces the search space for a maximum clique by employing a recoloring strategy. Not all vertices with colors greater than the threshold are promising. Vertices that do not fit a certain criteria and will not lead to a maximum clique, but still have colors greater than the threshold are recolored to have colors smaller than the threshold to prevent them from being selected.

The order by which the vertices are introduced to the coloring algorithm is important. While Tomita and Seki [53] order the vertices by their color in the candidate set R, Konc and Janezic [30] present an improvement to the algorithm in which the current clique is enlarged by vertices with large color number. The vertices are reordered by their degree in a decreasing order, keeping the vertices that have color numbers below the threshold in their original order. This makes the new developed approach, MaxCliqueDyn, faster than Tomita and Seki's original MCQ algorithm.

An improvement to MaxCliqueDyn is presented in the work of Segundo et. al. [41]. BB-MaxClique is an efficient bit parallel implementation, that uses basic operations between bit strings to obtain induced subgraphs during the search. The main contribution of the algorithm is that the order of the vertices in the set R does not change. The vertices are ordered in an increasing order of their degree and then presented to the coloring algorithm. The authors also present an improvement to the coloring algorithms, BB-Color, which obtains a color class in each iteration instead of assigning colors to vertices independently.

BB-MaxClique was later improved by Segundo et. al. [44] by integrating the recoloring strategy introduced in MSC into the bit parallel implementation.

Segundo and Tapia [42] introduced a relaxation to the basic coloring algorithm [53] in which they only compute $k_{min} - 1$ color classes at each iteration, where k_{min} is the minimum order of hidden clique in the induced subgraph that might lead to a clique larger than Q_{max} in the branch.

MCT is a final, recent improvement to MCS by Tomita et. al. [51] which is based on the observations that the algorithm used to find an initial clique from which the lower bound of the solution is obtained is important for the efficiency of the algorithm. The algorithm they used for finding an initial solution is k-opt local search [29] developed by Katayama et. al., that will be discussed later in Heuristic Solutions section. While the algorithm does not always produce the best solutions, it is used because of its high speed. The numbering and renumbering process used in MSC reduces the search space, but also introduces an overhead to the running time of the algorithm. MCT uses a three stage operation technique. In the first stage, the vertices are sorted based on their degree. At certain level of the tree, the algorithm switches to the regular sorting based on numbering (coloring). Finally, near the leafs of the tree, the ordering of the vertices is inherited from previous levels. The level at which the switching occurs depends on the density of the graph at that level.

The work introduced by Segundo et. al. [45] presents an improvement to the initial sorting of vertices that is intended to work with different algorithms. The paper presents the `NEW_SORT` algorithm that automatically switches between two novel sorting strategies, a degree based sorting and a color based sorting based on the density of the given graph. The degree based sorting operates by first sorting the vertices based on their width, and then sorting the fist k vertices in descending order based on their degrees. The color based ordering operates as follows. Initially, all vertices of the set V are copied to a set w_1. Iteratively, a vertex $v \in W_1$ is removed and assigned to a color C_k. Vertices connected to v are removed from W_1 and copied to W_2. When $W_1 = \emptyset$ the next color class is built.

The maximum clique problem is reducible to maximum satisfiablity problem, but max sat solvers are not very useful when it comes to finding maximum cliques. However, they can be used to improve the value of the upper bound.

Li and Quan presented the first attempt to use max sat and propositional logic as an upper bound in [33]. In their approach, MaxCLQ, Li and Quan first partition the graph into independent sets, and an independent set based encoding of the maximum clique into max sat is devised. A set of soft and hard clauses are extracted from each independent set. If the graph contains k independent sets in the graph, and the independent set MaxSat encoding contains s disjoint soft clauses, it is proven that $\omega(G) \leq k - s$.

Improvements to MaxCLQ are presented by Li and Quan [32] that adopts the incremental computation of the maximum clique degree presented in MaxCliqueDyn [30], and a relaxation of the soft clauses which allows the detection of more inconsistent subsets of clauses. Further improvements presented by Li et. al. [31] consist of incremental upper bound computation resulting in a more efficient algorithm.

A combination of MCS, and MaxSat [54, 32, 33] is presented by Maslov [36], resulted in an efficient exact maximum clique algorithm.

Cliquer is an algorithm developed by Östergård [49] that uses a completely different approach for finding maximum cliques. The algorithm starts by trying to find the maximum clique in a subset of the vertices of size $S_n = \{v_n\}$, and then proceeds to finding the maximum clique in $S_{n-1} = \{v_n, v_{n-1}\}$ the $S_{n-2} = \{v_n, v_{n-1}, v_{n-2}\}$ and so on until the maximum clique in S_1, which is the original graph is found. The algorithm uses information from the previously found larger graphs as a bound. This algorithm is not as fast as other developed approaches.

Exact algorithms guarantees extracting the maximum clique from any given graph but are usually very time consuming. In some applications, efficiency is more important than the size of the obtained cliques.

3.2 Heuristic Approaches for the Maximum Clique Problem

Some maximum clique applications are more sensitive to time constraints, and can tolerate some error in the size of the maximum cliques founds. For this purpose, many heuristic approaches have been developed, that use different strategies for finding maximum cliques, we present some of these approaches in this section.

3.2.1 Greedy Algorithms

Very few greedy algorithms for the maximum clique and related problems exist in the literature. Greedy algorithms are generally efficient but not very effective in finding maximum clique and in most cases face the problem of local optimality. The general framework for greedy algorithms is to either begin with an empty set of vertices and add vertices to it in a greedy manner until a maximum clique is found. Or starting with the full set of vertices, greedily remove vertices until a maximum clique is found. To avoid finding locally optimal solutions, some algorithms use a multi-start strategy. An example of this is the approach developed by Jagota and Sanchis [25], in which an initial, possibly empty clique C is expanded by adding vertices from the set $S = \{v \in G \backslash Cv$ is adjacent to every vertex in $C\}$. Positive integer weights are associated to vertices in S which determine the probability by which the vertex v_i can be added to the current clique. Vertices with larger weight values have higher probability of being selected. In the nonadaptive version (NA), the algorithm restarts k times with the same set of weights, and the largest clique in all iterations is recorded. In the adaptive weights (AW) version of the algorithm, the weights are modified between restarts. The last version is the adaptive initial

state (AI) version, in which the initial state of each iteration is influenced by whether or not a larger clique was found in the previous iteration.

The deep adaptive greedy search approach (DAGS) developed by Grosso et. al. [18] is a greedy approach that is based on swap moves and vertex weights. The swap moves are used to replace the current clique by another, more promising clique of equal size. As in [25], the vertecies are weighted and the weights are adaptively adjusted in each iteration.

3.2.2 Search Heuristics

Search heuristics are the most successful approach used for solving the maximum clique and related NP-Complete problems. The general framework of search heuristics is trying to satisfy an evaluation function by exploring a defined search space using a neighborhood function.

STABULUS [15] is one of the earliest attempts for utilizing tabu search as a solution to the maximum independent set problem. In their approach Friden et. al. first fix an upper bound k for the cardinality of the independent set $\alpha(G)$ to be obtained. A feasible solution partitions the set of nodes V into two sets, S, the set of nodes that are independent or almost independent, and $\bar{S}$, the set of remaining nodes. The neighborhood $N(s)$ of a solution $s = (S, \bar{S})$ is the set of all solutions that can be obtained as a permutation of one node $x \in S$ with one node $y \in \bar{S}$. The new solution s' is obtained from the initial solution s such that $s^{'} = (S\backslash\{x\} \cup \{y\}, \bar{S}\backslash\{y\} \cup \{x\})$. A Tabu list T is maintained, it contains the T last solutions. The selected pair (x, y) at each step is the pair that has the smallest value of $E(S\backslash\{x\} \cup \{y\})$, and that is not a member of T. If after $nbmax$ iterations an independent set was not found, another initial solution is randomly generated and the procedure is applied on it, where $nbmax$ is an

preset parameter. If after q initial solutions an independent set of size k is still not found, the size of k is reduced by 1 and the process is repeated.

Adaptive Multistart Tabu Search (AMTS) is a more recent approach developed by Wu and Hao [55] for solving the maximum clique problem. It is based on STABULUS and introduces a few improvements to it. The neighborhood in $N(s)$ is replaced by the constrained neighborhood $CN(s)$ that is restricted to permutations of vertices $u \in A$, where A contains the nodes that have the smallest degrees relavent to the subset S, and that do not appear in the tabu list. And $v \in B$, where B the nodes of $S \backslash V$ not in the tabu list T and that have the largest degree relevant to the subset S. This improvement allows for a more focused search space that is smaller in size than the one used in STABULUS. Some probabilistic move selection rules are applied when the search is stuck in local optimum to allow the exploring the search space.

Swap-Based Tabu Search (SBTS) developed by Jin and Hao [27] has one the best performances in the literature, it successfully finds the maximum independent set for all instance of the two most used benchmarks. The general idea of the algorithm is to randomly generate an initial solution S and then improve it using a set of intensification and diversification steps. The basic operation of this approach is the $(k, 1) - swap$, where $(k = 0, 1, 2, \geq 2)$. The swap operation consists in swapping a vertex from $V \backslash S$ with k adjacent vertices from current maximal independent set S. The set $V \backslash S$ is divided into 4 subset NS_0, NS_1, NS_2 and $NS_{>2}$ where the subscript corresponds to the number of vertices $v_j \in S$ that are adjacent to vertex $v_i \in V \backslash S$. When the value of k in a $(k, 1) - swap$ is 0 or 1, the performed operation is called an intensification step, it aims at obtaining a local optimum from the current solution by either adding a randomly selected vertex from the set NS_0 to S or replacing one vertex of NS_1 by one vertex of S following a set of rules that will guarantee an improvement to the current result. When the value of k in a $(k, 1) - swap$ is

2 or > 2, the performed operation is called an diversification step, it aims at perturbing the current solution to explore other search regions by replacing 2 or more vertices of S by one vertex of NS_2 or $NS_{>2}$. The vertices removed from the set S are added to the tabu list to ovoid cycling. Intensification operations are preferred over diversification operations and are given priority. A diversification operation is only performed when the sets NS_0 and NS_1 are empty, or when the vertices in them are members of the tabu list.

Reactive Local Search (BLS) by Battiti and Protasi [5] is a local search heuristic that dynamically determines the amount of diversification using historical information.

Variable Neighborhood Search (VNS) by Hansen et. al. [20] is a combination of a greedy heuristic and a local search heuristic that changes the neighborhood by computing choosing a solution with a closer distance.

Hoos and Pullan [24] presented a Dynamic Local Search (DLS) algorithm that alternates between two phases. During the iterative improvement phase the algorithm adds vertecies to the current clique to enlarge it. During the plateau search phase, swap operation with vertices outside the clique occur.

Breakout Local Search (BLS) developed by Benlic and Hao [7] is a local search strategy that can be used as a solution to the maximum weight clique and the maximum clique problems. BLS generates an initial solutions greedily by randomly selecting a vertex v and adding it to the set C. Vertices $u \in V \backslash C$ are iteratively added to the set C such that $\forall c \in C \ \{u, c\} \in E$. This process ends when no nodes can be added to C, it requires no initial upper bound on the size of the clique to be found, and it results in a valid clique. The proposed approach first uses local search to find a local optimum from the initial solution C by either adding v to the set C such that $\forall c \in C \ \{v, c\} \in E$, by adding a vertex v to C such that v is connected to all vertices of C except for one vertex u which is then excluded from C, or by just removing a vertex v from C. The

move that will result in a more improved solution is the one that is considered. The previously mentioned three moves are called *directed perturbations* to the initial solution C. The purpose of directed perturbations is to move the search from one local optimum to another. When no improved result is found after a certain number of iterations, the search is said to be stagnating, and stronger *random perturbation* is applied. The purpose of the random perturbation is to restart the search and explore another unseen area of the search space. The probability of performing one type of perturbation or another is determined by the search state, the probability of performing a random perturbation becomes higher as the number of non-improving local optima visited increases.

3.2.3 Evolutionary Algorithms

Evolutionary Algorithms are very efficient in practice but they do not give good results when used on their own and are usually paired with other approaches.

In 1993 Carter and Park [12] conducted an experiment to evaluate the usefulness of genetic algorithms in solving the maximum clique problem. They came to the conclusion that at the time, genetic algorithms are not as powerful as other general purpose heuristics. Since then, genetic algorithms have been used in combination with statistical information and search heuristics and produced acceptable results.

Marchioni's algorithm [35] GENE is a hybrid genetic algorithm, local search approach, When a local optimal solution is obtained, genetic operators, crossover and mutation, are applied to it. The stopping criteria of the algorithm is when a solution is reached or when a preset maximum number of iterations is exceeded.

Guturu and Dantu's [19] algorithm is another more recent example of the combination of evolutionary algorithms with search heuristics. The algorithm integrates several features of previous, successful approaches.

Zhang et. al. [57] developed an approach that outperformed GENE, that combines a genetic algorithm with statistical information. The algorithm introduced an evolutionary algorithm guided mutation (EA/G) operator, which uses global statistical information and the location of the current solution to make more informed decisions in the generation of the offsprings.

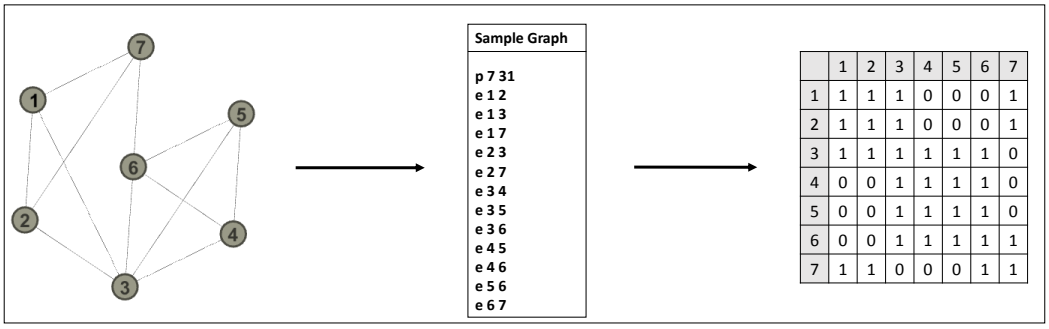
Singh and Gupta's Algorithm [46] consists of genetic algorithm where a steady state population replacement method is used, a greedy heuristic and an exact algorithm. While their algorithm outperformed some of the best genetic algorithms of the time, it did not perform better than heuristics that are not based on evolutionary methods.

Chapter 4: Methodology

Three main algorithm design approaches were used in our solution, a greedy approach, a partial backtracking approach and a partial branch and bound approach, all of which are based on the hyper concept algorithm presented earlier with some minor alterations made to it. Here we present the general framework used 4.1 and the details of each approach 4.2, 4.3 and 4.4.

4.1 General Framework

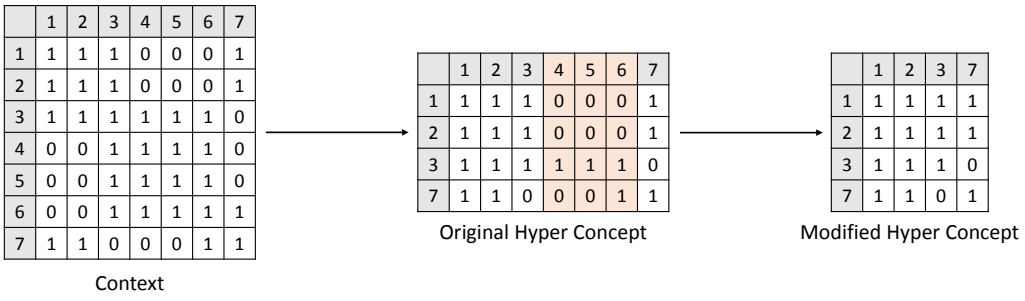
The first step of the solution is mapping the graph to a formal concept. The graph is initially stored in a text file that contains some metadata about the graph, like the method of generation, the density, the number of vertices and edges. The file also contains all edges of the graph in a two column format that indicate the source vertex and the destination vertex, proceeded by the letter "e" to indicate that the line corresponds to a graph edge. The file is read line by line and the formal context is built from it. The formal context is represented by a two dimensional array C in which both the rows and columns represent vertices of the graph. $C_{i,j} = 1$ in the produced context signifies that there is an edge between the nodes, $C_{i,j} = 0$ signifies that an edge does not exist. Even though the graphs are simple and they do not contain loops, the diagonal is set to 1 to facilitate the extraction of hyper concepts. Figure 4.1 shows the representation of the graph in the input file, and the formal context constructed from the input file.



	1	2	3	4	5	6	7
1	1	1	1	0	0	0	1
2	1	1	1	0	0	0	1
3	1	1	1	1	1	1	0
4	0	0	1	1	1	1	0
5	0	0	1	1	1	1	0
6	0	0	1	1	1	1	1
7	1	1	0	0	0	1	1

Figure 4.1: Graph to Formal Context

The hyper concept tree is constructed from the generated formal context by computing the hyper concept corresponding to each attribute and sorting them according to their weights. To maintain the symmetry of the relations, vertices that are not contained in the domain of the resulting hyper concept are also removed from the co-domain. Figure 4.2 shows the formal context built from the graph on the left, the hyper concept corresponding to attribute "1" in the middle, and the modified hyper concept on the right. The vertices in the domain of the original hyper concept are $\{1, 2, 3, 7\}$ and the vertices in the co-domain are $\{1, 2, 3, 4, 5, 6, 7\}$. Because vertices "4", "5", and "6" are not contained in the domain, they are also removed from the co-domain.



	1	2	3	4	5	6	7
1	1	1	1	0	0	0	1
2	1	1	1	0	0	0	1
3	1	1	1	1	1	1	0
4	0	0	1	1	1	1	0
5	0	0	1	1	1	1	0
6	0	0	1	1	1	1	1
7	1	1	0	0	0	1	1

Context

	1	2	3	4	5	6	7
1	1	1	1	0	0	0	1
2	1	1	1	0	0	0	1
3	1	1	1	1	1	1	0
7	1	1	0	0	0	1	1

Original Hyper Concept

	1	2	3	7
1	1	1	1	1
2	1	1	1	1
3	1	1	1	0
7	1	1	0	1

Modified Hyper Concept

Figure 4.2: Hyper Concept from Graph

Hyper concepts are prioritized by their weight as presented earlier. Different weight formulas have been used. The weight formulas are summarized in Table 4.1.

Table 4.1: Hyper Concept Weights

Weight ID	Original Formula	Modified Formula
weight1	d	d
weight2	r	r
weight3	$\frac{r}{d \times c} \times (r - (d + c))$	$\frac{r}{d \times d} \times (r - (d + d))$
weight4	$\log_{10}(\frac{r}{d \times c}) \times (r - (d + c))$	$\log_{10}(\frac{r}{d \times d}) \times (r - (d + d))$
weight5	$3 + \frac{\sqrt{9 - 8 * (d - r)}}{2}$	$3 + \frac{\sqrt{9 - 8 * (d - r)}}{2}$

Weight1 and weight2 are the cardinality of the domain and of the relation respectively. Weight1 gives priority to hyper concepts corresponding to vertices with higher cardinality, while weight2 give priority to hyper concepts containing the largest number of edges. Weight3 is the gain of a relation defined earlier. In weight4, the quantity $\frac{r}{d \times c}$ in the gain formula is replaced by $\log_{10}(\frac{r}{d \times c})$. The change made gives less value to the density of the relation than the economy of information. Finally, weight5 is a theoretical upper bound to the size of the maximum clique in a graph defined in [2]. Weight5 prioritizes hyper concepts that might contain larger cliques.

Due to the symmetry in the resulting hyper concept, the cardinality of the domain is equal to the cardinality of the co-domain ($d = c$). "c" is replaced in weight3 and weight4 by "d". Figure 4.3 shows the hyper concept tree corresponding to the context of Figure 4.1 and taking the cardinality of the domain as a weight (weight1). The leaf node of each branch is represents by a maximal clique. The graph contains three maximal cliques $\{3, 4, 5, 6\}$, $\{1, 2, 3\}$ and $\{1, 2, 7\}$, all of which are extracted by the hyper concept algorithm at different levels of the tree. Each node is represented by the nodes that are contained in the domain and codomain of the hyper concept. Some redundancy between hyper concepts might occur. For example, the hyper concept corresponding to

both node “1” and node “2” is $\{1, 2, 3, 7\}$. Only the hyper concept corresponding to node “1” is added to the tree.

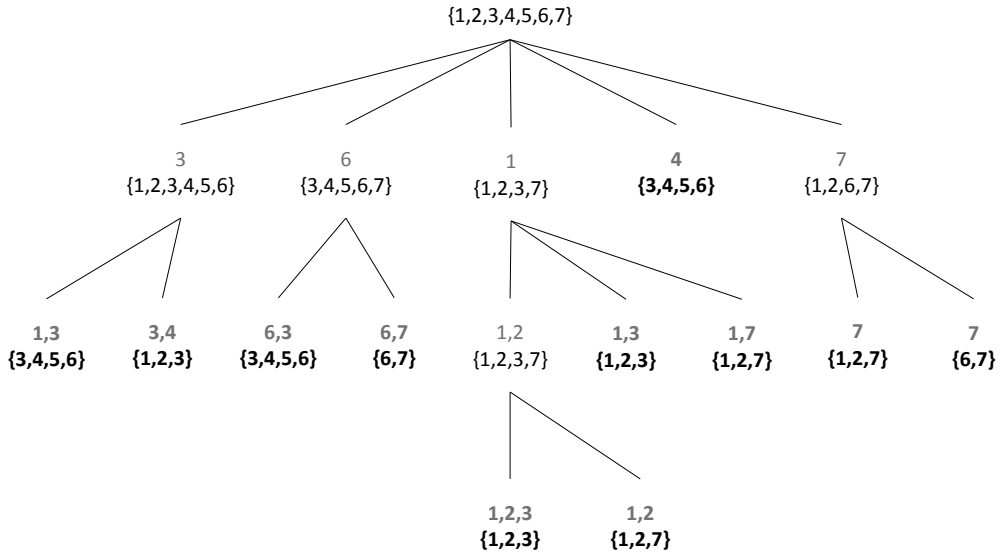


Figure 4.3: Hyper Concept Tree

Algorithm 1 shows the pseudocode of the hyper concept extraction procedure, which is the main step of the developed approach.

The algorithm takes a context or a hyper concept as and a “tabuList”. At the first level of the tree, the context is passed to *get_HC* and the hyper concept with the highest weight is returned. At any given level l of the tree, the hyper concept with the highest weight extracted from the current hyper concept in level $l + 1$ is returned. The method also takes a tabu list of vertices as input. The tabu list contains vertices that have already been chosen as vertices with highest weight in previous levels, and prevents them from being chosen again. In lines 2 to 8, the cardinality of the domain of hyper concepts corresponding to all vertices are computed and the values are stored in the array d. Lines 11 to 16, show the computation of the value of “r”. The index with the highest weight is recorded and the rows are copied from the context to the new hyper

Algorithm 1: Hyper Concept Extraction

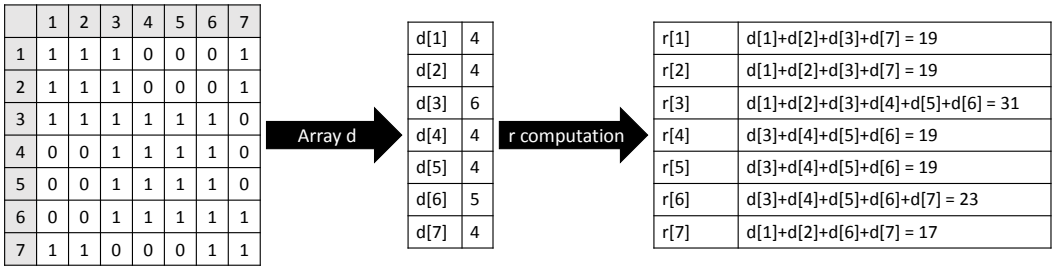
```
1  get_HC(context, tabuList);
   Input  : A formal context in the form of a 2D Integer array and a Tabu
            list
   Output: The hyper concept with the highest weight
2  for columnIndex ∈ columns do
3  |  for rowIndex ∈ rows do
4  |  |  if C[columnIndex][rowIndex] == 1 then
5  |  |  |  countd++;
6  |  |  end
7  |  end
8  |  d[i] = countd;
9  |  countd = 0;
10 end
11 for columnIndex ∈ columns do
12 |  for rowIndex ∈ rows do
13 |  |  if C[columnIndex][rowIndex] == 1 then
14 |  |  |  r+ = d[i];
15 |  |  end
16 |  end
17 |  //use value of r and d[i] to compute weight[i] according to the
   |   selected formula
18 |  if weight[i] > largestWeight and tabuList[i]! = 1 then
19 |  |  largetWeight = weight[i];
20 |  |  largestWeightIndex = i1
21 |  end
22 end
23 TabuList[largestWeightIndex] = 1
24 //Copy HC rows and return HC
```

concept. If two vertices have the same value for any given weight value the one that occurs first is selected.

Taking the operation at line 14 "$r+ = d[i]$" as basic operation, the worst case time complexity of this procedure is $O(n^2)$ where n is the number of attributes in the hyper concept.

Figure 4.4 shows how the computation of the cardinality of the relation is computed in a graphical form. The table in the middle shows the computation of the domain cardinality of each vertex, which is done by lines 2 – 10 of the

algorithms. The second table shows the computation of an r value of a specific vertex in each iteration of the loop starting at line 11.



	1	2	3	4	5	6	7
1	1	1	1	0	0	0	1
2	1	1	1	0	0	0	1
3	1	1	1	1	1	1	0
4	0	0	1	1	1	1	0
5	0	0	1	1	1	1	0
6	0	0	1	1	1	1	1
7	1	1	0	0	0	1	1

d[1]	4
d[2]	4
d[3]	6
d[4]	4
d[5]	4
d[6]	5
d[7]	4

r[1]	d[1]+d[2]+d[3]+d[7] = 19
r[2]	d[1]+d[2]+d[3]+d[7] = 19
r[3]	d[1]+d[2]+d[3]+d[4]+d[5]+d[6] = 31
r[4]	d[3]+d[4]+d[5]+d[6] = 19
r[5]	d[3]+d[4]+d[5]+d[6] = 19
r[6]	d[3]+d[4]+d[5]+d[6]+d[7] = 23
r[7]	d[1]+d[2]+d[6]+d[7] = 17

Figure 4.4: Graphical Representation of Algorithm1

In some cases a list of hyper concepts with highest weights needs to be returned instead of a single hyper concept. In this case, the process of recording extracting the highest weight index in lines 18 to 21 is discarded. The array weights is sorted and a list of the highest weight indecies is returned. The time complexity of the sorting process is $O(nlogn)$, this makes the time complexity of get_HCList $O(n^2 + nlogn)$ which is equivalent to $O(n^2)$.

Three different approaches have been used for extracting maximum cliques from a graph that are based on the hyper concept algorithm. The first is the greedy approach, which is our baseline, and the two improvements made to it, a partial backtracking approach, and a partial branch and bound approach.

4.2 Greedy Approach

The greedy algorithm is designed to provide a conceptual basis for solving the maximum clique problem and investigate the usefulness of the hyper concept algorithm in solving the maximum clique problem. Other greedy methods do exist in the literature, but a conceptual greedy algorithm is needed as a basis for further development of the solution.

The greedy approach is a depth first traversal of the tree, that only considers the first branch. At each level of the tree, the hyper concept with the highest

weight is selected. Figure 4.5 shows a depth first traversal of the hyper concept tree that starts with a formal context and ends with a clique in the leaf of the first branch. Here we use "d" as the weight of the hyper concept. The hyper concept corresponds to a clique when the cardinality of the relation is equal to the number of vertices squared $r = d^2$.

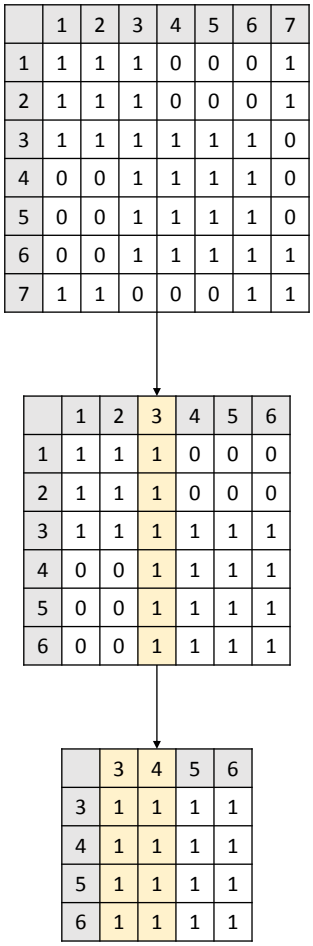


Figure 4.5: Clique Extraction Using the Greedy Approach

Algorithm 2 shows the recursive procedure followed for finding the maximum clique using the greedy method. Line 2 of the algorithm checks if the current context is a clique, in which case the size of the clique found is returned and the algorithm terminates. If it is not a clique, the function get_HC(context) is called in line 5. get_HC(context) extracts all hyper concepts from the context, computes their weights according to the selected formula and returns the

hyper concept with the highest weight. greedy_getClique(context) procedure is then called recursively until a clique is found. The worst case time complexity of the greedy approach is equivalent to $O(n^3)$

Algorithm 2: Finding the largest clique using the greedy approach

```
1 greedy_getClique (context);
  Input  : A formal context in the form of a 2D Integer array
  Output: the size of the maximum clique
2 if isClique(context) then
3 |   return clique_size;
4 else
5 |   context = get_HC(context, tabuList);
6 |   greedy_getClique(context);
7 end
```

4.3 Partial Backtracking

The greedy method often falls in the trap of local optimality. The backtracking method was designed to escape the local optimum and extract cliques from different branches.

Partial backtracking is an improvement to the greedy approach. Based on the assumption that the first clique obtained using the greedy approach is not necessarily the largest in the graph. After the largest clique is obtained, a normal backtracking algorithm would backtrack to the first node with an unvisited child, in our case, we backtrack all the way to the first level of the tree and select the node with the next highest weight. Figure 4.6 shows the order in which the nodes of the hyper concept tree are traversed in the backtracking approach. This approach entails traversing all the nodes of the first level, and for each node, the hyper concept with the highest weight is extracted and the algorithm proceeds similar to the greedy method. At each level of the tree at most n nodes are traversed and get_HC(context) is called in each node, where n is the cardinality of a graph.

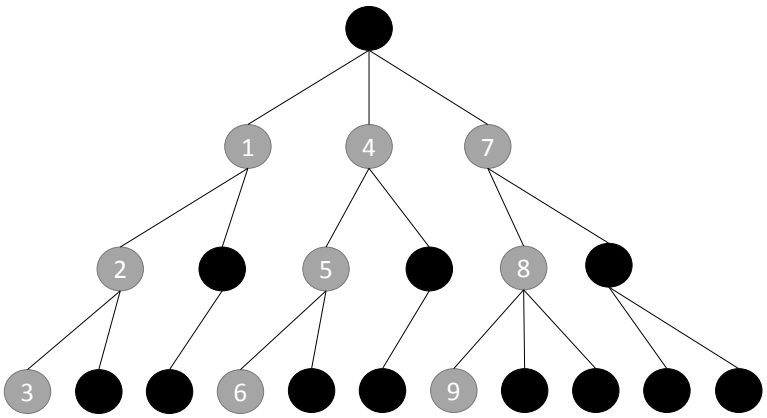


Figure 4.6: Backtracking Illustration

The backtracking procedure is explained in Algorithm 3. First, the indecies of the hyper concepts of the first level of the hyper concept tree are sorted according to their weights and stored in *indexList* using sortCliques(context). greedy_Cliques(index, context) in line 4 is used to get the clique at the leaf of the branch corresponding to the first level hyper concept of the current index. The index is a parameter that can be used to limit the number of branches traversed. In each iteration, the largest clique of each branch is returned, and the largest clique obtained from all iterations is reported by backtracking_getClique(context). The worst case time complexity of the backtracking approach is $O(mn^3)$, where m is the number of branches to be explored.

Algorithm 3: Finding the largest clique using the backtracking approach

```
1 backtracking_getClique (context);
  Input  : A formal context in the form of a 2D Integer array
  Output: The size of the maximum clique
2 indexList = sortCliques(context);
3 for index ∈ indexList do
4 |   cliqueSize = greedy_getClique(index, context) ;
5 |   if cliqueSize > largestClique then
6 |   |   largestClique = cliqueSize ;
7 |   end
8 end
9 return largestClique ;
```

4.4 Partial Branch and Bound

The branch and bound method was developed to escape the local optimum faced by the greedy method by exploring a larger area of the hyper concept tree.

Branch and bound is traditionally used for finding exact solutions for optimization problems. We here explore its use for obtaining an approximate solution by exploring only a portion of the tree by limiting the depth and the width at which the branching stops. In Figure 4.7 the depth is set to 2 and the width is set to 3. The nodes that are visited are highlighted in gray. For level 1 and level 2, the 3 nodes with the highest weight are considered, for the rest of the tree only the node with the highest weight is considered. It is clear here that the number of node that will be visited is significantly larger than that of the backtracking approach, and it grows fast as the values of depth and width increase.

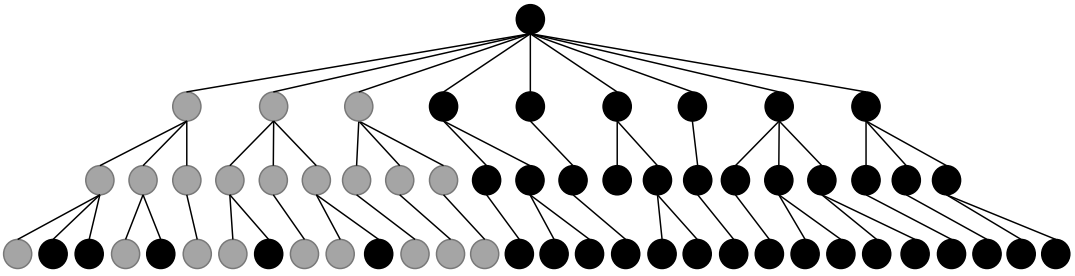


Figure 4.7: Branch and Bound Illustration

Algorithm 4 presents the branch and bound procedure. The algorithm uses a priority queue data structure that orders hyper concepts according to their weight in each level of the hyper concept tree. The queue is populated by objects of type "HC" which have the level, weight and a list the hyper concept vertices as attributes. For each hyper concept that is popped from the priority queue, one or more hyper concepts are extracted from it depending on the preset

depth and width parameters. If the level of the hyper concept is less than or equal to a preset value of the maximum level l, a list of w hyper concepts are extracted, where w is a preset value of the width. The extracted hyper concepts are added to the priority queue only if the size of the hyper concept is greater than the size of largest hyper concept obtained so far. If the size of the hyper concept is smaller than the largest clique, it is guaranteed that a larger clique cannot be obtained from it and it is pruned from the tree. If the hyper concept is at a level that is larger than the preset level l, then only one hyper concept with the highest weight is extracted from it. The extracted hyper concept(s) is added to the queue if it is not a clique. If it is a clique however, its size is checked, if larger than the current largest clique, it replaces it. The worst case time complexity of the branch and bound approach is $O(wdn^3)$ where w is the value of the width and d is the value of the depth of the hyper concept tree.

Algorithm 4 starts by pushing the passed context to the priority queue in line 2. The context is popped from the queue in line 4 and its level is checked. If the level is less than "l", a list of "w" hyper concepts are extracted. Each of the extracted hyper concepts is checked whether it is not a clique and whether its size is greater than the largest clique found so far, if that's the case then the hyper concept is added to the clique. If the hyper concept is a clique, and its size is greater than the current largest size, then its size is recorded as the largest in line 11. If the level is greater than l, only one hyper concept with the highest weight needs to be extracted, it is added to the queue if it is not a clique and if its size is greater that the current clique. This process is repeated until the queue is empty.

Algorithm 4: Finding the largest clique using the branch and bound approach

```
1  branchAndBound_getClique(context);
   Input  : A formal context in the form of a 2D Integer array
   Output: The size of the maximum clique
2  HCQueue.push(context) ;
3  do
4  |  HC = HCQueue.pop() ;
5  |  if HC.level() <= l then
6  |  |  HCList = get_HCList(HC, w) ;
7  |  |  for HC ∈ HCList do
8  |  |  |  if HC.size() > largestClique and !isClique(HC) then
9  |  |  |  |  HCQueue.puch(HC) ;
10 |  |  |  else if HC.Size() > largestClique and isCliqe(HC) then
11 |  |  |  |  largestClique = HC.Size();
12 |  |  end
13 |  else
14 |  |  HC = get_HC(HC);
15 |  |  if HC.size() > largestClique and !isClique(HC) then
16 |  |  |  HCQueue.puch(HC) ;
17 |  |  else if HC.Size() > largestClique and isCliqe(HC) then
18 |  |  |  largestClique = HC.Size();
19 |  end
20 while !HCQueue.isEmpty();
```

Chapter 5: Evaluation and Discussion

In this chapter we present the set up of the experiment 5.1 followed by a description of the used benchmark instances 5.2. After that we present the results of the greedy approach 5.3, the backtracking approach 5.4 and the branch and bound approach 5.5. Finally we present a comparison between the developed approach and the best published results 5.6.

5.1 Experimental Setup

The experiments have been performed on an HP envy laptop running windows 8.1. The PC has an Intel(R) Core i7-4702MQ processor and 16GB RAM. The implementation of all algorithms is sequential, written in Java programming language.

5.2 Benchmark

The 3 different variations of the algorithm are tested on DIMACS benchmark. DIMACS is the Center for Discrete Mathematics and Theoretical Computer Science that is a collaborative project of different institutions. The center held a few challenges, one of which focused on three NP-Hard problems: Satisfiability, Graph Coloring, and Maximum Clique. The graphs used during the challenge became the most widely benchmark for testing algorithms for the maximum clique, maximum weighted clique and even for the maximum independent set and vertex cover problems since 1992. The benchmark contains 80 different

cliques belonging to 12 different families. The families differ in the method by which the clique are generated which provides a wide variety of graphs. The graph sizes range between 28 to 4000 vertices. The following is a summary of the families.

- brock: Based on the work done by [11] which attempts to hide large cliques in graphs between smaller size cliques.
- C: Randomly generated graphs.
- c-fat: Generated from fault diagnosis data.
- DSJC: Randomly generated graphs.
- gen: Artificially generated graphs by Sanchis [43]
- hamming: Graphs that are based on the hamming distance between words. If the words are at least a hamming distance apart an edge is formed between them.
- johnson: Undirected graphs generated from systems of sets. An edge is formed if the intersection of the sets contains at lease k-1 elements.
- keller: Graphs based on Keller's conjecture on tilings.
- MANN: Graphs generated from converting the set covering formulation.
- p-hat: Randomly generated graphs.
- san: Randomly generated graphs.
- sanr: Randomly generated graphs.

The BHOSLIB benchmark was used to evaluate the optimal parameters for the branch and bound algorithm. The benchmark instances are transformed from a SAT benchmark where the vertices correspond to variables and the edges correspond to binary clauses.

5.3 Results of Greedy Approach

In this experiment, we tested the greedy algorithm previously presented on the full set of benchmark files using the 5 different weight values. The obtained results are summarized in Table 5.1. Weight, refers to the weight formula presented in Table 4.1. The second column, Clique Count, shows the number of maximum cliques found in the 80 benchmark files, and the percentage between parenthesis. When the maximum clique is not found, we use the distance between the largest clique found by the algorithm and the clique number of the graph ω as an indication of the quality of the given weight. We report here the maximum, minimum and average distance. Time (s) is the total time in seconds needed to find solutions to all the graphs.

Table 5.1: Greedy Method Results

Weight	Clique Count(%)	max distance	min distance	avg distance	Time (s)
weight1	17(21.25)	26	1	6.60	186.39
weight2	18(22.5)	28	1	7.01	116.44
weight3	13(16.25)	50	3	13.61	93.44
weight4	19(23.75)	28	1	7.16	174.32
weight5	18(22.5)	28	1	7.01	182.60
All Weights	19(23.75)	26	1	6.40	489.72

Weight4 produced the best result for the greedy approach which is the modified gain formula that gives more importance to the economy of information. using weight4, 19 out of 80 maximum cliques were extracted, and the average distance from the maximum clique is 7.16 nodes. weight1, weight2 and weight5, produced results that are close to that given by weight1. Weight1 has a smaller minimum distance and smaller average distance. The last row is the result of sequentially running the greedy algorithm with all possible weight values, and reporting the largest clique found. The number of cliques is the same as that found using weight4, but it results in a smaller distance average. The time taken to extract cliques from all the benchmark graphs is 489.72 seconds,

which is acceptable. The full results of each weight on all benchmark graphs are listed in the Appendix.

5.4 Results of Partial Backtracking

A similar experiment has been conducted to test the backtracking approach. The results in Table 5.2 correspond to backtracking 50 times to the first level for each given weight.

Table 5.2: Partial Backtracking Results

Weight	Clique Count(%)	max distance	min distance	avg distance	Time (s)
weight1	32(40.0)	26	1	4.33	4323.81
weight2	33(41.25)	25	1	4.77	4323.81
weight3	15(18.75)	32	1	9.68	3760.69
weight4	33(41.25)	26	1	4.81	4235.93
weight5	33(41.25)	25	1	4.79	4538.24
All Weights	33(41.25)	25	1	4.78	56964.89

Weight2, weight4, weight5 and weight1 produce the best results of this approach, with 33 or 32 maximum cliques extracted out of 80 benchmark graphs, which presents a slight improvement to the greedy approach. Weigh3 produces worse results. The time taken by this approach is significantly larger than that taken by the greedy algorithm. Performing the same experiment by sequentially trying all weights results in the same number of maximum cliques, which indicates that one weight is sufficient to find all the maximum cliques. The detailed set of obtained results on all graphs is listed in Appendix.

5.5 Results of Branch and Bound

This is the last modification made to basic approach. We tested the method on the different benchmark files, using the different weight measures and with varying values of depth and width for branching. We tested the approach with

depths 1 and 2, and width values 10 to 100 with increments of 10. Figures 5.1, 5.2, 5.3, 5.4, and 5.5 show the summary of the results obtained using each weight formula. In each figure, the horizontal axis represents the corresponding depth and width values, the value 1–10 signifies depth 1 and width 10. The primary vertical axis represents the total number of maximum cliques found at each depth and width values out of the 80 benchmark cliques. The secondary vertical axis represents the total time in seconds. Two values for the total time are reported, the total time taken by algorithm for the 80 benchmark graphs, shown in red, and the total time needed to find the maximum clique in a graph, shown in green.

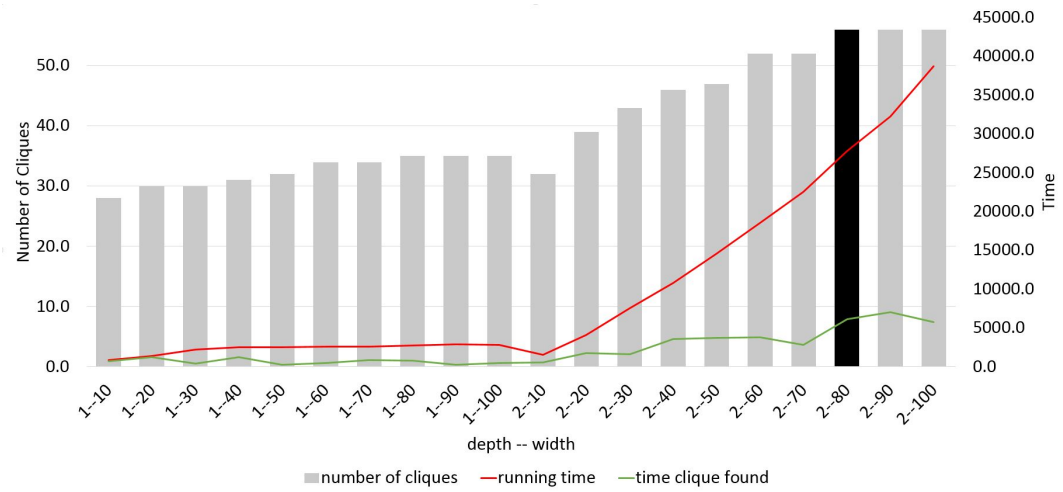


Figure 5.1: Branch and Bound Summary for Weight1

Weight1 shows good performance for this approach. The number of maximum cliques obtained by weight1 ranges from 28 at depth 1 and width 10, to 56 at depth 2 width 80.

The maximum number of maximum cliques obtained by weight2 is slightly lower than than obtained by weight1, 54 out of 80, but is still presented a good overall performance.

Weight3 shows a poor performance, with only 28 out of 80 maximum cliques are extracted at most.

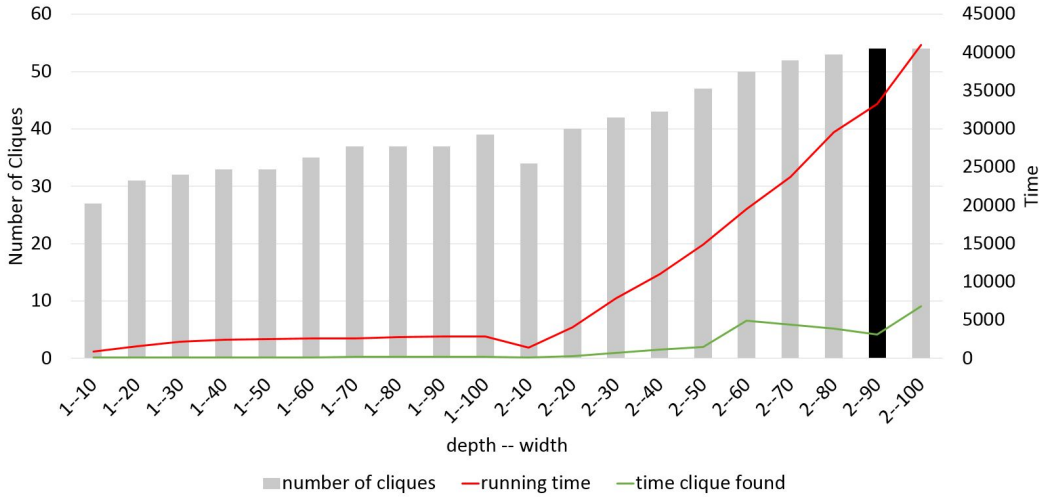


Figure 5.2: Branch and Bound Summary for Weight2

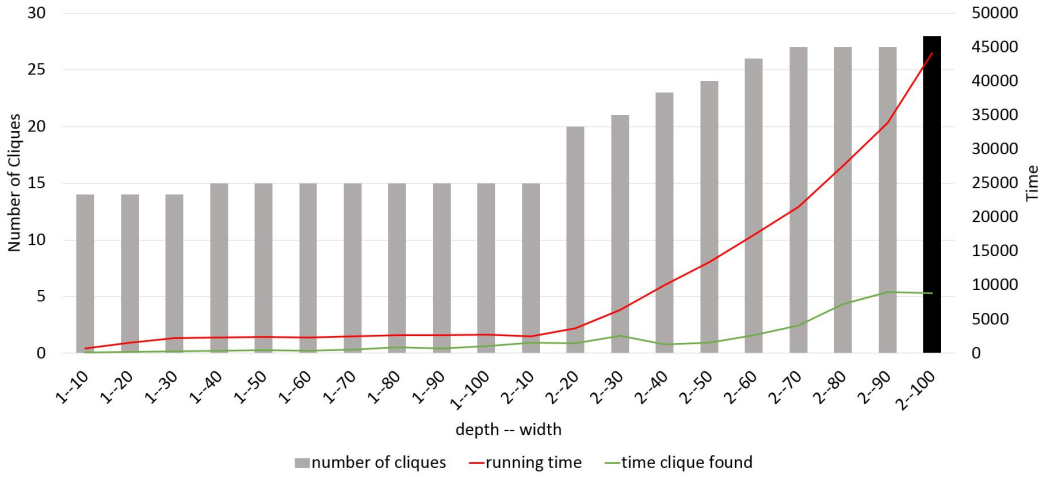


Figure 5.3: Branch and Bound Summary for Weight3

Weight4 and weight5 both have a similar acceptable performance with 47 out of 80 cliques extracted, but not as good as the results obtained by weight1 and weight2.

The maximum number of maximum cliques is found using weight1 at depth 2 and width 70, where 57 cliques are found out of 80 (70% of the graphs).

Table 5.3 summarizes the results obtained from all the weights. The average distance obtained using weight1, weight2, weight4 and weight5 is less than 5 nodes. This means that even when the largest clique is not obtained, the size of the clique is close.

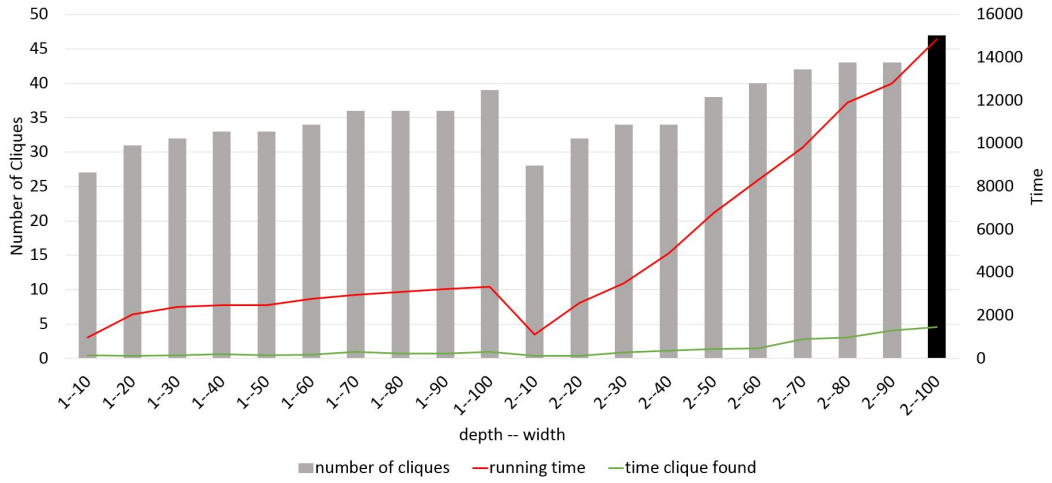


Figure 5.4: Branch and Bound Summary for Weight4

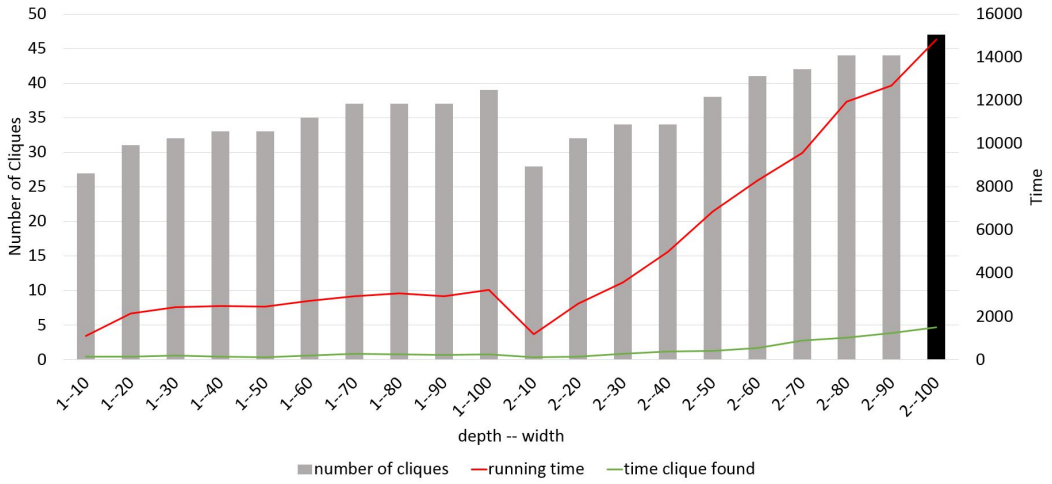


Figure 5.5: Branch and Bound Summary for Weight5

Table 5.3: Branch and Bound Results

Weight	Clique Count (%)	max distance	min distance	avg distance
weight1	56 (70.0)	22	1	4.75
weight2	54 (67.5)	21	1	4.27
weight3	28 (35.0)	219	1	12.94
weight4	47 (58.7)	21	1	4.27
weight5	47 (58.7)	21	1	4.27

The time needed to find the maximum clique increases, the running time of the algorithm also increases as the values of depth and width increase. But the time needed for finding the maximum clique increases at a much smaller rate, and is always much lower than the running time of the algorithm. This

means that the clique is found in the first few branches of the hyper concept tree, but the algorithm keeps searching until all the branches are explored.

Initially, the a time limit of 2000 seconds was used. If a solution was not found withing the first 2000 seconds of running the algorithm, the largest clique size is reported and the algorithm is forced to stop. The selected time limit was not very tight, and resulted in a the running time of the algorithm being very high, especially that the first clique if found usually found a lot before. We also observed that the time at which the first clique is found by the algorithm is indicative of the time needed to find the largest clique. Similarly, graphs with higher density are more likely to have a longer running time. Based on these observations, multivariate regression was used to obtain a dynamic estimation of the time that the largest clique will be found based on the time that the first clique was found and the graph densities.

As weight1 gave the best results using a depth of 2 and width of 70, we used the time values of obtaining the first clique output by this configuration along with the graph densities as input to the multivariate regression, and the output was coefficients of the time and density parameters along with the intercept shown in Table 5.4.

Table 5.4: Regression Parameter Coefficients

intercept	-66.395
density	188.65
first clique time	5.74

The experiments conducted so far in this section can be seen as a parameter tuning phase, to devise the best value of the weight, depth, width and time limit for the running time of the algorithm. To evaluate the effectiveness and efficiency of the tuned parameters, we tested these setting on some the graphs of the BHOSLIB benchmark. Table 5.5 is a summary of the obtained results.

Table 5.5: BHOSLIB Benchmark Results

graph	$\omega(G)$	density	with time limit		without time limit	
			clique size	run time	clique size	run time
frb30-15-1	30	0.82	29	91.49	29	83.63
frb30-15-2	30	0.82	29	91.15	29	104.51
frb30-15-3	30	0.82	28	90.38	28	103.81
frb30-15-4	30	0.82	28	90.55	28	89.22
frb30-15-5	30	0.82	28	89.85	28	95.98
frb35-17-1	35	0.84	32	94.86	32	191.22
frb35-17-2	35	0.84	32	98.18	33	159.42
frb35-17-3	35	0.84	32	95.70	33	189.27
frb35-17-4	35	0.84	32	93.86	32	167.53
frb35-17-5	35	0.84	32	93.57	33	180.94
frb40-19-1	40	0.86	36	101.50	36	504.01
frb40-19-2	40	0.86	36	98.96	37	546.28
frb40-19-3	40	0.86	36	98.29	37	531.90
frb40-19-4	40	0.86	37	98.62	37	467.40
frb40-19-5	40	0.86	37	97.20	37	466.56

In most cases, the cardinality of the obtained cliques without setting a time limit is the same as the size of the clique obtained using the time limit except for graphs frb35-17-2, frb35-17-3, frb40-19-2 and frb40-19-3, where the size of the clique found is one node smaller than that found after applying the limit. The running time on the other hand, is much lower when a time limit is applied.

5.6 Comparison with Other Approaches

Table 5.6 compares the best results obtained from the developed approach, branch and bound method using weight1 at depth 2 and width 70, to the most effective results in the literature as reported by [56]. The table shows the performance on the hardest instances of the benchmark. The reported times of all the algorithms are the times reported in the respective publications. The times are relative and only listed for comparison purposes. All the reported results are of sequential algorithms. The reported results correspond to [5, 35, 18, 20, 29, 57, 24, 46, 19, 55, 7, 27].

Table 5.6: Comparison Between Approaches

Instance	brock400_2		brock400_4		brock800_2		brock800_4		C2000.9		C4000.5		MANN_a45		MANN_a81		keller6	
	best(avg)	time	best(avg)	time	best(avg)	time	best(avg)	time	best(avg)	time	best(avg)	time	best(avg)	time	best(avg)	time	best(avg)	time
DAGS	29(28.10)	0.62	33	0.62	24(20.82)	3.73	26(22.60)	3.75	76(75.40)	405.33	18(17.50)	717.55	344(343.95)	426.95			57(56.40)	2739.11
IEA-PTS	29(27.52)	1.08	33	0.84	24(21.06)	1.03	26(21.4)	2.07	79(76.4)	19.71	18(17.66)	104.21	345(343.97)	9.43	1099(1097.01)	237.55	59(57.06)	45.4
VNS	29(27.4)	4.17	33	2.69	21	0.85	21	3.16	78(77.2)	22.74	18	310.71	345(344.5)	1.51	1100(1099.3)	65.47	59(58.2)	17.9
GENE	24(22.5)	0.27	25(23.6)	0.19	20(19.3)	0.75	20(18.9)	1.12	72(68.2)	4.89	16(15.4)	1.95	343(342.4)	19.56	1097(1096.3)	401.41	55(51.8)	8.71
RLS	29(26.06)	3.06	33(32.42)	7.89	21	0.34	21	0.48	78(77.58)	59.83	18	158.65	345(343.60)	28.98	1098	205.72	59	13.79
SBTS	29	11.97	33	0.49	24(22.29)	464.12	26(25.90)	249.47	80(77.29)	896.78	18	919.07	345	16.3	1100	13.43	59	446.67
DLS	29	0.12	33	0.02	24	3.97	26	2.24	78(77.93)	48.79	18	45.76	344	13.12	1098(1097.96)	66.66	59	43.05
KLS	25(24.84)	0.04	25	0.01	21(20.86)	0.16	21(20.67)	0.39	77(74.90)	4.8	18(17.02)	7.76	345(343.88)	2.02	1100(1098.07)	12.88	57(55.59)	17.08
HSSGA	29(25.1)	0.14	33(27.0)	0.29	21(20.7)	1.35	21(20.1)	0.38	74(71.0)	14.83	17(16.8)	19.97	343(342.6)	8.22	1095(1094.2)	503.99	57(54.2)	39.67
AMTS	29	0.69	33	0.35	24	19.61	26	9.01	80(78.95)	266.33	18	74.93	345(344.04)	66.77	1098	16.3	59	6.39
EA/G	25(24.7)	1.42	33(25.1)	1.42	21(20.1)	3.42	21(19.9)	3.42	72(70.9)	17.38	17(16.1)	23.46	345(343.7)	30.84	1098(1097.2)	319.04	56(53.4)	24.26
BLS	29	10.29	33	1.87	24(23.04)	637.94	26	356.05	80(78.6)	2846.84	18	387.33	342(340.82)		1094(1092.17)		59	14.67
HC	24	67.60 (1.09)	25	68.24 (27.22)	21	159.05 (131.28)	21	171.36 (49.3)	72	2000 (462.50)	17	1920.24 (71.42)	344	2000 (1732.98)	1098	2000 (1061.00)	54	2000 (6.47)

The results show that the maximum clique found by the hyper concept algorithm (in the last row) using the branch and bound design approach always has a size that is in the same range as some of the best approaches in the literature, the running time however, is slightly higher in most cases. Some of the approaches rely on randomization, the results between parenthesis are the average of 100 runs. The reported running time results are obtained without applying the learning approach. The time needed to find the largest clique is presented between parenthesis, it is usually a lot lower than the running time, and it can be obtained using regression. Furthermore, the hyper concept algorithm is implemented in Java, while most of the other solutions are implemented in lower level languages like C and C++.

Chapter 6: Conclusion

In the scope of this thesis, a heuristic conceptual approach for solving the maximum clique problem that runs in polynomial time has been developed. Several algorithm design approached have been designed and tested on the DIMACS benchmark graphs. The developed greedy approach is very efficient but the obtained results are not satisfactory in terms of obtained clique sizes. The backtracking approach slightly improved the greedy approach in terms of clique sizes but still does give the desired results. The branch and bound approach was the most successful in terms of obtained clique sizes but proven to no to be efficient. The parameters of the branch and bound approach have been optimized in a learning phase and tested on the BHOSLIB benchmark graphs.

The best obtained results we compared to the best result in the literature. The actual running time was high when compared to other approaches, but the time needed for finding the largest clique by the algorithm is much smaller and can be predicted through regression using historical information for obtaining previous cliques.

In the future, many improvements can be made to the algorithm to improve its performance. The current solution is implement in Java programming language, an implementation in a lower level language, like C might be much more efficient. Further improvements can be done to the data structures used. Conceptually, better upper and lower bounds to the branch and bound approach can be used to increase the efficiency of the algorithm. Learning through the re-

gression method can be extended to include the learning from the times needed to find the first two or three cliques instead of just the first clique to obtain a better estimation of the time time needed for finding the largest clique.

An improvement to the obtained results can be done by considering a sub ordering method when the vertices have equal weights for the main weight formula. For example, when the weight formula used is $\frac{r}{d \times c} \times (r - d - c)$, and two hyper concepts have equal highest weights, giving selecting the hyper concept with the higher value of $\frac{r}{d \times c}$ might result in obtaining a better results values faster. More consideration can be given to special numerical situations that could be causing a disadvantage for some of the weight formulas. For example, if the value of $(r - (d + c))$ is negative, the log of it will be undefined. In addition, squaring the matrix immediately after obtaining the vertex of the highest weigh then checking if the obtained hyper concept is clique will decrease the running time even further for the greedy approach. The performance of the developed approach might be assessed using many other weight formulas, for example the formula $\frac{r}{d}$ might produce better results.

Bibliography

[1] M. H Alsuwaiyel. *Algorithms*. World Scientific, 1 edition, 2010.

[2] A. T. Amin and S. L. Hakimi. Upper bounds on the order of a clique of a graph. *SIAM Journal on Applied Mathematics*, 22(4):569–573, 1972.

[3] B. Balasundaram. Cohesive subgroup model for graph-based text mining. In *2008 IEEE International Conference on Automation Science and Engineering*, pages 989–994, Aug 2008.

[4] Balabhaskar Balasundaram, Sergiy Butenko, and Illya V. Hicks. Clique Relaxations in Social Network Analysis: The Maximum k-Plex Problem. *Operations Research*, 59(1):133–142, February 2011.

[5] R. Battiti and M. Protasi. Reactive Local Search for the Maximum Clique Problem1. *Algorithmica*, 29(4):610–637, April 2001.

[6] Mihir Bellare, Oded Goldreich, and Madhu Sudan. Free Bits, PCPs, and Nonapproximability—Towards Tight Results. *SIAM J. Comput.*, 27(3):804–915, June 1998.

[7] Una Benlic and Jin-Kao Hao. Breakout Local Search for maximum clique problems. *Computers & Operations Research*, 40(1):192–206, January 2013.

[8] Vladimir Boginski, Sergiy Butenko, and Panos M. Pardalos. Mining market data: A network approach. *Computers & Operations Research*, 33(11):3171–3184, November 2006.

[9] Immanuel M. Bomze, Marco Budinich, Panos M. Pardalos, and Marcello Pelillo. The Maximum Clique Problem. In Ding-Zhu Du and Panos M. Pardalos, editors, *Handbook of Combinatorial Optimization*, pages 1–74. Springer US, 1999. DOI: 10.1007/978-1-4757-3023-4_1.

[10] J. A Bondy and U. S. R Murty. *Graph theory with applications*. Wiley, 1 edition, 2002.

[11] Mark Brockington and Joseph C. Culberson. Camouflaging independent sets in quasi-random graphs. In *In*, pages 75–88. American Mathematical Society, 1994.

[12] Robert Carter and Kihong Park. How Good are Genetic Algorithms at Finding Large Cliques: An Experimental Study. Technical Report, Boston University Computer Science Department, November 1993.

[13] F. Chen, H. Zhai, and Y. Fang. Available bandwidth in multirate and multihop wireless ad hoc networks. *IEEE Journal on Selected Areas in Communications*, 28(3):299–307, April 2010.

[14] T. Etzion and P. R. J. Ostergard. Greedy and heuristic algorithms for codes and colorings. *IEEE Transactions on Information Theory*, 44(1):382–388, Jan 1998.

[15] C. Friden, A. Hertz, and D. de Werra. STABULUS: A technique for finding stable sets in large graphs with tabu search. *Computing*, 42(1):35–44, March 1989.

[16] Prof Dr Bernhard Ganter and Prof Dr Rudolf Wille. Concept Lattices of Contexts. In *Formal Concept Analysis*, pages 17–61. Springer Berlin Heidelberg, 1999. DOI: 10.1007/978-3-642-59830-2_2.

[17] Michael R Garey and David S Johnson. *Computers and intractability*. Freeman, 1 edition, 2009.

[18] A. Grosso, M. Locatelli, and F. Della Croce. Combining Swaps and Node Weights in an Adaptive Greedy Approach for the Maximum Clique Problem. *Journal of Heuristics*, 10(2):135–152, March 2004.

[19] P. Guturu and R. Dantu. An Impatient Evolutionary Algorithm With Probabilistic Tabu Search for Unified Solution of Some NP-Hard Problems in Graph and Set Theory via Clique Finding. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 38(3):645–666, June 2008.

[20] Pierre Hansen, Nenad Mladenović, and Dragan Urošević. Variable neighborhood search for the maximum clique. *Discrete Applied Mathematics*, 145(1):117–125, December 2004.

[21] Ahmed Hasnah, Ali Jaoua, and Jihad Jaam. Conceptual Data Classification: Application for Knowledge Extraction. In Muhammad Sarfraz, editor, *Computer-Aided Intelligent Recognition Techniques and Applications*, pages 453–467. John Wiley & Sons, Ltd, 2005. DOI: 10.1002/0470094168.ch23.

[22] A. Hassaine, S. Elloumi, F. Ferjani, and A. Jaoua. Hyper rectangular trend analysis application to islamic rulings (fatwas). In *2014 IEEE/ACS 11th International Conference on Computer Systems and Applications (AICCSA)*, pages 318–325, Nov 2014.

[23] Abdelaali Hassaine, Souad Mecheter, and Ali Jaoua. *Text Categorization Using Hyper Rectangular Keyword Extraction: Application to News Articles Classification*, pages 312–325. Springer International Publishing, Cham, 2015.

[24] H. H. Hoos and W. Pullan. Dynamic Local Search for the Maximum Clique Problem. *arXiv:1109.5717 [cs]*, September 2011. arXiv: 1109.5717.

[25] Arun Jagota and Laura A. Sanchis. Adaptive, Restart, Randomized Greedy Heuristics for Maximum Clique. *Journal of Heuristics*, 7(6):565–585, November 2001.

[26] Kamal Jain, Jitendra Padhye, Venkata N. Padmanabhan, and Lili Qiu. Impact of Interference on Multi-hop Wireless Network Performance. In *Proceedings of the 9th Annual International Conference on Mobile Computing and Networking*, MobiCom '03, pages 66–80, New York, NY, USA, 2003. ACM.

[27] Yan Jin and Jin-Kao Hao. General swap-based multiple neighborhood tabu search for the maximum independent set problem. *Engineering Applications of Artificial Intelligence*, 37:20–33, January 2015.

[28] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.

[29] Kengo Katayama, Akihiro Hamamoto, and Hiroyuki Narihisa. An effective local search for the maximum clique problem. *Information Processing Letters*, 95(5):503–511, September 2005.

[30] Janez Konc and et al. *An improved branch and bound algorithm for the maximum clique problem.* 2007.

[31] C. M. Li, Z. Fang, and K. Xu. Combining MaxSAT Reasoning and Incremental Upper Bound for the Maximum Clique Problem. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, pages 939–946, November 2013.

[32] C. M. Li and Z. Quan. Combining Graph Structure Exploitation and Propositional Reasoning for the Maximum Clique Problem. In *2010 22nd IEEE International Conference on Tools with Artificial Intelligence*, volume 1, pages 344–351, October 2010.

[33] Chu-Min Li and Zhe Quan. An Efficient Branch-and-bound Algorithm Based on MaxSAT for the Maximum Clique Problem. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, AAAI'10, pages 128–133, Atlanta, Georgia, 2010. AAAI Press.

[34] Noël Malod-Dognin, Rumen Andonov, and Nicola Yanev. Maximum Cliques in Protein Structure Comparison. In *Experimental Algorithms*, pages 106–117. Springer, Berlin, Heidelberg, May 2010.

[35] Elena Marchiori. Genetic, Iterated and Multistart Local Search for the Maximum Clique Problem. In *Applications of Evolutionary Computing*, pages 112–121. Springer, Berlin, Heidelberg, April 2002. DOI: 10.1007/3-540-46004-7_12.

[36] Evgeny Maslov, Mikhail Batsyn, and Panos M. Pardalos. Speeding up branch and bound algorithms for solving the maximum clique problem. *Journal of Global Optimization*, 59(1):1–21, May 2014.

[37] Richard Neapolitan. *Foundations Of Algorithms*. Jones Bartlett Learning, 5 edition, 2014.

[38] J. A. Otaibi, Z. Safi, A. Hassaïne, F. Islam, and A. Jaoua. Machine learning and conceptual reasoning for inconsistency detection. *IEEE Access*, 5:338–346, 2017.

[39] Panos M. Pardalos and Jue Xue. The maximum clique problem. *Journal of Global Optimization*, 4(3):301–328, April 1994.

[40] Martín Gómez Ravetti and Pablo Moscato. Identification of a 5-Protein Biomarker Molecular Signature for Predicting Alzheimer's Disease. *PLOS ONE*, 3(9):e3111, September 2008.

[41] Pablo San Segundo, Diego Rodriguez-Losada, and Agustin Jimenez. An exact bit-parallel algorithm for the maximum clique problem. *Computers & Operations Research*, 38(2):571–581, February 2011.

[42] Pablo San Segundo and Cristobal Tapia. Relaxed approximate coloring in exact maximum clique search. *Computers & Operations Research*, 44:185–192, April 2014.

[43] Laura A. Sanchis. Generating hard and diverse test sets for NP-hard graph problems. *Discrete Applied Mathematics*, 58(1):35–66, March 1995.

[44] Pablo San Segundo, Fernando Matia, Diego Rodriguez-Losada, and Miguel Hernando. An improved bit parallel exact maximum clique algorithm. *Optimization Letters*, 7(3):467–479, March 2013.

[45] P.S. Segundo, A. Lopez, M. Batsyn, A. Nikolaev, and P.M. Pardalos. Improved initial vertex ordering for exact maximum clique search. *Applied Intelligence*, 45(3):868–880, 2016.

[46] Alok Singh and Ashok Kumar Gupta. A hybrid heuristic for the maximum clique problem. *Journal of Heuristics*, 12(1-2):5–22, March 2006.

[47] Prem Kumar Singh, Kumar Cherukuri Aswani, and Abdullah Gani. A comprehensive survey on formal concept analysis, its research trends and applications. *International Journal of Applied Mathematics and Computer Science*, 26(2):495–516, 2016.

[48] N. J. A. Sloane. Unsolved problems in graph theory arising from the study of codes. In *in Graph Theory Notes of New York 18*, pages 11–20, 1989.

[49] Patric R. J. Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120(1–3):197–207, August 2002.

[50] Dawn M. Strickland, Earl Barnes, and Joel S. Sokol. Optimal Protein Structure Alignment Using Maximum Cliques. *Operations Research*, 53(3):389–402, June 2005.

[51] E. Tomita, K. Yoshida, T. Hatta, A. Nagao, H. Ito, and M. Wakatsuki. A much faster branch-and-bound algorithm for finding a maximum clique. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9711:215–226, 2016.

[52] Etsuji Tomita and Toshikatsu Kameda. An Efficient Branch-and-bound Algorithm for Finding a Maximum Clique with Computational Experiments. *Journal of Global Optimization*, 37(1):95–111, January 2007.

[53] Etsuji Tomita and Tomokazu Seki. An Efficient Branch-and-Bound Algorithm for Finding a Maximum Clique. In *Discrete Mathematics and Theoretical Computer Science*, pages 278–289. Springer, Berlin, Heidelberg, 2003. DOI: 10.1007/3-540-45066-1_22.

[54] Etsuji Tomita, Yoichi Sutani, Takanori Higashi, Shinya Takahashi, and Mitsuo Wakatsuki. A Simple and Faster Branch-and-Bound Algorithm for Finding a Maximum Clique. In *WALCOM: Algorithms and Computation*, pages 191–203. Springer Berlin Heidelberg, February 2010. DOI: 10.1007/978-3-642-11440-3_18.

[55] Qinghua Wu and Jin-Kao Hao. An adaptive multistart tabu search approach to solve the maximum clique problem. *Journal of Combinatorial Optimization*, 26(1):86–108, July 2013.

[56] Qinghua Wu and Jin-Kao Hao. A review on algorithms for maximum clique problems. *European Journal of Operational Research*, 242(3):693–709, May 2015.

[57] Qingfu Zhang, Jianyong Sun, and E. Tsang. An evolutionary algorithm with guided mutation for the maximum clique problem. *IEEE Transactions on Evolutionary Computation*, 9(2):192–200, April 2005.

Chapter A: Detailed Results

Table A.1: Greedy Method Results - Brock Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)
brock200_1	21	18	0.112	19	0.070	14	0.058	19	0.049	19	0.087
brock200_2	12	8	0.008	9	0.007	8	0.007	9	0.006	9	0.009
brock200_3	15	12	0.011	13	0.008	10	0.010	13	0.010	13	0.009
brock200_4	17	14	0.014	14	0.013	10	0.007	14	0.010	14	0.008
brock400_1	27	22	0.076	22	0.043	16	0.023	22	0.024	22	0.050
brock400_2	29	22	0.047	22	0.041	22	0.030	22	0.028	22	0.040
brock400_3	31	21	0.047	23	0.038	17	0.025	21	0.024	23	0.040
brock400_4	33	22	0.049	22	0.039	17	0.026	22	0.027	22	0.044
brock800_1	23	17	0.186	18	0.106	15	0.090	18	0.206	18	0.107
brock800_2	24	18	0.204	17	0.114	15	0.077	17	0.215	17	0.100
brock800_3	25	17	0.178	17	0.113	14	0.076	17	0.155	17	0.082
brock800_4	26	18	0.183	18	0.095	14	0.071	18	0.128	18	0.090

Table A.2: Greedy Method Results - keller Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)
keller4	11	9	0.005	7	0.002	7	0.001	7	0.003	7	0.004
keller5	27	24	0.131	15	0.052	17	0.039	16	0.093	15	0.093
keller6	59	49	5.929	31	1.403	35	1.020	32	3.542	31	4.169

Table A.3: Greedy Method Results - C Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)
C1000.9	68	57	0.848	58	0.391	48	0.229	59	0.814	58	0.891
C125.9	34	33	0.011	33	0.007	28	0.003	33	0.016	33	0.017
C2000.5	16	12	0.692	13	0.236	12	0.845	13	0.880	13	0.587
C2000.9	80	68	3.573	67	3.205	55	1.164	67	3.522	67	1.406
C250.9	44	40	0.056	39	0.031	36	0.010	39	0.045	39	0.026
C4000.5	18	14	3.152	14	1.541	11	0.948	14	3.451	14	1.043
C500.9	57	50	0.147	51	0.239	45	0.031	51	0.203	51	0.053

Table A.4: Greedy Method Results - c-fat Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)
c-fat200-1	12	12	0.004	12	0.002	12	0.001	12	0.002	12	0.002
c-fat200-2	24	24	0.005	24	0.003	24	0.001	24	0.003	24	0.003
c-fat200-5	58	58	0.017	58	0.007	58	0.008	58	0.009	58	0.010
c-fat500-1	14	14	0.015	14	0.007	14	0.004	14	0.011	14	0.007
c-fat500-10	126	126	0.167	126	0.084	126	0.004	126	0.123	126	0.073
c-fat500-2	26	26	0.026	26	0.012	26	0.004	26	0.017	26	0.016
c-fat500-5	64	64	0.062	64	0.039	64	0.027	64	0.043	64	0.033

Table A.5: Greedy Method Results - DSJC Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)
DSJC1000_5	15	13	0.129	13	0.260	12	0.053	13	0.220	13	0.058
DSJC500_5	13	11	0.036	11	0.084	9	0.011	11	0.058	11	0.013

Table A.6: Greedy Method Results - gen Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)
gen200_p0.9_44	44	37	0.026	37	0.040	35	0.006	37	0.030	37	0.008
gen200_p0.9_55	55	39	0.018	38	0.042	32	0.005	38	0.038	38	0.009
gen400_p0.9_55	55	47	0.081	49	0.160	40	0.023	47	0.125	49	0.047
gen400_p0.9_65	65	45	0.076	45	0.136	40	0.026	45	0.129	45	0.033
gen400_p0.9_75	75	49	0.074	47	0.176	43	0.023	47	0.149	47	0.050

Table A.7: Greedy Method Results - hamming Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)
hamming10-2	512	512	6.392	512	5.682	496	0.524	512	6.731	512	7.028
hamming10-4	40	36	0.488	36	0.190	19	0.072	36	0.273	36	0.385
hamming6-2	32	32	0.003	32	0.001	32	0.000	32	0.002	32	0.003
hamming6-4	4	4	0.001	4	0.000	4	0.000	4	0.000	4	0.001
hamming8-2	128	128	0.105	128	0.036	121	0.008	128	0.085	128	0.117
hamming8-4	16	16	0.009	16	0.003	6	0.002	16	0.009	16	0.010

Table A.8: Greedy Method Results - jhonson Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)
johnson16-2-4	8	8	0.003	8	0.001	8	0.001	8	0.002	8	0.003
johnson32-2-4	16	16	0.041	16	0.016	16	0.014	16	0.028	16	0.048
johnson8-2-4	4	4	0.000	4	0.000	4	0.000	4	0.000	4	0.000
johnson8-4-4	14	14	0.004	14	0.002	8	0.002	14	0.005	14	0.007

Table A.9: Greedy Method Results - MANN Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)
MANN_a27	126	125	0.203	125	0.096	123	0.124	125	0.165	125	0.346
MANN_a45	345	342	3.942	342	2.079	335	1.953	342	4.781	342	4.303
MANN_a81	1100	1096	153.756	1096	93.391	1084	83.404	1096	141.621	1096	156.028
MANN_a9	16	16	0.000	16	0.001	16	0.000	16	0.001	16	0.001

Table A.10: Greedy Method Results - p_hat Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)
p_hat1000-1	10	9	0.035	9	0.082	4	0.095	9	0.064	9	0.095
p_hat1000-2	46	45	0.691	45	0.336	24	0.181	44	0.387	45	0.391
p_hat1000-3	68	62	0.665	61	0.911	34	0.356	61	0.821	61	0.930
p_hat1500-1	12	10	0.213	10	0.354	6	0.190	10	0.337	10	0.278
p_hat1500-2	65	60	0.704	61	1.714	26	0.493	61	1.414	61	1.100
p_hat1500-3	94	86	1.680	86	1.749	44	0.494	86	1.441	86	1.208
p_hat300-1	8	7	0.009	7	0.009	4	0.004	7	0.006	7	0.007
p_hat300-2	25	24	0.026	25	0.027	14	0.006	25	0.038	25	0.022
p_hat300-3	36	33	0.036	34	0.033	24	0.021	34	0.073	34	0.029
p_hat500-1	9	8	0.016	8	0.017	5	0.016	9	0.029	8	0.016
p_hat500-2	36	34	0.070	32	0.074	22	0.028	32	0.129	32	0.079
p_hat500-3	50	46	0.115	46	0.112	26	0.045	45	0.202	46	0.094
p_hat700-1	11	8	0.035	8	0.035	7	0.029	8	0.053	8	0.037
p_hat700-2	44	43	0.154	43	0.150	23	0.044	43	0.228	43	0.164
p_hat700-3	62	58	0.229	59	0.202	42	0.095	59	0.373	59	0.187

Table A.11: Greedy Method Results - san - sanr Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)
san1000	15	9	0.087	9	0.075	8	0.086	9	0.157	9	0.078
san200_0.7_1	30	16	0.006	17	0.005	16	0.005	17	0.012	17	0.008
san200_0.7_2	18	14	0.008	14	0.005	12	0.005	14	0.015	14	0.006
san200_0.9_1	70	47	0.021	47	0.015	34	0.009	48	0.042	47	0.020
san200_0.9_2	60	38	0.017	41	0.013	27	0.009	39	0.028	41	0.017
san200_0.9_3	44	34	0.010	34	0.011	25	0.008	34	0.019	34	0.011
san400_0.5_1	13	9	0.013	9	0.010	7	0.014	9	0.022	9	0.011
san400_0.7_1	40	21	0.024	21	0.018	21	0.022	21	0.039	21	0.029
san400_0.7_2	30	18	0.022	17	0.020	16	0.020	17	0.038	17	0.022
san400_0.7_3	22	16	0.024	16	0.021	12	0.023	16	0.036	16	0.025
san400_0.9_1	100	80	0.084	82	0.066	76	0.025	82	0.120	82	0.079
sanr200_0.7	18	16	0.005	15	0.004	13	0.003	15	0.008	15	0.006
sanr200_0.9	42	41	0.017	40	0.008	35	0.006	41	0.026	40	0.015
sanr400_0.5	13	11	0.015	12	0.009	8	0.008	12	0.022	12	0.014
sanr400_0.7	21	18	0.023	18	0.014	16	0.013	18	0.030	18	0.025

Table A.12: Backtracking Method Results - Brock Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)
brock200_1	21	20	0.95	20	0.68	19	0.42	20	0.35	20	0.48
brock200_2	12	11	0.46	11	0.32	10	0.21	11	0.16	11	0.17
brock200_3	15	14	0.57	14	0.38	12	0.23	14	0.19	14	0.21
brock200_4	17	16	0.62	16	0.45	14	0.24	16	0.22	16	0.26
brock400_1	27	23	3.16	23	3.61	20	1.33	23	1.15	23	1.25
brock400_2	29	24	3.29	24	3.59	22	1.21	24	1.22	24	1.22
brock400_3	31	24	3.41	24	3.11	20	1.03	24	1.09	24	1.29
brock400_4	33	24	3.22	24	2.09	20	0.94	24	1.12	24	1.25
brock800_1	23	20	9.6	20	4.79	18	2.53	20	3.36	20	3.91
brock800_2	24	20	8	19	3.16	17	2.27	20	3.7	19	3.8
brock800_3	25	20	4.4	20	4.66	18	5.71	20	3.74	20	3.56
brock800_4	26	20	2.88	20	5.05	17	2.91	20	3.86	20	3.52

Table A.13: Backtracking Method Results - keller Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)
keller4	11	11	0.12	11	0.11	9	0.07	11	0.12	11	0.12
keller5	27	26	4.19	21	3.42	19	2.3	20	3.38	20	3.41
keller6	59	52	124.8	43	96.15	35	60.99	43	96.41	43	99.26

Table A.14: Backtracking Method Results - C Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)
C1000.9	68	61	14.19	63	16.22	57	9.82	62	15.16	63	15.53
C125.9	34	33	0.24	34	0.22	32	0.11	34	0.23	34	0.23
C2000.5	16	15	15.85	15	17.04	13	13.63	15	14.61	15	15.19
C2000.9	80	69	60.49	70	64.41	61	44.31	70	64.06	70	67.33
C250.9	44	43	0.77	44	0.89	40	0.46	44	0.96	44	0.97
C4000.5	18	15	63.55	16	64.62	14	54.09	16	60.62	16	61.88
C500.9	57	54	3.48	53	3.92	48	2.35	53	3.63	53	3.94

Table A.15: Backtracking Method Results - c-fat Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)
c-fat200-1	12	12	0.04	12	0.05	12	0.03	12	0.06	12	0.05
c-fat200-2	24	24	0.07	24	0.09	24	0.05	24	0.12	24	0.09
c-fat200-5	58	58	0.2	58	0.25	58	0.26	58	0.38	58	0.28
c-fat500-1	14	14	0.24	14	0.3	14	0.16	14	0.34	14	0.32
c-fat500-10	126	126	4.87	126	3.04	126	0.16	126	3.61	126	3.05
c-fat500-2	26	26	1.03	26	0.47	26	0.17	26	0.51	26	0.48
c-fat500-5	64	64	2.09	64	1.15	64	1.2	64	1.22	64	1.16

Table A.16: Backtracking Method Results - DSJC Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)
DSJC1000_5	15	14	3.76	13	3.77	12	3.42	13	3.68	13	3.93
DSJC500_5	13	13	0.89	13	0.9	11	0.8	13	0.88	13	0.93

Table A.17: Backtracking Method Results - gen Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)
gen200_p0.9_44	44	39	0.53	39	0.59	37	0.32	39	0.59	39	0.59
gen200_p0.9_55	55	41	0.55	44	0.57	55	0.3	44	0.6	44	0.61
gen400_p0.9_55	55	49	2.33	49	2.41	43	1.46	49	2.39	49	2.64
gen400_p0.9_65	65	49	2.15	49	2.44	53	1.36	49	2.36	49	2.75
gen400_p0.9_75	75	49	2.36	50	2.51	49	1.52	49	2.45	50	3.09

Table A.18: Backtracking Method Results - hamming Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)
hamming10-2	512	512	190.79	512	192.41	503	30.53	512	194.55	512	199.14
hamming10-4	40	36	9.59	36	9.71	19	4.65	36	8.98	36	9.18
hamming6-2	32	32	0.05	32	0.05	32	0.02	32	0.06	32	0.05
hamming6-4	4	4	0.01	4	0.01	4	0.01	4	0.01	4	0.01
hamming8-2	128	128	3.14	128	2.82	128	0.66	128	2.92	128	2.87
hamming8-4	16	16	0.25	16	0.22	6	0.13	16	0.22	16	0.23

Table A.19: Backtracking Method Results - jhonson Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)
johnson16-2-4	8	8	0.05	8	0.05	8	0.05	8	0.05	8	0.05
johnson32-2-4	16	16	1.22	16	1.1	16	1.11	16	1.17	16	1.2
johnson8-2-4	4	4	0	4	0	4	0	4	0	4	0
johnson8-4-4	14	14	0.03	14	0.03	8	0.02	14	0.03	14	0.03

Table A.20: Backtracking Method Results - MANN Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)
MANN_a27	126	126	6.01	125	6.96	123	5.8	125	6.72	125	6.32
MANN_a45	345	343	126.91	344	125.43	335	117.6	344	121.96	344	123.13
MANN_a81	1100	1098	3575.39	1096	3531	1084	3331.46	1096	3481.55	1096	3605.78
MANN_a9	16	16	0.01	16	0.01	16	0.01	16	0.01	16	0.03

Table A.21: Backtracking Method Results - p_hat Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)
p_hat1000-1	10	10	1.95	10	1.92	8	1.57	10	2	10	3.87
p_hat1000-2	46	45	8.66	45	8.66	30	2.81	45	8.59	45	9.84
p_hat1000-3	68	64	13.45	65	14.41	45	5.62	65	13.16	65	25.66
p_hat1500-1	12	11	4.55	11	4.86	9	3.6	11	4.4	11	4.98
p_hat1500-2	65	64	27.98	64	24.77	33	6.97	64	24.14	64	39.74
p_hat1500-3	94	91	59.98	90	37.01	64	12.64	90	36.23	90	100.42
p_hat300-1	8	8	0.6	8	0.16	7	0.14	8	0.16	8	0.47
p_hat300-2	25	25	1.99	25	0.55	18	0.23	25	0.56	25	1.53
p_hat300-3	36	34	2.06	34	0.9	27	0.39	34	0.81	34	2.17
p_hat500-1	9	9	0.87	9	0.48	8	0.36	9	0.47	9	1.75
p_hat500-2	36	36	2.61	36	1.89	25	0.63	36	1.82	36	6.68
p_hat500-3	50	48	3.33	48	2.88	35	1.17	48	2.91	48	9.41
p_hat700-1	11	9	0.9	9	0.95	8	0.68	9	0.95	9	2.15
p_hat700-2	44	44	3.8	43	4.2	26	1.36	43	4.1	43	9.64
p_hat700-3	62	60	15.1	62	13.71	43	2.44	62	6.55	62	22.4

Table A.22: Backtracking Method Results - san - sanr Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)
san1000	15	10	3.52	10	7.88	9	2.94	10	2.9	10	8.06
san200_0.7_1	30	30	0.2	30	0.39	16	0.18	30	0.23	30	0.78
san200_0.7_2	18	17	0.19	15	0.36	14	0.18	15	0.22	15	0.77
san200_0.9_1	70	68	0.52	70	1.09	55	0.33	70	0.68	70	2.22
san200_0.9_2	60	54	0.43	55	0.87	54	0.29	55	0.48	55	1.73
san200_0.9_3	44	35	0.36	36	0.67	32	0.29	36	0.43	36	1.52
san400_0.5_1	13	13	0.37	13	0.62	8	0.41	13	0.43	13	1.52
san400_0.7_1	40	40	0.87	40	1.11	21	0.81	40	0.99	40	3.44
san400_0.7_2	30	30	0.78	19	0.88	17	0.76	19	0.87	19	3.03
san400_0.7_3	22	17	0.72	17	0.85	14	0.66	17	0.81	17	2.81
san400_0.9_1	100	100	1.85	100	2.35	76	1.25	100	2.29	100	7.84
sanr200_0.7	18	18	0.19	18	0.22	15	0.16	18	0.19	18	0.71
sanr200_0.9	42	41	0.39	41	0.55	37	0.3	41	0.52	41	1.74
sanr400_0.5	13	12	0.45	12	0.53	10	0.45	12	0.51	12	1.8
sanr400_0.7	21	19	0.72	20	0.83	18	0.64	20	0.8	20	2.82

Table A.23: Branch and Bound Method Results - Brock Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)
brock200_1	21	21	22.3(1.2)	21	42.1(3.2)	20	96.8(5.6)	21	42.5(7.2)	21	39.4(8.6)
brock200_2	12	12	10.1(0.0)	12	21.0(7.8)	12	46.8(44.3)	12	4.1(4.1)	12	5.6(5.5)
brock200_3	15	15	13.1(9.1)	14	25.0(0.2)	15	67.2(0.7)	14	14.7(0.8)	14	15.2(0.9)
brock200_4	17	17	16.7(0.8)	17	30.7(0.4)	17	78.9(17.1)	17	25.0(0.5)	17	15.5(0.5)
brock400_1	27	25	68.5(6.8)	25	121.1(12.8)	23	254.3(92.5)	24	21.2(2.2)	24	19.1(2.2)
brock400_2	29	24	67.6(1.1)	25	128.4(51.2)	22	267.5(0.7)	24	29.2(3.7)	24	33.8(3.6)
brock400_3	31	24	71.3(4.4)	25	124.0(2.8)	22	259.5(6.8)	25	38.2(16.9)	25	39.5(16.6)
brock400_4	33	25	68.2(27.2)	25	131.6(23.0)	33	259.5(84.3)	24	35.9(8.2)	24	27.3(8.1)
brock800_1	23	21	164.6(40.5)	21	255.8(100.5)	19	487.6(108.2)	20	57.0(48.2)	20	64.1(52.2)
brock800_2	24	21	159.0(131.3)	20	262.0(2.1)	19	493.0(274.3)	20	71.8(49.7)	19	69.1(1.8)
brock800_3	25	21	156.3(6.9)	21	246.4(4.1)	19	487.9(12.4)	20	72.6(3.9)	20	76.1(3.8)
brock800_4	26	21	171.4(49.4)	21	252.8(166.2)	19	503.0(3.4)	20	61.7(35.7)	20	57.3(29.9)

The time at which the clique is found is added between parenthesis in the tables of the branch and bound method to the time column.

Table A.24: Branch and Bound Method Results - keller Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)	clique size	time(s)
keller4	11	11	13.1(0.2)	11	20.7(1.0)	9	39.0(0.1)	11	14.9(1.7)	11	10.7(2.1)
keller5	27	27	284.3(1.7)	27	378.4(4.8)	19	791.8(2.3)	24	36.8(0.6)	24	31.3(0.6)
keller6	59	54	2000.0(705.5)	53	2000.0(7.9)	37	2000.0(33.5)	49	535.6(21.6)	51	548.2(47.9)

Table A.25: Branch and Bound Method Results - C Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		size	time(s)	size	time(s)	size	time(s)	size	time(s)	size	time(s)
C1000.9	68	64	707.0(289.8)	65	1220.4(201.3)	58	1989.7(218.6)	64	453.0(48.4)	64	429.9(38.2)
C125.9	34	34	25.4(1.2)	34	45.7(0.0)	34	72.6(7.6)	34	57.1(0.1)	34	56.4(0.1)
C2000.5	16	16	514.4(163.2)	16	813.3(24.2)	14	1145.6(22.8)	15	315.2(77.5)	15	321.6(84.2)
C2000.9	80	72	2000.0(462.5)	73	2000.0(244.0)	64	2000.0(1192.3)	72	398.9(9.7)	72	375.0(9.4)
C250.9	44	44	72.5(1.7)	44	150.3(6.7)	42	365.2(156.4)	44	135.0(0.9)	44	116.7(0.6)
C4000.5	18	17	1920.2(71.4)	17	2005.8(179.1)	15	2008.1(85.9)	16	1129.6(239.4)	16	1110.4(234.9)
C500.9	57	56	216.4(137.4)	56	350.8(42.4)	51	787.3(41.7)	55	379.3(14.5)	56	418.4(95.1)

Table A.26: Branch and Bound Method Results - c-fat Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		size	time(s)	size	time(s)	size	time(s)	size	time(s)	size	time(s)
c-fat200-1	12	12	0.7(0.0)	12	0.7(0.0)	12	0.9(0.0)	12	0.4(0.0)	12	1.1(0.1)
c-fat200-2	24	24	2.8(2.4)	24	3.3(3.0)	24	4.3(4.0)	24	1.0(0.8)	24	1.8(1.5)
c-fat200-5	58	58	102.7(0.0)	58	128.0(0.0)	58	91.3(0.0)	58	9.9(0.0)	58	7.1(0.0)
c-fat500-1	14	14	3.6(0.1)	14	3.8(0.1)	14	6.1(1.5)	14	2.6(0.2)	14	2.0(0.1)
c-fat500-10	126	126	2000.1(0.1)	126	2000.0(0.1)	126	196.2(0.1)	126	77.9(0.2)	126	51.5(0.2)
c-fat500-2	26	26	40.0(0.1)	26	52.9(0.1)	26	25.9(2.7)	26	6.7(0.1)	26	9.6(0.1)
c-fat500-5	64	64	416.8(0.1)	64	641.7(0.1)	64	388.4(29.5)	64	19.4(0.1)	64	26.1(0.1)

Table A.27: Branch and Bound Method Results - DSJC Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		size	time(s)	size	time(s)	size	time(s)	size	time(s)	size	time(s)
DSJC1000_5	15	15	144.6(113.3)	15	266.5(221.9)	13	611.7(17.1)	14	95.6(32.4)	14	67.1(34.5)
DSJC500_5	13	13	43.8(0.6)	13	87.2(0.6)	12	240.4(0.4)	13	13.5(7.6)	13	29.4(17.4)

Table A.28: Branch and Bound Method Results - gen Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		size	time(s)	size	time(s)	size	time(s)	size	time(s)	size	time(s)
gen200_p0.9_44	44	40	45.0(2.6)	40	86.3(3.8)	43	241.9(13.8)	40	113.0(3.6)	40	112.6(0.3)
gen200_p0.9_55	55	55	46.9(1.3)	55	101.8(4.1)	55	230.1(6.3)	55	123.4(15.3)	55	130.7(29.8)
gen400_p0.9_55	55	51	125.6(99.0)	51	249.3(35.1)	50	738.1(270.5)	51	256.5(0.6)	51	283.1(67.7)
gen400_p0.9_65	65	53	129.8(91.2)	54	253.9(9.6)	59	812.1(227.0)	55	314.5(77.6)	54	268.8(31.5)
gen400_p0.9_75	75	53	131.7(92.4)	54	254.1(123.0)	65	815.9(218.1)	54	288.9(167.6)	54	320.1(197.1)

Table A.29: Branch and Bound Method Results - hamming Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		size	time(s)	size	time(s)	size	time(s)	size	time(s)	size	time(s)
hamming10-2	512	512	2000.0(2.5)	512	2000.0(3.3)	503	2000.0(1706.6)	512	2000.0(2.6)	512	2000.0(9.4)
hamming10-4	40	36	2000.0(0.6)	36	2000.0(0.7)	19	2000.0(353.6)	36	159.9(0.9)	36	174.3(1.3)
hamming6-2	32	32	3.5(0.0)	32	4.5(0.0)	32	4.4(3.2)	32	5.1(0.0)	32	3.9(0.0)
hamming6-4	4	4	0.2(0.0)	4	0.2(0.0)	4	0.3(0.0)	4	0.2(0.0)	4	0.1(0.0)
hamming8-2	128	128	183.9(0.2)	128	326.3(0.1)	128	646.1(129.0)	128	499.2(0.2)	128	502.9(0.1)
hamming8-4	16	16	112.7(0.0)	16	192.4(0.1)	6	187.0(0.1)	16	32.1(0.1)	16	30.2(0.1)

Table A.30: Branch and Bound Method Results - jhonson Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		size	time(s)	size	time(s)	size	time(s)	size	time(s)	size	time(s)
johnson16-2-4	8	8	31.8(0.0)	8	57.6(0.0)	8	83.1(0.0)	8	42.5(0.0)	8	38.1(0.0)
johnson32-2-4	16	16	293.1(0.1)	16	492.5(0.2)	16	850.4(0.2)	16	165.5(0.2)	16	170.8(0.2)
johnson8-2-4	4	4	0.0(0.0)	4	0.0(0.0)	4	0.0(0.0)	4	0.0(0.0)	4	0.0(0.0)
johnson8-4-4	14	14	2.1(0.0)	14	2.6(0.0)	14	4.1(0.0)	14	1.8(0.0)	14	2.5(0.0)

Table A.31: Branch and Bound Method Results - MANN Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		size	time(s)	size	time(s)	size	time(s)	size	time(s)	size	time(s)
MANN_a27	126	126	414.8(0.8)	125	421.8(0.2)	124	2000.0(309.7)	125	573.9(0.2)	125	586.3(0.2)
MANN_a45	345	344	2000.0(1733.0)	344	2000.0(93.0)	126	2000.0(105.8)	344	2000.0(105.8)	344	2000.0(112.9)
MANN_a81	1100	1098	2000.1(1061.0)	1096	2000.0(71.8)	1013	2000.1(63.2)	1096	2000.2(59.4)	1096	2000.1(63.2)
MANN_a9	16	16	0.7(0.0)	16	1.0(0.0)	16	0.6(0.0)	16	1.3(0.0)	16	0.6(0.0)

Table A.32: Branch and Bound Method Results - p_hat Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		size	time(s)	size	time(s)	size	time(s)	size	time(s)	size	time(s)
p_hat1000-1	10	10	68.0(0.8)	10	110.6(1.1)	10	242.1(163.9)	10	40.6(3.2)	10	42.2(1.7)
p_hat1000-2	46	46	417.8(48.0)	46	600.3(24.6)	36	626.2(248.4)	46	97.9(8.3)	46	108.7(17.2)
p_hat1000-3	68	65	712.4(46.6)	67	1064.8(940.0)	51	1390.6(675.4)	65	124.0(60.0)	65	113.6(65.1)
p_hat1500-1	12	11	149.0(1.5)	11	226.5(1.8)	10	380.4(3.4)	11	80.9(18.5)	11	66.7(15.0)
p_hat1500-2	65	65	1175.4(341.1)	65	1672.0(60.8)	43	1327.0(214.3)	64	204.9(116.7)	64	229.1(107.3)
p_hat1500-3	94	93	1755.6(89.5)	93	2000.0(136.1)	69	2000.0(759.4)	92	279.9(103.0)	91	249.1(6.1)
p_hat300-1	8	8	16.7(0.1)	8	19.9(0.7)	8	33.2(19.3)	8	3.4(0.2)	8	2.3(0.1)
p_hat300-2	25	25	45.9(8.3)	25	62.0(2.0)	24	115.5(13.4)	25	7.2(1.1)	25	14.1(1.0)
p_hat300-3	36	36	67.7(25.9)	36	95.2(1.4)	33	228.6(22.4)	36	48.8(5.5)	36	52.1(3.6)
p_hat500-1	9	9	22.9(0.5)	9	36.9(0.2)	9	102.2(31.5)	9	6.8(0.8)	9	13.2(1.9)
p_hat500-2	36	36	116.9(5.0)	36	177.8(0.5)	30	289.8(78.3)	36	17.8(3.7)	36	18.6(3.7)
p_hat500-3	50	50	194.5(90.0)	50	296.3(15.6)	43	502.2(93.1)	50	91.6(25.1)	50	70.1(18.9)
p_hat700-1	11	11	46.6(8.3)	11	65.8(9.2)	9	161.7(1.4)	10	14.3(1.4)	10	19.6(2.7)
p_hat700-2	44	44	266.1(5.0)	44	358.6(4.7)	34	438.1(151.5)	44	42.8(1.3)	44	41.9(4.0)
p_hat700-3	62	62	434.9(4.5)	62	555.4(32.1)	49	792.5(52.4)	62	74.2(6.8)	62	76.0(7.0)

Table A.33: Branch and Bound Method Results - san - sanr Family

graph	w	weight1		weight2		weight3		weight4		weight5	
		size	time(s)	size	time(s)	size	time(s)	size	time(s)	size	time(s)
san1000	15	10	235.0(1.3)	11	343.5(163.1)	10	481.2(1.4)	10	55.1(1.8)	10	53.9(1.6)
san200_0.7_1	30	30	22.6(5.1)	30	30.4(2.9)	30	127.5(0.4)	30	27.1(0.1)	30	18.6(0.1)
san200_0.7_2	18	18	21.3(1.9)	18	32.0(1.0)	15	92.0(0.5)	18	23.0(3.4)	18	24.0(3.1)
san200_0.9_1	70	70	90.2(0.1)	70	106.5(1.1)	70	388.4(16.6)	70	124.7(0.1)	70	134.3(0.0)
san200_0.9_2	60	60	63.8(5.1)	60	83.3(9.6)	60	324.1(12.2)	60	123.9(0.1)	60	107.3(0.1)
san200_0.9_3	44	44	58.5(55.4)	44	68.1(5.5)	41	205.2(6.4)	44	87.8(4.1)	44	89.8(3.7)
san400_0.5_1	13	13	50.8(30.5)	13	61.0(1.2)	9	135.2(0.7)	13	14.6(1.2)	13	10.8(0.8)
san400_0.7_1	40	40	120.1(19.1)	40	136.0(3.6)	22	415.8(1.7)	40	15.0(0.3)	40	12.1(0.2)
san400_0.7_2	30	30	82.8(5.7)	30	91.4(8.9)	18	274.3(33.5)	30	14.0(5.9)	30	11.7(4.7)
san400_0.7_3	22	22	66.3(2.9)	22	78.6(2.0)	15	170.9(1.0)	17	13.2(0.9)	17	11.9(0.7)
san400_0.9_1	100	100	273.3(6.5)	100	356.9(4.8)	87	1470.7(200.5)	100	344.2(10.4)	100	359.9(10.3)
sanr200_0.7	18	18	26.3(0.4)	18	32.2(2.2)	17	105.3(0.3)	18	18.7(0.3)	18	20.5(0.6)
sanr200_0.9	42	42	67.3(1.6)	42	82.8(3.8)	41	221.7(137.6)	42	103.4(0.7)	42	128.5(0.7)
sanr400_0.5	13	13	37.1(2.2)	13	57.1(5.8)	13	139.1(66.9)	12	10.2(0.3)	12	14.5(0.6)
sanr400_0.7	21	21	60.4(9.2)	21	98.3(0.2)	20	266.9(146.6)	21	18.1(0.4)	21	11.7(0.5)