University of Bergen

Department of Informatics

Algorithms

Faster enumeration of minimal connected dominating sets in split graphs

Student:
Ida Bredal Skjørten

Supervisor:
Professor Pinar Heggernes

Master Thesis

June 2017

Acknowledgements

First of all I like to thank my supervisor Pinar Heggernes. Throughout the work with this thesis she has been a great help, motivation and inspiration. I would also like to thank my fellow master students and the whole algorithms group for motivating me. Last, but not least I would like to thank my family, especially my mom, for their help and support.

Contents

Chapter 1

Introduction

Graphs are mathematical objects that can be used to model many real world problems. An example is a roadmap, where the nodes in the graph represent cities and the edges of the graph represent roads. An interesting and important task is to find certain objects or node subsets with a specified property in a graph, and in this thesis our main focus will be on finding dominating sets in graphs.

An everyday example of a dominating set problem can be that we are asked to place hospitals in a city. Every district of the city has to either have its own hospital or have a direct road to a district with a hospital. This problem can be modeled as a graph where the nodes of the graph represent districts in the city and the edges of the graph represent roads between the districts. A valid solution to this problem can be to place a hospital in every district. However, it is expensive to build and operate hospitals, so maybe it would be more economically feasible if we are asked to build as few hospitals as possible. This corresponds to finding a minimum dominating set in the graph.

Suppose that we are not just seeking to find one minimum dominating set in a graph. Instead we are interested in how many dominating sets we can find in a graph so that if we remove any node from a solution, this solution would no longer be a dominating set. This corresponds exactly to finding the number of minimal dominating sets in a graph.

Important questions in both algorithms and combinatorics concern how many objects of a certain type there can there be in a graph. It is exactly this type of question this thesis will study. In many applications the best known way to find a minimum object of a certain type can be to list all the minimal objects and then pick the smallest one among these mini-

mal objects. Thus, knowing the number of such objects and being able to list them in reasonable time might help us get faster algorithms for hard problems.

1.1 Notation and Definitions

In this section we will define some of the terms that are used in this thesis, and explain the notation that we will use.

In this thesis we will look at simple undirected graphs, and we will use $G = (V, E)$ to denote such a graph, where V is the set of nodes in G, and E is the set of edges in G. When the node set or edge set of a graph is not specified will use $V(G)$ to denote the nodes of a graph G, and $E(G)$ to denote the edges of a graph G. Two nodes in a graph are considered to be neighbors if there is an edge between them. We will use $N_G(x)$ to denote the neighbors of a node x in the graph G. A node is adjacent to all of its neighbors. A graph G induced by a subgraph S is denoted as the induced subgraph $G[S]$, and is defined as the graph consisting of all the nodes in S and it contains all the edges in G that connect a pair of nodes in S.

A *clique* is a set C of nodes such that every node in C is adjacent to all other nodes in C.

An *independent set* I is a set of nodes such that none of the nodes in I are adjacent to any of the other nodes in I.

1.2 Minimal dominating and connected dominating sets

A *dominating set* of a graph G is a subset D of the nodes in G such that every node of G is either in D or has a neighbor in D. In Figure 1.1 the nodes marked as red in G_1, G_2, and G_3 form dominating sets, because every node in these graphs is either red, or is adjacent to a red node.

We say that a node u is dominated by a node v in a dominating set D, if u and v are connected by an edge, and $v \in D$. In a dominating set D all $d \in D$ are dominated by themselves. For a dominating set D we define a *private node* of some node $d \in D$ to be a node x that is dominated only by d. It is important to note that a node can be its own private node.

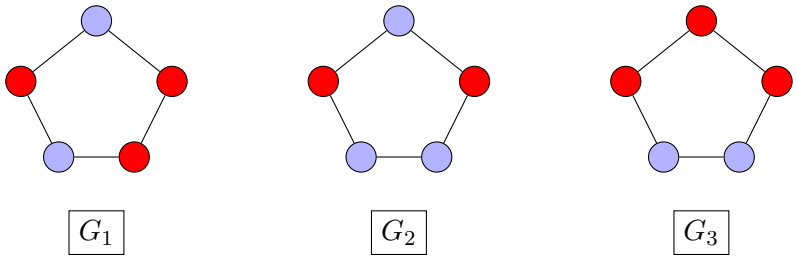

Figure 1.1: Graphs where the red nodes in G_1 form a dominating set, the red nodes in G_2 form a minimal dominating set, and the red nodes in G_3 form a minimal connected dominating set.

A dominating set D is *minimal* if no proper subset of D is a dominating set. That is, D is minimal if every node of D has a private node in G. In Figure 1.1 the nodes marked as red in G_2 form a minimal dominating set, because every red node in G_2 has a private node. The nodes marked as red in G_1 and G_3 do not form minimal dominating sets, because some of the red nodes in G_1 and G_3 do not have any private nodes. In other words the nodes marked as red in G_1 and G_3 do not form minimal dominating sets because in both of these graphs some of the red nodes can be removed from the dominating set (in G_3 the upper red node is not needed to dominate the graph, and in G_1 either of the bottom right or the middle right is not needed to dominate the graph), by changing color to light blue, and the set consisting of the nodes that are red would still from a dominating set.

A *connected dominating set* (cds) of a graph G is a dominating set D, where $G[D]$ is connected.

A *minimal connected dominating set* (mcds)[1] of a graph G is a connected dominating set D such that no subset of D is a connected dominating set. In Figure 1.1 the red nodes in G_3 form a minimal connected dominating set. Because no subset of the red nodes is a connected dominating set, the red nodes form a dominating set and they are connected. The red nodes in G_1 and G_2 do not form minimal connected dominating sets, because the set consisting of the red nodes in G_1 is not connected or minimal, and the set consisting of the red nodes in G_2 is not connected.

A practical example comes from ad-hoc wireless networks [17]. The nodes of the graph represent cellphones or wireless units. There is an edge between two wireless units if they are close to each other in euclidian distance. The goal is to create a set of wireless units that work as providers to this network,

[1]We use the same abbreviation mcds also for the plural form; minimal connected dominating sets.

so that all communication in the network go through the providers. We want the set of providers to be as small as possible, but at the same time we need all the providers to be close to each other. If all communication have to go though the providers we know that every wireless unit has to be close to a provider. One measure of the size of a solution can be that we say that a solution is as small as possible if the solution is invalid if any of the providers are removed from the solution. The valid solutions to this routing problem are exactly the mcds of the graph, where the wireless units in a mcds of the graph form a set of providers.

The following lemma shows that checking whether a connected dominating set is minimal can be easily done.

Lemma 1.1. *Let D be a connected dominating set. Then D is minimal if and only if $\forall d \in D$, $D \setminus \{d\}$ is not a connected dominating set.*

Proof. For the first direction it follows from the definition that if D is minimal then $D \setminus \{d\}$ is not a cds $\forall d \in D$. For the other direction, assume for contradiction that D is a cds, but not minimal and that $\forall d \in D$, $D \setminus \{d\}$ is not a cds, meaning that we cannot remove a single node, and still have a connected dominating set. Under the assumption that D is not minimal there must exist a $S \subset D$ such that $D \setminus S$ is a cds, and $\forall s \in S$, $D \setminus \{s\}$ is not a cds. There are two possible reasons for why we cannot remove a node, either it is needed to connect the cds, or it has a private node in the graph (it is needed to dominate the graph). If $\exists s \in S$ that has a private node $z \in G$, then z will not have a neighbor in $D \setminus S$ so $D \setminus S$ cannot be a dominating set. Therefore, none of the nodes in S can have a private node, but then every node in S is necessary to connect D. If that is the case then $\forall s \in S$, $D \setminus \{s\}$ is not connected, but D and $D \setminus S$ is connected. This means that at least one of the nodes in S is needed to connect $X \subseteq S$ to D, but if that is the case then there would $\exists x \in X$ so that $D \setminus \{x\}$ is a cds, but this is a contradiction to our assumption so $D \setminus S$ cannot be a cds. □

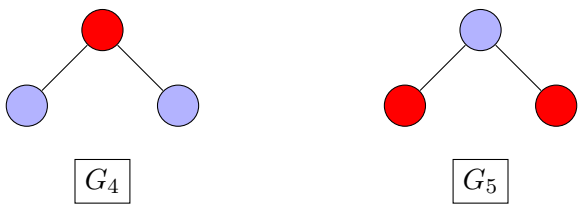

Figure 1.2: Graph where the red nodes in both G_4 and G_5 form a minimal dominating set, but only the red node in G_4 is a minimum dominating set.

A *minimum dominating set* in a graph is a dominating set with the smallest cardinality. A natural assumption might be to assume that every minimal dominating set is also a minimum dominating set, but this is not always the case. In Figure 1.2 we can see that the red nodes in both G_4 and G_5 form minimal connected dominating sets. The red node in G_4 has three private nodes, and each of the red nodes in G_5 has itself as a private node. Clearly the red nodes in G_5 do not form a minimum dominating set, since this dominating set consists of 2 nodes, whereas the red node in G_4 forms a dominating set of size 1.

1.3 Enumeration

Various types of problems can be defined related to a node set satisfying some property in a graph. Let us consider connected dominating set as an example. We can ask whether the graph has a connected dominating set of size at most a given integer. This is an example of a decision problem. We can also try to find a connected dominating set of smallest size. This is an example of an optimization problem. Finally, since the smallest connected dominating set must be among the minimal connected dominating sets we can ask to list all distinct minimal connected dominating sets in a graph. This is an example of an enumeration problem.

An enumeration algorithm is an algorithm that lists all objects of a certain type in a given input graph. Enumeration algorithms can be used to solve optimization problems by simply listing or enumerating all the solutions and picking the best one. For many problems there exist algorithms that solve the problem faster than an enumeration algorithm, but for some problems the current best known solution to an optimization problem is by an enumeration algorithm. For example, for the problem of finding a minimum subset feedback vertex set on chordal graphs, the best known algorithm enumerates all minimal subset feedback vertex sets and picks the smallest one [12]. Enumeration algorithms can also be used to solve decision problems. We can do this by checking the size of each minimal set that the enumeration algorithm lists, and check if any of these are smaller than the threshold given as input to the decision problem.

Extremal graph theory is a field that contains problems that ask for the maximum or minimum number of objects in a graph [16]. A common way to approach these problem is to try to bound this number with algorithmic methods.

An upper bound is simply a bound that shows that there cannot be any more

objects than the given bound. An upper bound that we are able to prove can be higher than the actual maximum number of objects that we are looking for. Upper bounds are very often obtained by branching algorithms, and cannot be less than the number of enumerated objects, and hence bounds their number. We will address branching algorithms in Chapter 3.

A lower bound is the maximum number of objects that we can verify by examples. It is the largest number of objets in any graph for which we can prove an exact maximum number.

In general, our aim is to narrow or close the gap between the upper and lower bounds. If the lower and upper bounds are equal we have a tight or closed bound, and we know exactly the maximum number of the specified object that can occur in any graph. If we do not have a closed bound we know that either the upper bound is too high, and we need to find a faster algorithm that lists or enumerates these objects in a given graph, or the lower bound is too low, and we need to find a graph example that contains more of these objects then the current lower bound.

1.4 The classes of P and NP

Let us formally define the decision version of finding minimum dominating sets and minimum cds:

DOMINATING SET
Input: A graph G, and an integer k
Task: Determine whether G has a dominating set of size $\leq k$

CONNECTED DOMINATING SET
Input: A graph G, and an integer k
Task: Determine whether G has a connected dominating set of size $\leq k$

DOMINATING SET and CONNECTED DOMINATING SET are two of the classical NP-complete problems. In theoretical computer science there has been a lot of focus on NP-complete problems. We will now give a brief introduction to the classes P, NP, and NP-complete problems. This is a prerequisite to understand the complexity of the DOMINATING SET problem. We will also encounter these classes of problems when we discuss the graph isomorphism problem in Chapter 4.

For a lot of problems, like sorting a list of numbers, there exits algorithms that solve the problems in polynomial time in the size of the input. All such problems, where we have algorithms that solve the problem in poly-

nomial time, are in the class called P. There are many problems for which we have not been able to find a polynomial time algorithm that solves the problem. Some of these problems, however, are well-behaved in the sense that if somebody provides a suggestion of a solution, an algorithm can check in polynomial time whether it is indeed a solution or not. These problems belong to the class called NP.

The fact that no one has been able to find polynomial time algorithms for these problems does not mean that there does not exist such algorithms for these problems. In fact no one has been able to prove that there does not exist any polynomial time algorithms for the problems in NP. This means that we do not know if the problems in NP are in fact members of P, which would imply that P=NP. What we do know is that P$\subseteq$NP. The problem of whether P=NP or P$\subset$NP is one of the major open questions in theoretical computer science. However, it is widely believed that P$\neq$NP, and therefore there is a lot of focus on finding fast exponential algorithms for the problems in NP.

The class of NP-complete problems consists of the hardest problems in NP. All problems in NP can be reduced in polynomial time to each of the NP-complete problems. This means that if someone finds a polynomial time algorithm to any of the NP-complete problems, we would have a polynomial time algorithm for all problems in NP, which would imply that P=NP. For the problems in NP which are not known to be NP-complete, finding a polynomial time algorithm for any of these problems would not imply that P=NP, but only that the specific problem is in P.

DOMINATING SET and CONNECTED DOMINATING SET are not only NP-complete for general graphs, but for a range of different graph classes like split graphs [4, 9].

As we mentioned in the previous section, for some, but not all, NP-complete problems enumeration of all feasible solutions and checking their sizes is the only way we know that solves them. Whether one can avoid enumeration for the solution of all NP-complete problems is an important open question.

For CONNECTED DOMINATING SET on general graphs we know that an enumeration algorithm of minimal sets is not the fastest way to solve the optimization version of the problem. On general graphs, the best enumeration algorithm that lists minimal cds known is the trivial one that runs in time $\mathcal{O}^*(2^n)$ [11]. Fomin, Grandoni and Kratsch gave an algorithm that solves CONNECTED DOMINATING SET on general graphs with running time $\mathcal{O}(1.9407^n)$ [7]. The current fastest algorithm for solving this problem on general graphs is due to Abu-Khzam, Mouawad, and Liedloff [2] and has run-

ning time $\mathcal{O}(1.8619^n)$. Even though enumeration of minimal objects is not the fastest way to solve CONNECTED DOMINATING SET on general graphs, it might be the case for some specific graph classes.

In fact, Fomin, Kratsch and Woeginger [9] gave an algorithm with best running time for DOMINATING SET and CONNECTED DOMINATING SET on split graphs, which was $\mathcal{O}(1.41422^n)$ [9]. They also showed that for both DOMINATING SET and CONNECTED DOMINATING SET in split graphs, the solution can be found among the sets of mcds. In 2015, Golovach, Heggernes and Kratsch [11] gave an $\mathcal{O}(1.3803^n)$-time algorithm for enumerating the mcds in split graphs. Thereby, they decreased the best running time on both DOMINATING SET and CONNECTED DOMINATING SET in split graphs. To the best of our knowledge, a faster algorithm for CONNECTED DOMINATING SET in split graphs that avoids enumeration has not been discovered. This constitutes another example of a hard problem for which a fastest solution is obtained by enumerating all solutions by a clever algorithm and then picking the best solution.

1.5 The abstract of this thesis

This above $\mathcal{O}(1.3803^n)$-time algorithm for enumerating mcds in split graphs exhibits several of the issues we have discussed so far. The algorithm by Golovach et al. [11] gives the best known upper bound on the maximum number of mcds in split graphs. They also give the best known lower bound example on the maximum number of mcds in split graphs, which has 1.3195^n mcds, so there is an albeit small gap between the lower and the upper bound.

In this thesis we implement and test the mentioned enumeration algorithm, and with these tests we find a new lower bound example with 1.3195^n mcds. On the theoretical side, we improve the running time of the enumeration algorithm to $\mathcal{O}(1.3674^n)$, thereby proving that the maximum number of mcds in split graphs is at most 1.3674^n. Consequently, we narrow the gap between the known upper and lower bound on the maximum number of mcds in split graphs.

We now give a brief overview of this thesis:

In Chapter 2 we define split graphs, and study lower bounds. We study the graph that gives the best known lower bound, and we try to search for a new or better lower bound by generating all split graphs of size up to 11 nodes. We report on a new lower bound example that we found, with the same number of mcds as the best lower bound example, and we present this

graph and its mcds.

In Chapter 3 we analyze the algorithm by Golovach et al. [11] for enumerating mcds that gives the best known upper bound. We correct a couple of small errors in this algorithm, and we analyze the running time of this new and amended algorithm.

In Chapter 4 we go through the details of how we implemented the enumeration algorithm and our graph generation algorithms. We also discuss some of the obstacles that we encountered such as how to generate only non-isomorphic split graphs.

Chapter 5 contains the main contributions of this thesis. We start by presenting some of our test results, which lead to understanding where the real upper bound might lie. Then we analyze the branching rules with the highest running time in the enumeration algorithm by Golovach et al. [11], and see what type of structures a graph requires to use these rules. Using the insight from these examples and our tests, we improve the running time on all of these worst case rules in the algorithm.

In Chapter 6 we summarize the work and results of this thesis. We discuss how the results of this thesis can be applied to co-bipartite graphs. We also give our thoughts and remarks on further work with the subject of this thesis.

Chapter 2

Split graphs and lower bounds

Before we start to examine the details of the enumeration algorithm by Golovach et al. [11], we will look at split graphs and the lower bound on the maximum number of mcds in split graphs. We will do this by studying some specific graph examples. We will also present the graph that gives the best known lower bound on the maximum number of mcds in split graphs, and we will give a brief discussion on this lower bound. As the main contribution of this chapter, we will present a new lower bound example that we discovered with the same lower bound as the current best known graph example. Before we give a formal definition of split graphs, we will begin by providing a brief discussion on why it might be useful to look at upper and lower bounds on a specific graph class like split graphs instead of just looking at general graphs.

2.1 Graph classes

A graph class is a subclass of general graphs containing only the graphs that have a specified property. Most graph classes contain an infinite number of graphs. Some graph classes have forbidden or required structures. An example of a graph class that has a forbidden structure is the class of trees. The class of trees consists exactly of those graphs that are connected and that do not contain any cycles.

As we briefly discussed in Chapter 1, when we are looking for the maximum number of various objects in graphs, there might be a gap between what we

are able to prove as an upper bound and the example that we are able to find as a lower bound. It is important to note that in reality there is never a gap between the upper and lower bounds, there is one bound, which is the maximum number of a specified object that any graph can have. To try to find this correct maximum number of the specified object that a graph can have, we try to bound this number from above and below, which corresponds to the upper and lower bounds. The goal is to get equal upper and lower bounds, because then we know that we have found the correct maximum number of the specified object that any graph can have. If there is a gap between the upper and lower bounds that we are able to find, we know that the correct maximum number of the specified object a graph can have is somewhere between these bounds. So the smaller the gap is between the upper and lower bounds, the closer we are to knowing the correct maximum number of objects that any graph can have.

For some problems it can be difficult to close or tighten the gap between the upper and lower bound that we are able to provide for general graphs. In some cases we can obtain a smaller gap between the upper and the lower bounds, or even close the gap completely, by looking at a graph class, and try to exploit some of its structure. Another reason why it can be useful to only concentrate on a specific graph class can be that some applications of a problem might only concern that specific graph class. Even if we have a tight bound for a problem on general graphs, this bound might be high. In some cases we can get a better bound on the same problem if we restrict it to a specified graph class.

For the maximum number of mcds in general graphs, the upper bound is given by 2^n, and the lower bound is given by $3^{(n-2)/3} \sim 1.4423^n$ in [11]. The upper bound is the trivial bound, thus no algorithm has been able to enumerate these sets in less time. As we will see later in this thesis there is a smaller upper bound of 1.3803^n on the maximum number of these sets in split graphs, but also a smaller lower bound of $4^{n/5} \sim 1.3195^n$. The reason why we get a smaller lower bound on the maximum number of mcds in split graphs than in general graphs is because the graph example that gives the lower bound for general graphs is not a split graph. Thus, it cannot be used to give a lower bound on the maximum number of mcds in the class of split graphs. As we will see, the upper bound for the class of split graphs is achieved by a non-trivial enumeration algorithm.

Graph Class	Lower Bound	Upper Bound
general	$15^{n/6}$	1.7159^n
chordal	$3^{n/3}$	1.6181^n
split	$3^{n/3}$	1.4656^n
proper interval	$3^{n/3}$	1.4656^n
cograph	$15^{n/6}$	$15^{n/6}$
trivially perfect	$3^{n/3}$	$3^{n/3}$

Table 2.1: Lower and upper bounds on the maximum number of minimal dominating sets. Note that $15^{n/5} \approx 1.5704^n$ and $3^{n/3} \approx 1.4422^n$.

In Table 2.1 we can see some results by Couturier, Heggernes, van 't Hof and Kratch [5]. These results are bounds on the maximum number of minimal dominating sets, and not on mcds which we are working with in this thesis, but they serve well as an illustration. As we can see the bounds for general graphs is not tight, but they have been able to find a tight bound for both the class of cographs and trivially perfect graphs. Furthermore the upper bound for trivially perfect graphs is less than the lower bound on general graphs. This means that the maximum number of minimal dominating sets in general graphs is higher than in trivially perfect graphs. Some of the gaps in Table 2.1 have been closed or narrowed since the publication of these results. For example, the best known upper bound for chordal graphs is now 1.5214^n [1], for split graphs $3^{n/3}$ [6], and for proper interval, in fact even interval graphs, is $3^{n/3}$ [10].

2.2 Split graphs

A split graph G is a graph where the nodes can be partitioned into a clique C and an independent set I, such that every node in G is either in I or in C. Any number of edges between C and I can be present in G. In Figure 2.1 we see two examples of split graphs.

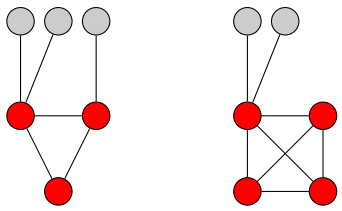

Figure 2.1: Examples of split graphs, where the red nodes represent the clique, and the gray nodes represent the independent set.

In the rest of this thesis when we are working with split graphs we will denote the set of all nodes in the clique by C, and the set of all nodes in the independent set by I.

First let us have a look at how mcds in split graphs look like.

Lemma 2.1. *Let G be a split graph, if G has a mcds D containing a node from I then $|D| = 1$.*

Proof. Assume for contradiction that there is a mcds D that contains a node $i \in I$ and $|D| > 1$. For D to be connected, one of i's neighbors has to be in D, let us call this node c. Since $i \in I$, it is only adjacent to nodes in C. This implies that $c \in C$, which again implies that c is adjacent to every node in C, and some number of nodes in I. It follows that $N_G(i) \subseteq N_G(c)$, so i cannot have a private node in G, and therefore the set D is not minimal. This is a contradiction to our assumption that there was a mcds D with $|D| > 1$ that contained a node from I. □

Corollary 2.1. *Let G be a split graph with a mcds D such that $|D| \geq 2$. Then $D \subseteq C$.*

Note that if a graph contains such a mcds D with $|D| = 1$ containing only a single node $i \in I$, then G must be complete. The reason for this is that D needs to dominate the entire graph. Which means that i needs to be adjacent to every node in the graph. Since there are no edges between nodes in I, I can only contain i, and i needs to be adjacent to the whole clique. Which means that in the cases where G is not complete, all mcds are subsets of C.

2.3 Lower bound

The lower bound on the maximum number of specified objects in any graph is obtained from the graph that contains the highest known number of the sought for object. The lower bound gives us the highest number of the specified object that we can prove that any graph contains, and it is given as a function of the number of nodes in the graph. It can be used to show that the running time of an algorithm that enumerates these objects would have to spend at least this amount of time. The graphs that make up the lower bounds can often have a special structure. This structure is usually several copies of identical subgraphs. The graph that makes up the lower bound needs to have the property that it can be expanded infinitely so that

we can have a function for the maximum number of objects that grows as the number of nodes in the graph grows.

In the rest of the figures of split graphs in this thesis we will not draw any of the edges between pairs of nodes in the clique.

For split graphs a graph example giving the lower bound might be given by a graph with structure similar to the one in Figure 2.2. This graph consists of several copies of the same subgraph. For ease of explanation we will call each such subgraph a component. This should not be confused with a connected component. It is important to note that there are no edges between the independent set in one component and the nodes of another component. Inside a component the edges might be different than those we draw as an illustration. If we have k nodes in each such component then we have n/k identical components, where n is the number of nodes in the graph, assuming that k divides n. If each such component has p mcds, then for a given component each of its p mcds can be combined with any of the p mcds in each of the other components, to create a mcds for the whole graph. So we have $\underbrace{p \cdot p \cdot ... \cdot p}_{n/k} = p^{n/k}$ mcds in the graph.

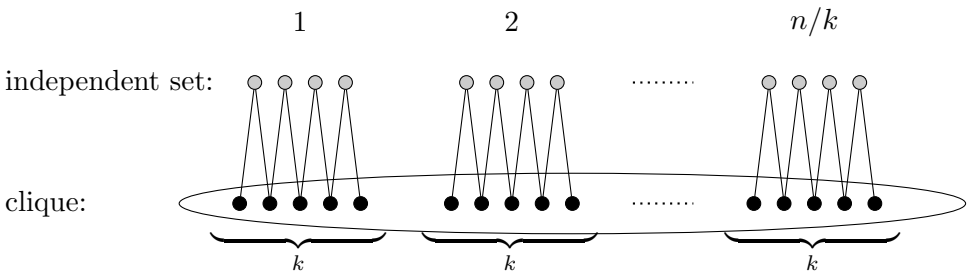

Figure 2.2: A graph having $p^{n/k}$ minimal connected dominating sets.

We are now going to look at different such graphs with different numbers of nodes in each component in the graph, to analyze how such graphs might be used to find the best lower bound. In each graph we only consider the nodes in the clique as potential members of the mcds.

Note that if we look at each component by itself, in some of these graphs it might look like the node in the independent set could be a mcds, but this is not feasible for our purpose. We need to combine the partial solution of each component to obtain a mcds for the whole graph. This combined solution would be of size greater than 1, so by Corollary 2.1 all members of the mcds need to be from the clique.

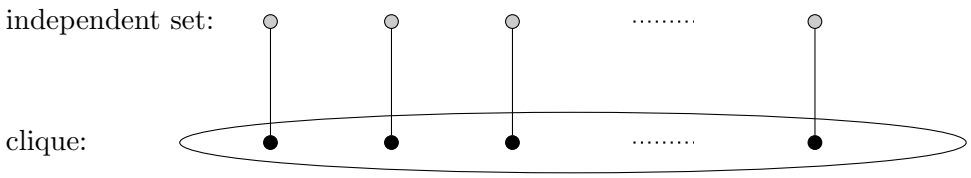

Figure 2.3: A graph having $1^{n/2} = 1$ mcds.

In Figure 2.3 we have 2 nodes in each component, thus $k = 2$. We have 1 mcds in each component, consisting of the node from the clique, so $p = 1$. Which gives us $1^{n/2} = 1$ mcds, no matter how many nodes we have in the graph.

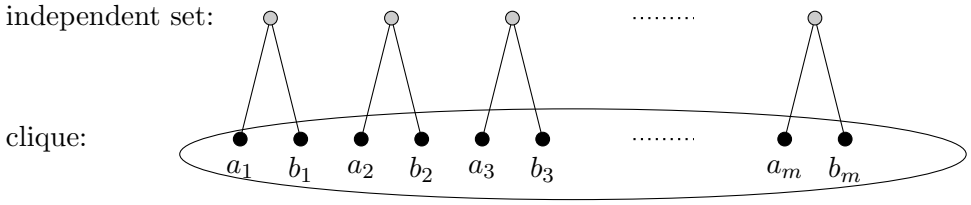

Figure 2.4: A graph having $2^{n/3} = 1.2599^n$ mcds.

In Figure 2.4 we have 3 nodes in each component, thus $k = 3$, and we have 2 mcds in each component, since we can choose a_i or b_i independently from each component, so $p = 2$. This gives us $2^{n/3} = 1.2599^n$ mcds.

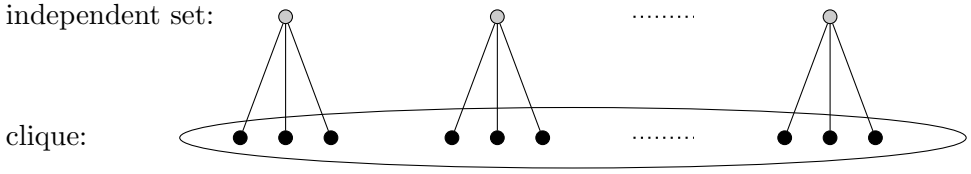

Figure 2.5: A graph having $3^{n/4} = 1.3161^n$ mcds.

In Figure 2.5 we have 4 nodes in each component, thus $k = 4$, and we have 3 mcds in each component, so $p = 3$. This gives us 1.3161^n mcds.

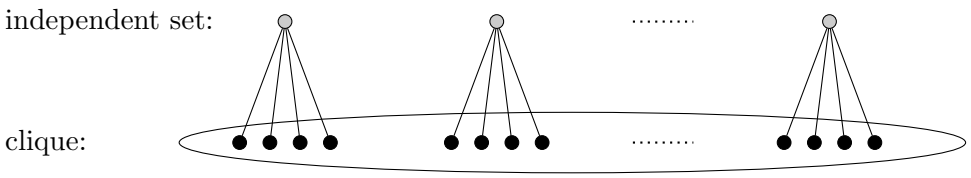

Figure 2.6: A graph having $4^{n/5} = 1.3195^n$ mcds.

In Figure 2.6 we have 5 nodes in each component, thus $k = 5$, and we have 4 mcds in each component, so $p = 4$. This gives us 1.3195^n mcds. This is actually the current best known lower bound on the maximum number of mcds in split graphs, first presented by [11]. This graph consists of several copies of the graph in Figure 2.7, which we will denote as L_1 in the rest of this thesis.

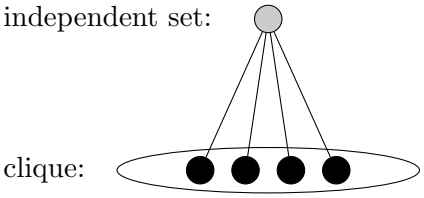

Figure 2.7: The graph L_1, that makes up the lower bound of 1.3195^n on the maximum number of mcds in split graph.

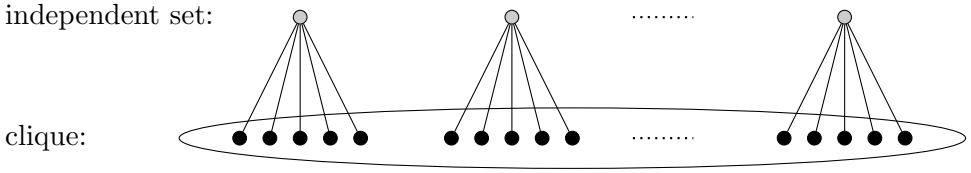

Figure 2.8: A graph having $5^{n/6} = 1.3077^n$ mcds.

In Figure 2.8 we have six nodes in each component, thus $k = 6$, and we have five mcds in each component, so $p = 5$. This gives us 1.3077^n mcds.

Keeping the same structure in the graph and increasing the number of nodes in each component to more than 5 will just decrease the function of the number of mcds in the graph further, so this will not give a better lower bound. However this does not mean that 1.3195^n is the correct lower bound, a different structure of the graph might still give a higher number of mcds.

In order to see if there exists other or better lower bound examples, we could investigate the possibility of finding relatively small split graphs with a relatively large number of mcds which we could combine to an arbitrarily large split graph by taking copies joined at the clique. For this we need a split graph on n nodes to have at least 1.3195^n mcds consisting of nodes in its clique. We have computed this value for $1 \leq n \leq 11$, and the corresponding numbers are given in the second column of Table 2.2. This means for example that a split graph on 9 nodes must have at least 13 mcds if we want to use it as a component to generate infinite lower bound examples.

Nodes in the graph	Smallest number of mcds in the clique needed to break the lower bound	Maximum number of mcds found in the clique
2	$> 1.3195^2 \sim 1.741 : 2$	1
3	$> 1.3195^3 \sim 2.297 : 3$	2
4	$> 1.3195^4 \sim 3.031 : 4$	3
5	$> 1.3195^5 = 4.000 : 4$	4
6	$> 1.3195^6 \sim 5.278 : 6$	5
7	$> 1.3195^7 \sim 6.964 : 7$	6
8	$> 1.3195^8 \sim 9.189 : 10$	9
9	$> 1.3195^9 \sim 12.125 : 13$	12
10	$> 1.3195^{10} = 16.000 : 16$	16
11	$> 1.3195^{11} \sim 21.111 : 22$	21

Table 2.2: Table of the smallest number of mcds needed in the clique part of a split graph to get a higher lower bound, and the maximum number mcds found for all graphs of size n.

In search of components to generate lower bound examples, we generated all split graphs of size up to 11 nodes, and for each graph we counted the number of mcds that were a subset of the clique part of the graph. We computed the maximum number of mcds for a graph size by taking the maximum over all graphs of this size. The results we got can be seen in column 3 of Table 2.2. As we see in this table, we found no graph of size less than or equal to 11 that can help us build examples to get a better lower bound. For $n = 5$, and $n = 10$ we hit the lower bound exactly, which is not surprising, since we found the lower bound example of 1.3195^n from Figure 2.6, which is a graph that can be expanded in size to any multiple of 5.

More surprisingly, among the graphs of size 10 that hit the lower bound exactly we were very happy to discover a new graph that we present, which serves as a new lower bound example. This graph is displayed in Figure 2.9. In the rest of this thesis we will denote the graph shown in Figure 2.9 as L_2.

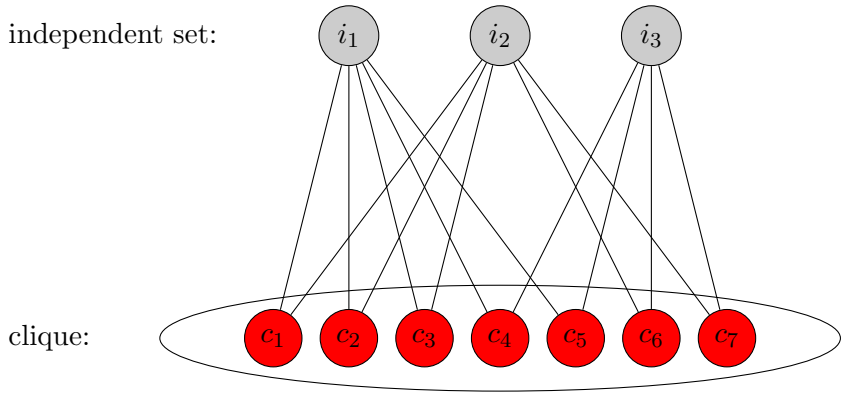

Figure 2.9: New lower bound example that gives the same lower bound as L_1. We denote this graph as L_2. This graph has 10 nodes and it has 16 mcds.

L_2 is one of the two graphs we have been able to find that matches the current lower bound. This graph is of size 10, and has 16 mcds. In this graph every node in the clique has exactly 2 neighbors in the independent set. In other words every node in the clique is adjacent to $|I| - 1$ nodes in the independent set. The other graph that gives the lower bound is the one we discussed earlier, L_1.

The mcds of L_2 are all of size 2, and they consist of all possible pairs of nodes in the clique that do not have the same neighbor set. For instance nodes c_1 and c_2 have exactly the same set of neighbors, so $\{c_1, c_2\}$ is not a mcds of L_2. Nodes c_1 and c_4 do not have the same set of neighbors so $\{c_1, c_4\}$ is a mcds of L_2. All of the 16 mcds of this graph are shown in Table 2.3.

$\{c_1, c_4\}$	$\{c_1, c_5\}$
$\{c_1, c_6\}$	$\{c_1, c_7\}$
$\{c_2, c_4\}$	$\{c_2, c_5\}$
$\{c_2, c_6\}$	$\{c_2, c_7\}$
$\{c_3, c_4\}$	$\{c_3, c_5\}$
$\{c_3, c_6\}$	$\{c_3, c_7\}$
$\{c_4, c_6\}$	$\{c_4, c_7\}$
$\{c_5, c_6\}$	$\{c_5, c_7\}$

Table 2.3: Table of all the mcds of the graph L_2.

As mentioned, for a graph to make up a lower bound it needs to have the property that it can be expanded infinitely and still have the same function of mcds in comparison to the number of nodes in the graph. If we expand

L_2 to be a graph of any size n by adding x copies of this graph, we would get the graph shown in Figure 2.10. There are $k = 10$ nodes in each such component, so we get a graph of size $n = x \cdot k$. Each copy of L_2 has $p = 16$ mcds. For a given component each of its p mcds can be combined with any of the p mcds in each of the other components, to create a mcds for the whole graph. So we get a total of $p^x = p^{n/k} = 16^{n/10} \sim 1.3195^n$ mcds.

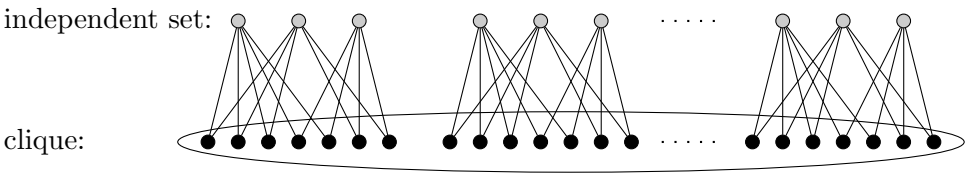

Figure 2.10: New lower bound example that gives the same lower bound as the previous graph example, that is, $16^{n/10} \sim 1.3195^n$ mcds.

One strategy to search for a graph with a higher number of mcds is to try to expand the graph that gives the current lower bound. Then check if it is possible to expand it in such a way that we get a higher number of mcds than the current lower bound. One way to expand L_2 is to keep the number of nodes in the independent set constant at 3. We can also keep the number of neighbors each node in the clique has in the independent set constant at 2. In L_2 the nodes in the independent set have roughly the same degree, so when we expand the graph we can also try to keep this property. In this general graph the clique would have $c = n - 3$ nodes, and the independent set would have $i = 3$ nodes. Each node in the independent set would have roughly $\frac{2c}{3}$ neighbors. Let us denote this expanded graph as L_{exp_1}.

The number of mcds in L_{exp_1} can be written as a function, so instead of drawing L_{exp_1} in all possible sizes to look for a new lower bound, we can plot this function and compare it to the current lower bound function. As mentioned the mcds of L_2 are all possible sets of two nodes from the clique $\{c_x, c_y\}$, where c_x and c_y do not have the exact same set of neighbors, this property also holds for L_{exp_1}. If we denote the nodes in the independent set of the generalized graph by i_1, i_2, i_3, there are three possible set of neighbors in the independent set a node in the clique can have. These possible neighbor sets are $\{i_1, i_2\}$, $\{i_1, i_3\}$ and $\{i_2, i_3\}$.

If there are c nodes in the clique there are $\frac{c}{3}$ nodes from the clique that have one specific set of neighbors $\{i_x, i_y\}$. For each node x in the clique there are $\frac{2c}{3}$ nodes in the clique that have a different set of neighbors than that of x, so every node in the clique is a part of $\frac{2c}{3}$ mcds. Thus there are $\frac{2c^2}{3}$ mcds, but every mcds in the graph is counted twice, once for each of the nodes in it, so we have to divide the whole function by 2 to get the correct number

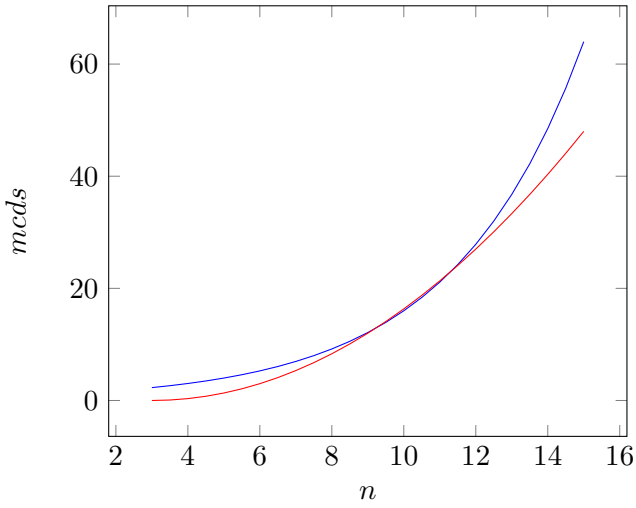

Figure 2.11: Plot where the blue function is the lower bound on mcds in split graphs, and the red function is the number of mcds in L_{exp_1}.

of mcds, hence there are $\frac{c^2}{3}$ unique mcds in the graph L_{exp_1}.

In Figure 2.11 we can see the plots of the two functions. The blue plot is the current lower bound, $f(n) = 1.3195^n$, and the red plot is the number of mcds in L_{exp_1}, given by the function $f(n) = \frac{c^2}{3} = \frac{(n-3)^2}{3}$. As we can see in Figure 2.11 the two functions intersect at approximately $n = 10$, and in all other cases the blue function exceeds the red function. This means that the current lower bound and L_{exp_1} have the same number of mcds when $n = 10$, and otherwise the current lower bound has more mcds than L_{exp_1}. So expanding L_2 to L_{exp_1} will not give a new lower bound.

The fact that we did not obtain a new lower bound does not mean that there does not exist any graph with a higher number of mcds, but if such a graph exits then each of the identical components used to generate this graph would have to be of size greater than 11. The problem with finding a graph with a higher lower bound, if it exists, is that it takes a lot of time and memory to generate and test all graphs from size 12 and onwards. As we will see in Chapter 4, during our tests and analyses of the current lower bound examples, we became convinced that the upper bound of 1.3803^n is too high. Thus we channelled our efforts into reducing the upper bound rather than making our implementation more efficient to be able to look for larger lower bound examples. In fact, as we will come back to in the last chapter, we belive that there are no better lower bound examples.

Chapter 3

The algorithm and the upper bounds

In this chapter we will go through the details of the enumeration algorithm that gives the best known upper bound on the maximum number of minimal connected dominating sets in split graphs. Since this is a branching algorithm, we provide details on branching algorithms in general before we present the algorithm that gives the upper bound.

3.1 Branching algorithms

A branching algorithm is a recursive algorithm. Branching is a well established method within enumeration and exact exponential time algorithms. The book by Fomin and Kratsch [8] gives a good explanation of branching. Here we give a brief introduction. Branching algorithms often consist of reduction rules and branching rules. The reduction rules are used to simplify the problem, or to terminate the algorithm. The branching rules are used to create several new problems of smaller size, and these are solved recursively. The computational pattern of a branching algorithm is called a search tree. If we consider each recursive call as a node in the tree, then the recursive calls made from the branching in a call will be its children. The root of this tree will be the input to the original problem.

An example of a branching algorithm is that we are looking for node subsets of an input graph with a certain property. We can keep a list of undecided nodes, and a list of nodes that are in the current solution. The undecided nodes would normally start out as the set of all nodes in the graph, and the

set of nodes in the current solution would start out empty. In each recursive call we select a node from the set of undecided nodes, and try two different possibilities. Either the selected node is in the solution, or it is not in the solution. In both cases it is removed from the set of undecided nodes, and the two recursive calls are made. This simple branching algorithm explained above would end up checking every possible subset of the nodes in the graph. Typically the recursion would have a base case, so that if the set of undecided nodes is empty we would test if the set of nodes in the current solution has the property that we are seeking. In this example the problem size would be the set of undecided nodes, which is decreased by 1 in each recursive call.

The running time of such branching algorithms is often exponential. Usually the time spent at each node in the search tree is polynomial in the input, so the important factor of the running time is the number of nodes in the search tree. It is important to note that the number of nodes in the search tree is not more than 2 times the number of leaves in the tree. So to bound the number of nodes, or the running time of the algorithm, it is sufficient to bound the number of leaves in the tree. In fact the number of leaves corresponds exactly to the number of objects produced by the algorithm. Some of them might be discarded, but each object we want to enumerate corresponds to a leaf of the tree.

To get an upper bound on the number of nodes in the tree we need to look at the branching rules, since these rules determine the number of recursive calls made, and the size of the subproblems that are solved. If a branching rule b makes m recursive calls $r_1, r_2, ..., r_m$, and the problem size in these calls are decreased by $c_1, c_2, ..., c_m$ respectively, then the branching vector of this branching rule is $b = (c_1, c_2, ..., c_m)$. If we define $T(n)$ to be the maximum number of leaves found in an input graph of size n, we can say that it is bounded by:

$$T(n) \leq T(n - c_1) + T(n - c_2) + ... + T(n - c_m)$$

The solution to this type of linear recurrence can be found by solving the equation $x^n - x^{n-c_1} - x^{n-c_2} - ... - x^{n-c_m} = 0$. The branching number of this branching rule is the unique positive real root of this equation. If the branching number of rule b is α, then the running time spent on a function that only contains branching rule b would be $\mathcal{O}^*(\alpha^n)$, where the $\mathcal{O}^*$-notation suppresses polynomial factors.

In the example given previously there is only one branching rule, it makes two recursive calls, and in both of these recursive calls the problem size is decreased by 1. So the branching vector for this branching rule would be $b = (1, 1)$. The running time of the algorithm can be found by solving the

equation $x^n - x^{n-1} - x^{n-1} = 0$. The unique real root to this equation is 2, so $\alpha = 2$, which gives a running time of $\mathcal{O}^*(2^n)$.

3.2 The enumeration algorithm

The algorithm that we will present for enumerating mcds in split graphs is from the paper by Golovach et al. [11]. This algorithm gives an upper bound of 1.3803^n on the maximum number of mcds in split graphs, which is the current best known upper bound on this problem. We will not go through the full proof of correctness for the algorithm in this thesis, as in this type of branching algorithms the proof and the implementation often go hand-in-hand. We will explain some of the steps in the algorithm to illustrate this.

The algorithm given by Golovach et al. [11] is displayed in Algorithm 3.1. Before we look at some of the steps in this algorithm we will start out by explaining the input and notations used in this algorithm.

Let G be an input split graph with C as its clique and I as its independent set. The algorithm takes three parameters K, S and X as input. Here K is a subset of C, S is a subset of I, and X is the current partial solution, which is a subset of $C \setminus K$. In particular the set X dominates the set $I \setminus S$. The goal of the algorithm is to generate all mcds of G that contain X. Initially the algorithm is called with $K = C$, $S = I$, $K = \emptyset$ and in this way it generates all mcds of G. In the algorithm we use the notation $d_S(x)$ to denote $|N_G(x) \cap S|$, and $d_K(x)$ to denote $|N_G(x) \cap K|$. The algorithm works on the subgraph of G induced by K and S. We call this subgraph H.

Algorithm 3.1 Algorithm for enumerating mcds in split graphs

```
1: function ENUMCDS(K, S, X):
2:     if X is a minimal connected dominating set of G then        ▷ 1
3:         return X and stop
4:     end if
5:     if X is a connected dominating set of G but not minimal then ▷ 2
6:         stop
7:     end if
8:     if there is an x ∈ K such that d_S(x) = 0 then              ▷ 3
9:         ENUMCDS(K \ {x}, S, X)
10:    end if
```

11: | **if** there is a $y \in S$ such that $d_K(y) = 1$ and x is the unique neighbor of y in H **then** ▷ 4
12: | | ENUMCDS$(K \setminus \{x\}, S \setminus \{y\}, X \cup \{x\})$
13: | **end if**
14: | **if** there is an $x \in K$ such that $d_S(x) = 1$ and y is the unique neighbor of x in S. Let $N_H(y) = \{x, x_2, ..., x_t\}$ for $t \geq 2$. **then** ▷ 5
15: | | **if** t == 2 **then** ▷ 5.1
16: | | | ENUMCDS$(K \setminus \{x, x_2\}, S \setminus \{y\}, X \cup \{x\})$
17: | | | ENUMCDS$(K \setminus \{x, x_2\}, S \setminus \{y\}, X \cup \{x_2\})$
18: | | **end if**
19: | | **if** $t > 2$ **then** ▷ 5.2
20: | | | ENUMCDS$(K \setminus \{x, x_2, x_3, ..., x_t\}, S \setminus \{y\}, X \cup \{x\})$
21: | | | ENUMCDS$(K \setminus \{x\}, S, X)$
22: | | **end if**
23: | **end if**
24: | **if** there is an $x \in K$ such that $d_S(x) \geq 3$. Let $N_H(x) \cap S = \{y_1, y_2, ..., y_t\}$ and $t \geq 3$ **then** ▷ 6
25: | | ENUMCDS$(K \setminus \{x\}, S \setminus \{y_1, y_2, ...y_t\}, X \cup \{x\})$
26: | | ENUMCDS$(K \setminus \{x\}, S, X)$
27: | **end if**
28: | **if** there is a $y \in S$ such that $d_K(y) = 2$ then let $N_H(y) = \{x_1, x_2\}$ and for all $i = 1, 2$ let w_i be the unique neighbor of x_i in S different from y. **then** ▷ 7
29: | | ENUMCDS$(K \setminus \{x_1\}, S \setminus \{y, w_1\}, X \cup \{x_1\})$
30: | | ENUMCDS$(K \setminus \{x_1, x_2\}, S \setminus \{y, w_2\}, X \cup \{x_2\})$
31: | **end if**
32: | **if** If there is a $y \in S$ such that $d_K(y) = 3$ then let $N_H(y) = \{x_1, x_2, x_3\}$ and for all $i = 1, 2, 3$ let w_i be the unique neighbor of x_i in S different from y; **then** ▷ 8
33: | | ENUMCDS$(K \setminus \{x_1\}, S \setminus \{y, w_1\}, X \cup \{x_1\})$
34: | | ENUMCDS$(K \setminus \{x_1, x_2\}, S \setminus \{y, w_2\}, X \cup \{x_2\})$
35: | | ENUMCDS$(K \setminus \{x_1, x_2, x_3\}, S \setminus \{y, w_3\}, X \cup \{x_3\})$
36: | **end if**
37: | **if** there is a $y \in S$ such that $d_K(y) = 4$ then let $N_H(y) = \{x_1, x_2, x_3, x_4\}$ and for all $i = 1, 2, 3, 4$ let w_i be the unique neighbor of x_i in S different from y; **then** ▷ 9
38: | | ENUMCDS$(K \setminus \{x_1\}, S \setminus \{y, w_1\}, X \cup \{x_1\})$
39: | | ENUMCDS$(K \setminus \{x_1, x_2\}, S \setminus \{y, w_2\}, X \cup \{x_2\})$
40: | | ENUMCDS$(K \setminus \{x_1, x_2, x_3\}, S \setminus \{y, w_3\}, X \cup \{x_3\})$
41: | | ENUMCDS$(K \setminus \{x_1, x_2, x_3, x_4\}, S \setminus \{y, w_4\}, X \cup \{x_4\})$
42: | **end if**
43: | **if** there is an $x \in K$ with neighbors y and y' in S then $d_K(y) \geq 5$ and $d_K(y') \geq 5$ **then** ▷ 10
44: | | ENUMCDS$(K \setminus N_H(y), S \setminus \{y, y'\}, X \cup \{x\})$
45: | | ENUMCDS$(K \setminus N_H(y'), S \setminus \{y, y'\}, X \cup \{x\})$
46: | | ENUMCDS$(K \setminus \{x\}, S, X)$
47: | **end if**
48: **end function**

Let us explain a few of the first steps of the algorithm to establish its correctness. The if-tests are labeled from 1 to 10 in Algorithm 3.1.

The first two if-tests are simply to decide whether the algorithm should stop. When the algorithm stops we are at a leaf of the search tree. The current set X is either output or discarded.

Note that the algorithm should also stop if K or S is empty. However, the way the algorithm proceeds, we know that X is a connected dominating set if this happens, so the first two tests take care of this case.

The third if-test is a reduction rule, so it does not branch. It checks if there is a node $x \in K$ that does not have any neighbors in S. This means that x cannot dominate any node in S. In this case we know that we can safely remove x from K, since x cannot be apart of any mcds. Thus we have a new call, with x removed from K, and with the same S and X as the input.

The fourth if-test is also a reduction rule. In fact we found an error in the original formulation of this rule from the paper [11], which is the one displayed in the algorithm. We will return to this error later. This if-test checks if there is a node $y \in S$ with only one unique neighbor x in the set K. In this case we know that x needs to be in all solutions, because it is the only node we can add to the current solution X to dominate y. Thus we have a new call, with x removed from K, x added to X, and y removed from S.

The fifth if-test is a branching rule. It checks if there is a node $x \in K$ with only one neighbor y in S. We know that y has at least two neighbors in K, otherwise the fourth if-test would have been applied. We know that we can either use x to dominate y, or y can be dominated by another one of its neighbors in K. If y is dominated by x, then none of the other neighbors of y can be in the same mcds, since that set would then not be minimal. If y is not dominated by x we can discard x from the solution, since x cannot be used to dominate any other node than y. The algorithm does this by branching in two different subproblems.

In 5.1 we found a similar error as in the fourth if-test that we will get back to later. This test is executed when y only has two neighbors in K, we call these x and x_2. We know that one of these are needed to dominate y. We do two different recursions, in both the recursions we remove both x and x_2 from K, and we remove y from S, and add either x or x_2 to X.

In 5.2 if y has more than two neighbors we also make two recursive calls. We know that either we use x to dominate y or we use another node to dominate y, and x can de discarded. If we use x to dominate y we can remove all of

y's neighbors in K from K, remove y from S and add x to X. If we do not use x to dominate y we can remove x from K and keep S and X as is.

When we implemented this algorithm we found an error that we edited. This error is in the fourth if-test in the algorithm. This part of the algorithm says that if there is a node $y \in S$ that only has one neighbor in K and we denote this neighbor by x, then we can remove x from K, and add x to X and remove y from S. However if x has more neighbors in S, then these should also be removed. We changed this so that instead of just saying that y is now dominated by x and removing it from S, we also remove all other neighbors of x in S. The way that this step was formulated previously was incorrect and could in some cases fail to enumerate all of the mcds in a graph. Let us illustrate this by an example and go through each step of the algorithm with this graph.

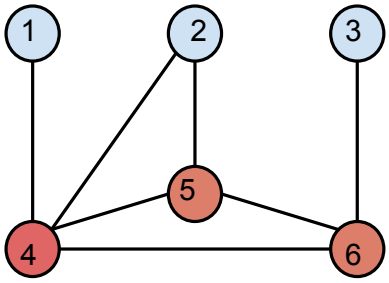

Figure 3.1: A split graph where the blue nodes form the independent set, and the red nodes form the clique.

Figure 3.1 is a split graph where the blue nodes form the independent set, and the red nodes form the clique of the graph. Thus $C = \{4, 5, 6\}$ and $I = \{1, 2, 3\}$. Let us run the algorithm on this graph, with the initial input $K = C = \{4, 5, 6\}$, $S = I = \{1, 2, 3\}$ and $X = \emptyset$. None of the steps 1-3 is applicable, since X is not a mcds or a cds, and none of the nodes in K have zero neighbors in S. Step 4 is applicable; there are two nodes in S that has one neighbor in K. Node 1 is in S and its only neighbor in K is node 4, and node 3 is in S and its only neighbor in K is node 6. Let us assume that we discover node 1 first. The algorithm would only make one recursive call in this case. The recursive call made would be to add 4 to X, remove 4 from K, and remove 1 from S.

We now continue to run the algorithm on the resulting subproblem which is $K = \{5, 6\}$, $S = \{2, 3\}$ and $X = \{4\}$. Again step 1-3 is not applicable

by the same arguments as before, but step 4 is applicable. There are two nodes in S and both of these have exactly one neighbor in K. So if the algorithm chooses node 2 as the node from S with exactly one neighbor in K, the algorithm would again make one recursive call, where 5 is removed from K and added to X, and 2 is removed from S.

If we continue to run the algorithm on the resulting subproblem which is $K = \{6\}$, $S = \{3\}$ and $X = \{4, 5\}$ step 1-3 would again not be applicable, but step 4 would be applicable. There is only one node in S, and this node only has one neighbor in K, so the recursive call made would be to remove 6 from K and add it to X, and remove 3 from S.

The final input to the recursive algorithm would be $K = \emptyset$, $S = \emptyset$ and $X = \{4, 5, 6\}$. The set X is a cds, but not a minimal one, since $\{4, 6\}$ which is a subset of X is also a cds. So the algorithm would go into the second if-test and stop, and it would not have found any mcds.

There was also a similar error in rule 5.1 in the algorithm. Where a node x_2 is added to the solution, so x_2 is also removed from the set of undecided nodes in the clique, K. At this point in the algorithm we do not know how many neighbors x_2 has. We know that x_2 has at least one neighbor, y, in the independent set. The way that the algorithm is displayed in Algorithm 3.1, y is dominated by x_2 when we add x_2 to the solution, but it does not say that any of the other potential neighbors of x_2 is dominated by x_2 when it is added to the solution. We changed this part of the algorithm so that we also say that all of the nodes in the independent set that is a neighbor of x_2 is dominated by x_2, and thereby removed from S, when x_2 is added to the current partial solution.

3.3 The upper bound

Let us briefly explain the running time of the algorithm. For this we need to check the branching vector of each rule. Note that we do not need to check the reduction rules, as these are all polynomial, so we will only look at the branching rules. In the algorithm we presented previously, all the if-tests are labeled with a number, and we will use these numbers to denote the branching rules of the algorithm.

In 5.1 we make two recursive calls, and in each of these the size of the problem is decreased by 3. So the branching vector for this rule is $(3, 3)$. The running time can be found by solving the equation $x^n - x^{n-3} - x^{n-3} = 0$. The unique real root of this equation is $\alpha = \sqrt[3]{2} \approx 1.2600$, so this branching

rule gives a running time of: $\mathcal{O}(1.2600^n)$.

In 5.2 we make two recursive calls, the size of the problems in these calls is decreased by 1 and 4. The branching vector for this rule is $(1, 4)$. By the same arguments as given in 5.1 this gives a running time of: $\mathcal{O}(1.3803^n)$.

In 6 we make two recursive calls, the size of the problems in these calls is decreased by 4 and 1. The branching vector for this rule is $(4, 1)$. By the same arguments as given in 5.1 this gives a running time of: $\mathcal{O}(1.3803^n)$.

In 7 we make two recursive calls, the size of the problems in these calls is decreased by 3 and 4. The branching vector for this rule is $(3, 4)$. By the same arguments as given in 5.1 this gives a running time of: $\mathcal{O}(1.2208^n)$.

In 8 we make three recursive calls, the size of the problems in these calls is decreased by 3, 4 and 5. The branching vector for this rule is $(3, 4, 5)$. By the same arguments as given in 5.1 this gives a running time of: $\mathcal{O}(1.3248^n)$.

In 9 we make four recursive calls, the size of the problems in these calls is decreased by 3, 4, 5 and 6. The branching vector for this rule is $(3, 4, 5, 6)$. By the same arguments as given in 5.1 this gives a running time of: $\mathcal{O}(1.3803^n)$.

In 10 we make three recursive calls, the size of the problems in these calls is decreased by 7, 7 and 1. The branching vector for this rule is $(7, 7, 1)$. By the same arguments as given in 5.1 this gives a running time of: $\mathcal{O}(1.3422^n)$.

When we have several branching vectors in an algorithm we use the one with the worst outcome to compute the running time. In our case this is $\mathcal{O}(1.3803^n)$ which corresponds to rule 9, 6 and 5.2. This is also an upper bound on the maximum number of mcds found by the algorithm. The number of mcds corresponds to the number of leaves in the search tree. This can be shown by using the fact that when a mcds is found, or when a cds is discarded, the algorithms stops, which means that we are in a leaf of the search tree. This implies that we cannot find more mcds than the number of leaves in the search tree.

This completes the description of the algorithm as we implemented it. However, in Chapter 5, we will come back to upper bound discussions, and suggest some improvements on some of the branching rules.

Chapter 4

Implementation details

In this chapter we will go through the implementation details. We will describe how we generated random split graphs, and how we generated all split graphs of a certain size. We will go into details on how we implemented the algorithm for enumerating mcds in split graphs described in Chapter 3.2. We will also describe how we tested the correctness of the algorithm and discuss the issue of generating non-isomorphic split graphs. We begin by explaining how we generated a random split graph of a given size.

4.1 Generation of random split graphs

We want to generate random split graphs to be able to test Algorithm 3.1 presented in Chapter 3.2 on large graphs. By doing this we can check how the running time of the algorithm compares to the number of mcds found by the algorithm, and try to determine if the algorithm discards a lot of mcds and therefore uses an unnecessary amount of time. We also use this generation algorithm to test different properties of split graphs, such as what percentage of nodes in the clique and independent set give the highest number of mcds. The algorithm for generating a random split graph is shown in Algorithm 4.1 and explained in the next paragraphs.

When we implemented the method to generate a random split graph we found it useful to take as input the size of the graph to be generated. We did this by taking three integers as input, the number of nodes in the clique, the number of nodes in the independent set and the number of edges in the graph.

Algorithm 4.1 Algorithm to generate a random split graph

```
1: function GENERATESPLIT(i, c, e):                                  ▷ Part 1
2:     if e > (c^2 − c)/2 + i ∗ c or e < (c^2 − c)/2 + i then
3:         return null
4:     end if
5:
6:     n = c + i
7:     G = new graph of size n
8:     clique = first c nodes in G
9:     independent = last i nodes in G
10:
11:    for each pair of nodes (c_x,c_y) in clique do
12:        add an edge between c_x and c_y in G
13:    end for
14:    decrease e by c ∗ (c − 1)/2
15:                                                                  ▷ Part 2
16:    for each node ind in independent do
17:        cli = random node from clique
18:        add an edge between cli and ind in G
19:    end for
20:    decrease e by i
21:
22:    while e > 0 do
23:        n_i = random node from independent
24:        n_c = random node from clique
25:        while there is an edge between n_i and n_c in G do
26:            n_c = random node from clique
27:            n_i = random node from independent
28:        end while
29:        add an edge between n_i and n_c in G
30:        decrease e by 1
31:    end while
32:    return G
33: end function
```

To simplify the calculation of the running time of this algorithm we split it into two parts. These two parts are marked in Algorithm 4.1. Part 1 consists of creating the graph and the edges between the nodes in the clique. Note that we do not actually need to add the edges between the nodes in the clique, as long as we know which nodes belong to the clique. The algorithm given by Golovach et al. [11] that we use to enumerate mcds in split graphs takes the set of nodes in the clique as input, so to run this algorithm the addition of edges between the nodes in the clique is redundant. We still chose to keep this in our implementation to generate actual split graphs, but this part can easily be removed. As mentioned, the input to our method consists of three integers c, i and e, which correspond to the number of nodes in the clique, the number of nodes in the independent set and the number of edges in the graph, respectively.

We started by checking if the input corresponds to a split graph. If $e < \frac{c^2-c}{2} + i$ the method returns null, since there does not exist any connected split graph satisfying the constraints given by the input. This is because every node in the clique needs to have an edge to all other nodes in the clique, which corresponds to $\frac{c^2-c}{2}$ edges. For the graph to be connected every node in the independent set needs to have an edge to at least one node in the clique which corresponds to i edges. If $e > \frac{c^2-c}{2} + ic$ we also return null. This is because there can be at most ic edges between the independent set and the clique, in addition to the $\frac{c^2-c}{2}$ edges in the clique, and there are no other edges in a split graph.

Then we created a graph G with the correct number of nodes, that is $c + i$. After that we added all the $\frac{c^2-c}{2}$ edges between the nodes in the clique to the graph, since these edges have to be in the graph to make it a split graph satisfying the constraints given by the input.

Part 2 of the algorithm consists of adding the edges that are between one node in the independent set and one node in the clique. We want the graph to be connected, since we are looking for mcds. So for each node in the independent set we add one edge to a random node in the clique. After this, we have added $\frac{c^2-c}{2} + i$ edges to the graph, which is the smallest number of edges we can have in a connected split graph with c nodes in the clique, and i nodes in the independent set.

Then we need to add the $e - \frac{c^2-c}{2} - i$ extra edges to satisfy the constraint given by the number of edges in the input. Each of these extra edges needs to be between one node from the independent set and one node in the clique. To do this we made a loop that runs while we still needed to add more edges. Inside the loop we picked two random nodes, one node n_c from the clique and one node n_i from the independent set. If n_i and n_c were not already

neighbors by the previously added edges, we could simply add this edge. However, to take care of this possibility, we added another loop that runs while G contains an edge between n_c and n_i. We picked new random nodes n_c and n_i until we found a pair that are not adjacent.

In Part 1 of Algorithm 4.1 the graph is created, and the edges between every pair of nodes in the clique are added. In Part 2 for each node in the independent set we add one edge to a random node in the clique, to make the graph connected. The rest of the edges are added to the graph, each of these is between one random node from the independent set and one random node from the clique.

Since in Part 1 we have exactly as many steps as there are edges in the clique, we have readily the following:

Lemma 4.1. *The running time of Part 1 of Algorithm 4.1 is linear in the size of the generated split graph.*

Calculating the running time of Part 2 of Algorithm 4.1 is not straightforward since in practice the same pair of random nodes can be picked out any number of times inside the while-loop. It follows that the running time of this part of the algorithm where we add the random edges between nodes in the independent set and nodes in the clique can get pretty high. For our use of the generation algorithm the running time of the generation had minor impact on the total calculation time. This is because we run an exponential time algorithm on all the graphs that we generate. However this algorithm might actually be the most "random" way of adding edges to a graph, since every edge has the same probability of being added to the graph in each iteration. As we will see in the next section we could have made Part 2 of the algorithm run in linear time. Since Part 1 of the algorithm runs in linear time by Lemma 4.1, this improvement would make the whole generation algorithm run in linear time.

4.2 Linear time generation of random split graphs

The way that we generate split graphs is not necessarily linear in the size of the generated graph. In other applications, it might be useful to have a faster generation of random split graphs. We will briefly discuss how we can obtain a linear time algorithm that generates a random split graph.

Most of the steps in Algorithm 4.1 run in linear time. Part 2 of the algorithm however does not run in linear time if we randomly pick the same edges to

add in the graph several times. This is the part of the algorithm where we add the edges between nodes in the clique and nodes in the independent set. To get a linear time algorithm that generates a random split graph, this is the only part we need to change.

Algorithm 4.2 Linear time generation of random split graphs

```
1: edges = list containing the number of neighbors in the clique for each
   node in the independent set.
2: list = list of all nodes in the clique.
3: for each node in the independent set do
4:     num = size of list
5:     for i = 0...edges[node] do
6:         neighbor = random number between 0 and num
7:         add edge between node and list[neighbor]
8:         swap(list[num], list[neighbor])
9:         decrease num by 1
10:    end for
11: end for
```

If we have a random way to decide the number of neighbors in the clique that each node in the independent set should have or vice versa we could use Algorithm 4.2, which runs in time linear in the size of the generated graph. The number of neighbors in the clique that each node in the independent set can have, has to be somewhere between 1 and $min(c, e-(c^2-c)/2-i+1)$, where i is the number of nodes in the independent set, e is the number of edges in the graph, and c is the number of nodes in the clique.

Algorithm 4.3 Random generation of the number of edges for each node in the independent set

```
1: edges = list containing all 1s of size equal to the number of nodes in the
   independent set.
2: list = list containing all nodes in the independent set
3: for i = 0...e − (c^2 − c)/2 − i do
4:     node = random node from list
5:     increase edges[node] by 1
6:     if edges[node] == c then
7:         remove node from list
8:     end if
9: end for
10: return edges
```

One way to randomly assign the number of neighbors in the clique for each node in the independent set is given in Algorithm 4.3. Here, for each edge, we decide a random node from the independent set that it should be assigned to. Note that c is the maximum number of neighbors any node in the

independent set can have since there cannot be any edges between the nodes in the independent set. Thus each node in the independent set can at most have an edge to every node in the clique. If one of the nodes in the independent set is assigned to c edges, we remove it from the list of nodes in the independent set that can be assigned new edges.

Algorithm 4.2 runs in linear time with respect to the number of edges. This can be shown by the fact that what this algorithm does is, that for each node i in the independent set, and for each edge that i is a part of, it adds this edge to the graph. Algorithm 4.3 also runs in linear time in the number of edges. This can be shown by the fact that all this algorithm does is to assign each edge that is added to the graph to a node in the independent set.

4.3 Generation of all split graphs of a given size

In addition to the random generation of split graphs, we wanted to create a method that generates all split graphs of a given size to be able to check if any new lower bound examples existed, or if there existed any graphs with a higher number of mcds than the lower bound. It is actually by generating all graphs of size up to 11 that we found the new lower bound example, L_2, that we discussed in Chapter 2. Since we did not find a new lower bound example in any of these graphs, we know that if there exists a better lower bound, the graph that makes up the components of this lower bound example would have to be of size at least 12.

When we implemented our method to generate all split graphs of a given size we took the size of the graphs to be generated as input, and returned the set of all the possible split graphs that have the given size. The first thing we did was to make a loop that decides the sizes of the clique and the independent set. We started with all split graphs of size n with 1 node in the clique, and the rest of the $n-1$ nodes in the independent set. After that we generated all split graphs of size n with 2 nodes in the clique and the rest of the $n-2$ nodes in the independent set and so on.

Once we had the number of nodes in the independent set, and the number of nodes in the clique, some properties of the graph were given. All split graphs with c nodes in the clique have to contain $\frac{c^2-c}{2}$ edges between the nodes in the clique, and they cannot contain an edge between any pair of nodes in the independent set. We created a list of possible edges that could be in the graph between one node from the independent set and one node from the clique. This list is of size ci where c is the number of nodes in the

clique and i is the number of nodes in the independent set.

We made a "base graph" for a split graph with a given number of nodes in the clique and independent set that contained all the nodes and all the edges between the nodes in the clique, since they have to be in all split graphs with the given properties. Note that again the edges between the nodes in the clique are redundant, since we know which nodes that are in the clique, so we could have skipped the part where we add these edges. We then ran a branching algorithm with the "base graph" and with the list of possible edges as input. This algorithm is shown in Algorithm 4.4.

If we run this algorithm with the inputs described above we obtain the set of all possible split graphs with the given size. Since we are searching for mcds, we only want graphs that are connected. After running this algorithm we removed the graphs that were not connected from the set of graps we generated, by using a simple depth first search to check for connectivity.

Algorithm 4.4 Algorithm that creates all graphs given a list of possible edges

```
1: function createAllSplitGraphs(G, listOfEdges):
2:     graphs = new set
3:     if listOfEdges is empty then
4:         add G to graphs
5:         return graphs and stop
6:     end if
7:     e = any edge from listOfEdges
8:     add createAllSplitGraphs(G∪{e}, listOfEdges\{e}) to graphs
9:     add createAllSplitGraphs(G, listOfEdges \ {e}) to graphs
10:    return graphs
11: end function
```

The algorithm given in Algorithm 4.4 branches in two different branches in each step, and in both branches the problem size is decreased by 1. The problem size is the size of the list of possible edges, which is of size ci. So the algorithm runs in time $\mathcal{O}(2^{ci})$. Since both c and i are bounded by n the running time is equal to $\mathcal{O}(2^{n^2})$. The running time of this algorithm corresponds to the number of graphs generated by the algorithm. This number is quite large, so with this initial method, we managed to generate all graphs of size up to 8, but when we tried to generate all graphs of size 9 we ran out of memory. In Section 4.6 we explain how we worked to be able to generate all connected split graphs with up to 11 nodes.

4.4 Implementation of the enumeration algorithm

We used Java to implement the main algorithm. We used adjacency lists to represent the graphs. How to implement most of the steps in the enumeration algorithm presented in Chapter 3 is pretty self-explanatory, so we will only go through a few of them. The steps of checking if a given subset of the nodes in a graph is a mcds, or a cds but not a minimal one, is not obvious, so we will focus on these steps. We will also give code examples on how we implemented these steps. We chose to change the formulation of the first two steps of the algorithm presented in Chapter 3.2 slightly. We did not change the way the algorithm works, we just combined the first two if-tests into one nested if-test as shown in Algorithm 4.5.

Algorithm 4.5 New formulation of the first two steps of the algorithm for enumerating mcds in split graphs

```
1: function ENUMCDS(K, S, X):
2:     if X is a connected dominating set of G then
3:         if X is minimal then
4:             return X and stop
5:         end if
6:         stop
7:     end if
8:     .
9:     .
10:    .
11:    .
12: end function
```

The first thing we do is to check if the set X is a cds of G. We do this by checking if X dominates G. Note that we do not need to check for connectedness since we only consider subsets from the clique, so all of the subsets that we consider are connected. If X dominates G, then we know that X is a cds of G. If X is a cds of G we tested if X is minimal, if it is minimal we know that X is a mcds, and we return X and stop. If X is not minimal we stop without returning X since we know that X is not a mcds of G.

As mentioned, we need to check if X dominates G. To do this we iterate though all the nodes in the independent set. For each node i in the independent set we iterate through its neighbors in G, and check that at least one of these is in X. We do not have to look at the nodes in the clique since we know that the nodes in the clique are dominated as long as $|X| \geq 1$. Our implementation of this is shown in Algorithm 4.6.

Algorithm 4.6 Algorithm to check if a subset X of the clique part of a split graph G dominates I, the independent set of G.

```
1: function isDominating(G, X, I):
2:     for Integer i in I do
3:         if X does not contain any of i's neighbors in G then
4:             return false
5:         end if
6:     end for
7:     return true
8: end function
```

To check if the set X is minimal we iterate through the nodes in X. For each node $x \in X$ we try to remove x from the set X and see if the set $X \setminus \{x\}$ is a cds of G. If the set $X \setminus \{x\}$ is a cds of G for some $x \in X$, then we know that X cannot be a mcds of G by Lemma 1.1, so we return false. The way that we chose to implement this is shown in Algorithm 4.7. It uses the metod isDominating is given by Algorithm 4.6.

Algorithm 4.7 Algorithm to check if a subset X of a graph G is minimal

```
1: function isMinimal(G, X):
2:     for node in X do
3:         remove node from X
4:         if isDominating(G, X) then
5:             return false
6:         end if
7:         add node to X
8:     end for
9:     return true
10: end function
```

4.5 Testing the correctness of the algorithm

When an algorithm is to be implemented, there are several ways errors can be inserted. There might be errors in the algorithm itself or there might be unintentional errors inserted during the implementation. Some algorithms are very detailed whereas other are more high level. Although all algorithms are accompanied with a correctness proof, there might be errors in the base cases or in the the stop criteria of the algorithm.

To test the correctness of the algorithm and its implementation we implemented in addition a trivial algorithm for enumerating mcds in a given graph. This trivial algorithm runs in time $\mathcal{O}^*(2^n)$, and it is shown in Al-

gorithm 4.8. This algorithm uses the methods createAllSubsets and isMcds, which are shown in Algorithm 4.9 and Algorithm 4.10 respectively. Algorithm 4.10 uses the metohds isDominating, and isMinimal shown in Algorithm 4.6 and Algorithm 4.7 respectively. Algorithm 4.8 creates all subsets of the nodes in the clique of the input graph, and tests whether or not each of these subsets is a mcds. We do not need to check the subsets that contain nodes from the independent set, as we are only interested in mcds consisting of nodes from the clique. We do not need to test for connectedness since we only look at subsets of the nodes in the clique, and we know that these are connected. When we tested the algorithm we ran both the trivial algorithm and the algorithm by Golovach et al. [11] which we presented in Chapter 3.2 on the same graphs, and tested if we found the same number of mcds.

Algorithm 4.8 Trivial algorithm for enumerating mcds in a given split graph

```
1: function enumCDS(graph, clique):
2:     powerset = createAllSubets(clique)
3:     MCDS = new Set
4:     for subset in powerset do
5:         if isMcds(graph, subset) then
6:             add subset to MCDS
7:         end if
8:     end for
9:     return MCDS
10: end function
```

Algorithm 4.9 Algorithm to create all subsets of a list

```
1: function createAllSubsets(nodes):
2:     powerset = new Set
3:     if nodes is empty then
4:         return powerset
5:     end if
6:     head = first element in nodes
7:     tail = list of elements 1..n − 1 from nodes
8:     for set in createAllSubets(tail) do
9:         subset = copy of set
10:        add head to subset
11:        add subset to powerset
12:        add set to powerset
13:    end for
14:    return powerset
15: end function
```

Algorithm 4.10 Checks whether a given subset of the nodes in a graph is a mcds

```
1: function isMcds(graph, subgraph):
2:     if subgraph is empty then
3:         return false
4:     else if isDominating(graph, subgraph) and isMinimal(graph,
   subgraph) then
5:         return true
6:     end if
7:     return false
8: end function
```

It was actually by doing this test that we discovered the errors in the algorithm that we discussed in Chapter 3.2. When we ran these tests we noticed that in some cases the trivial algorithm could produce a higher number of mcds than the algorithm presented in Chapter 3.2. Since every algorithm that enumerates mcds in a given graph should produce exactly the same set of mcds when run on the same input graph, we knew that something was wrong. After investigating the algorithm discussed in Chapter 3, and looking at graphs where this algorithm and the trivial algorithm found a different number of mcds, we discovered where in the algorithm the errors where located.

This test is so slow, because of the running time of the trivial algorithm, that we were only able to run them on graphs with at most 22 nodes. However, running both the test and the main algorithm on all examples up to 8 nodes gave us sufficient indication that our final implementation of the main algorithm is correct.

4.6 Isomorphism testing

The way that we generated all split graphs of a given size described previously, results in many isomorphic graphs. Ideally, it would be desirable to generate only non-isomorphic split graphs. Our initial thought was that there might be a simple and fast way to generate all non-isomorphic split graphs of a given size. We wanted to find an algorithm like this so that we did not have to generate all possible split graphs of a given size, and then remove all isomorphic graphs afterwards. We did not manage to come up with an algorithm that solves this problem. It turns out that it might not be a simple problem to solve. Royle has studied the number of non-isomorphic split graphs, and he proved that there is a one-to-one correspondence be-

tween split graphs on n nodes and minimal covers of a set on n nodes [14]. Hearne and Wagner have given a formula for the number of all minimal covers of a set on n nodes [13], but this does not imply an algorithm or a method for listing distinct set covers.

At this point, since our issue with generating graphs of size 9 or more is memory, we have two options. We can either implement isomorphism test to remove all isomorphic graphs and thereby decrease the size of the set of split graphs. Otherwise we can test the graphs during the generation method, so that we do not have to save all the graphs in memory at the same time. We wanted our generation algorithm to be useful independently of our work, thus we decided to implement isomorphism tests. This isomorphism test removes all isomorphic graphs, so that we do not have to run the algorithm that enumerates mcds on several graphs that can be considered identical. We also avoid running out of memory when trying to generate all graphs of size 9 or more. After we added the isomorphism test we were able to generate all graphs of size up to 10, and this is how we found the new lower bound example, L_2, that we discussed in Chapter 2. We will now briefly discuss the problem of testing graph isomorphism, and how we chose to implement the isomorphism test.

The problem of deciding whether two given unlabeled graphs are isomorphic is called GRAPH ISOMORPHISM. GRAPH ISOMORPHISM on general graphs is in the class of NP, and it is not known whether this problem is a NP-complete problem, or if it belongs to the class of P. This means that we do not have any polynomial time algorithm that solves the problem, and we do not know if one exists. Uehara, Toda and Nagoya [15] have proven that GRAPH ISOMORPHISM is as hard on split graphs as it is on general graphs.

We chose to implement a trivial isomorphism test to remove isomorphic graphs. By doing this we did not run out of memory as fast since we were able to reduce the size of the set of split graphs with a given size significantly. However, it took a lot more time to generate all graphs of a given size with the isomorphism test because we had to check for isomorphism between every pair of graphs that we generated. In Table 4.1 we give the number of graphs generated of all sizes up to 10 nodes with and without the isomorphism test. Royle gives a table of the number of non-isomorphic split graphs of small sizes in [14], and the numbers in his paper matches the numbers of non-isomorphic split graphs that we generated.

Nodes in the graph	Split graphs	Non-isomorphic split graphs
2	2	2
3	5	4
4	18	9
5	93	21
6	682	56
7	7 045	164
8	102 050	557
9	2 069 165	2 223
10	58 716 762	10 766

Table 4.1: Table of the number of split graphs generated of a given size with and without the isomorphism test.

As mentioned the isomorphism test that we implemented is trivial, and therefore it is not the fastest way to test for isomorphism. It takes as input two graphs. The algorithm tries to relabel the nodes in the first graph in every possible way, and then compares each of these relabeled graphs to the second graph. If any of these relabeled versions of the first graph is identical to the second graph then the two graphs are isomorphic. If none of the relabeled versions of the first graphs are identical to the second graph then the two graphs are not isomorphic.

This trivial isomorphism test was too slow, so we decided to add a few small improvements to decrease the running time. However we did not run out out memory when we tried to generate all graphs of size 9. We will now briefly describe the improvements that we added. The first improvement that we added was to check if the two graphs had an equal number of edges. If they did not have the same number of edges, they cannot be isomorphic, so if that was the case we did not have to check all possible relabelling of the nodes.

The second improvement that we added was to count the number of nodes with a specified degree from 0 to $n-1$, and check that the two graphs had the same number of nodes with each degree. If they did not have the same number of nodes with each degree from 0 to $n-1$ then they cannot be isomorphic, so again if this is the case then we did not have to check all possible relabelings of the nodes. The third and final improvement that we added, was that before we tried to relabel a node x to a node y in the first graph, we checked that the node with label y in the second graph had the same number of neighbors as the node labeled x in the first graph. This improvement decreased the number of relabeled graphs significantly.

With these improvements added to our isomorphism test we were able to

generate all graphs of size up to 10 in just a couple of hours. We also tried to generate all graphs of size 11. We let this algorithm run for a couple of days, but it did not terminate. This isomorphism test allowed us to generate all graphs of up to size 10, and enumerate all the mcds in these graphs.

There have been some new developments in the graph isomorphism problem recently. Babai has found a quasi-polynomial time algorithm that solves the graph isomorphism problem [3]. A quasi-polynomial time algorithm is slower than a polynomial time algorithm, but faster than an exponential time algorithm. So this does not solve the problem of whether or not the graph isomorphism problem is in P or is NP-complete, but it gives the best known running time for the graph isomorphism problem.

Even with our isomorphism test we were not able to generate all split graphs with 11 or more nodes. Therefore, we renounced the generality of our generation algorithm. Our main purpose in this work is not to be able to generate a set of all split graphs of a given size, but to check for new potential lower bound examples in these graphs. When we changed our approach we were actually able to check for a new lower bound in all graphs of size up to 11 nodes. What we did is that we generated all graphs of size 11 nodes without the isomorphism test. Instead of keeping all of these graphs in a set we just ran the enumeration algorithm for mcds on a graph immediately after it was generated. If it contained a higher number of mcds than the current lower bound we saved the graph, and otherwise we discarded it. Since we did not keep all of the graphs in memory at the same time we could run this algorithm on graphs of up to size 11 nodes and not run out of memory, even though we did not check for isomorphism. However when we tried this approach with 12 nodes in the graph and let it run for a couple of days, it did not finish. Unfortunately none of the graphs of size up to 11 nodes contained a higher number of mcds than the current lower bound.

If we would have written a faster isomorphism test we might have been able to generate all graphs of size 12 or more. We could also have used a graph generation algorithm to generate all non-isomorphic graphs, and pick only the graphs that are split graphs from this set. There exists several such libraries that generate all non-isomorphic graphs of a given size. These graph libraries typically have a faster isomorphism test than the one we implemented, so we would have been able to test more graphs.

The reason why we chose not to go further in this direction is twofold. First of all since the lower and upper bounds on the maximum number of mcds in split graphs are so close, 1.3195^n and 1.3803^n respectively, we did not think that there was a high probability of finding a new lower bound example. Secondly, as we will see in the next chapter our test results convinced us

that the upper bound was too high. Because of the natural time limitations on a master thesis we had to make a choice. Since we were convinced that the upper bound was too high we chose to put our efforts into improving the upper bound on the maximum number of mcds in split graphs. As we will se in the next chapter taking this approach paid off.

Chapter 5

Analyzing the gap between upper and lower bounds

This chapter presents the main contributions of this thesis, namely a significantly better upper bound on the maximum number of mcds in split graphs. We start by reporting on the results we got from experimenting with the enumeration algorithm given by Golovach et al. [11]. First we analyze what percentage of nodes in the clique and in the independent set that gives the highest number of mcds. We suspected that the upper bound might be too high, so to get some confirmation of this we run the algorithm on a lot of different graphs, including the lower bound examples. This gave us an indication that the upper bound might be to high.

Based on the intuition we got from these tests, we study the three rules of the enumeration algorithm given by Golovach et al. [11] that give the highest running time, and thereby the upper bound of 1.3803^n on the maximum number of mcds in split graphs. We adjust all of these rules with the highest running time, to obtain better branching vectors, and consequently correspondingly lower running times for these rules. Our efforts indeed result in a new and better upper bound of 1.3674^n on the maximum number of mcds in split graphs.

5.1 The sizes of the clique and the independent set

We were curious about how the percentage of nodes in the independent set affects the number of mcds found in the graph. Gathering information about

this could make it easier to know what type of split graphs to search for a possible new lower bound example in. We did this by generating graphs of constant size, and varying number of nodes in the independent set.

First we fixed the number of nodes in the graphs to $n = 70$. Then we generated 100 graphs with 5 nodes in the independent set, 100 graphs with 10 nodes in the independent set and so on, up until 65 nodes in the independent set. We calculated the average number of mcds found for each different size of the independent set. This plot is displayed in Figure 5.1.

As we can see in Figure 5.1 the highest number of mcds is found when almost 30 percent of the nodes are in the independent set.

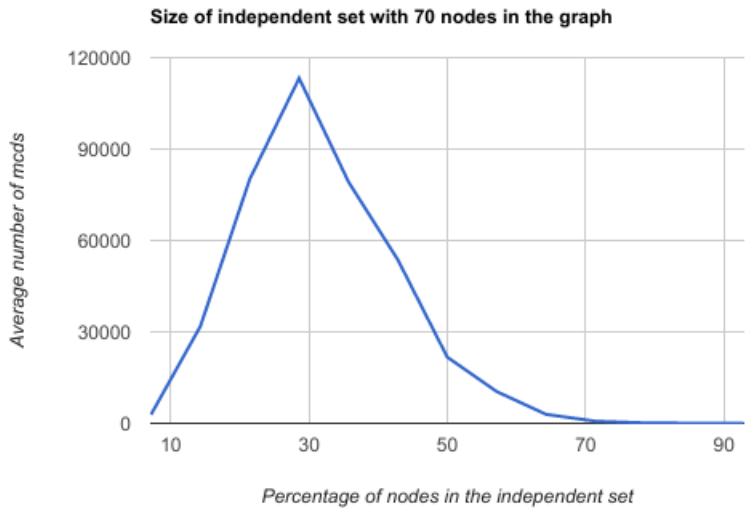

Figure 5.1: A plot of the number of mcds found in graphs of size 70 nodes with varying size of the independent set.

We also did the same with $n = 60$, and $n = 50$, these plots can be seen in Figure 5.2 and Figure 5.3 respectively.

As we can see in these plots, out of all the graphs that we generated, regardless of the number of nodes in these graphs, the highest number of mcds was obtained when the independent set contained almost 30 percent of the nodes and the clique contained slightly more than 70 percent of the nodes.

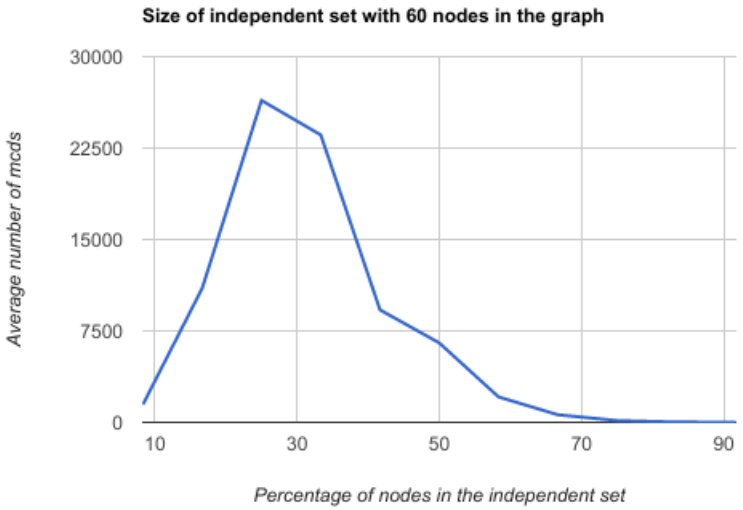

Figure 5.2: A plot of the number of mcds found in graphs of size 60 nodes with varying size of the independent set.

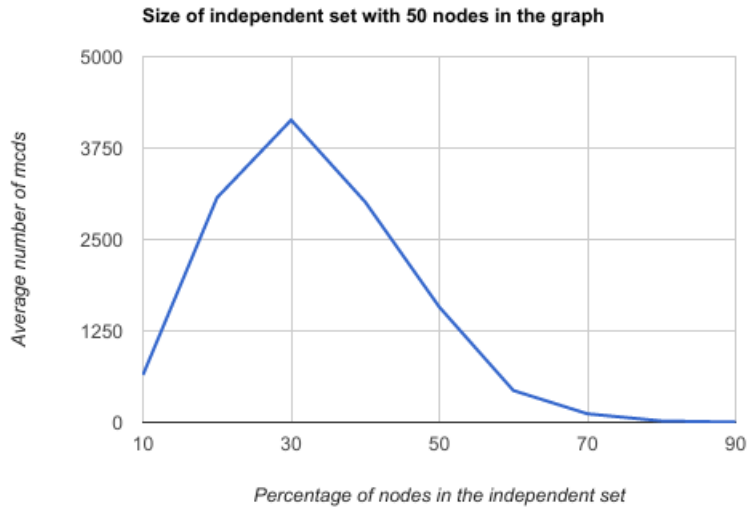

Figure 5.3: A plot of the number of mcds found in graphs of size 50 nodes with varying size of the independent set.

This can be useful is we want to search for a new lower bound, because we know that there is a higher probability of finding a graph with a high number of mcds among the graphs which have approximately 30 percent of their nodes in the independent set. This coincides with our current lower bound examples. The first lower bound example, L_1, contains 5 nodes, where 1 of the nodes is in the independent set, or 20 percent of its nodes are in the independent set. Our second lower bound example, L_2, contains 10 nodes, 3 of which are in the independent set, so 30 percent of its nodes are in the independent set.

5.2 Testing the algorithm

We will now look at some of the results we got from experimenting with the algorithm for enumerating mcds in split graphs given by Golovach et al. [11]. The first thing we did was to generate 100 graphs, with a constant number of nodes, number of edges and size of the clique and independent set. The number of nodes in all these graphs is 60, the number of edges is 940, and the size of the independent set is 20, so that the rest of the 40 nodes are in the clique.

Number of mcds found in the graph	Time (seconds)
105 603	32.572
156 053	29.616
110 015	16.347
179 518	33.434
174 882	26.651
138 714	25.072
139 948	21.709
126 557	19.796

Table 5.1: Table containing the number of mcds found by the algorithm, and the running time of the algorithm for a selection of graphs that we generated. All of the graphs in this table have 60 nodes, with 20 nodes in the independent set and 40 nodes in the clique and 940 edges.

In Table 5.1 we present information about a selection of the graphs that we generated. In some of the graphs shown in the table the running time and the number of mcds found do not seem to match that well. For instance if we compare the first two graphs in Table 5.1 we can see that the algorithm used more time on the first graph, but it found more mcds in the second graph. We can see the same thing if we compare the first graph and the third graph in the table. The algorithm used a lot more time on the first

graph, but it found more mcds in the third graph.

Our initial thought was that the algorithm sometimes discards a lot of leaves in the search tree, which causes the running time to be unnecessarily large in the cases where this happened. Table 5.2 contains information about the same graphs as Table 5.1, but it also contains the number of leaves that were discarded by the algorithm in each of these graphs. The sum of the number of mcds found by the algorithm and the number of leaves that were rejected by the algorithm correspond exactly to the number of leaves in the search tree produced by running the algorithm. Let us again look at the first three graphs, but compare the number of leaves with the running time of the algorithm. The number of leaves seems to match the running time better, so that for a graph a higher number of leaves found by the algorithm would give a higher running time of the algorithm.

In Table 5.2 we can see that the number of rejected leaves seem to be larger than the number of mcds found by the algorithm. This implies that the algorithm discards most of the leaves in the search tree. This might seem like an indication of the fact that the algorithm is not optimal when it comes to running time. It can seem like the algorithm spends most of its time on leaves that get rejected, and we might expect an optimal algorithm to spend most of its time on subsets that are mcds of the graph.

Number of mcds	Number of rejected leaves	Time (seconds)
105 603	604 576	32.572
156 053	471 476	29.616
110 015	163 902	16.347
179 518	508 413	33.434
174 882	343 449	26.651
138 714	288 899	25.072
139 948	317 647	21.709
126 557	292 771	19.796

Table 5.2: Table containing the number of mcds found by the algorithm, the number of leaves that were rejected by the algorithm and the running time of the algorithm for a selection of graphs that we generated. All of the graphs in this table have 60 nodes, with 20 nodes in the independent set and 40 nodes in the clique and 940 edges.

To examine further we can take a look at all of the hundred graphs that we generated, we counted the number of mcds found by the algorithm and the number of leaves that were discarded by the algorithm. As we can see in Figure 5.4 the running time matches the number of leaves pretty well, which corresponds to mcds plus the number of subsets that were rejected at

the base case of the algorithm. This makes sense since the running time is computed by counting the number of leaves in the search tree.

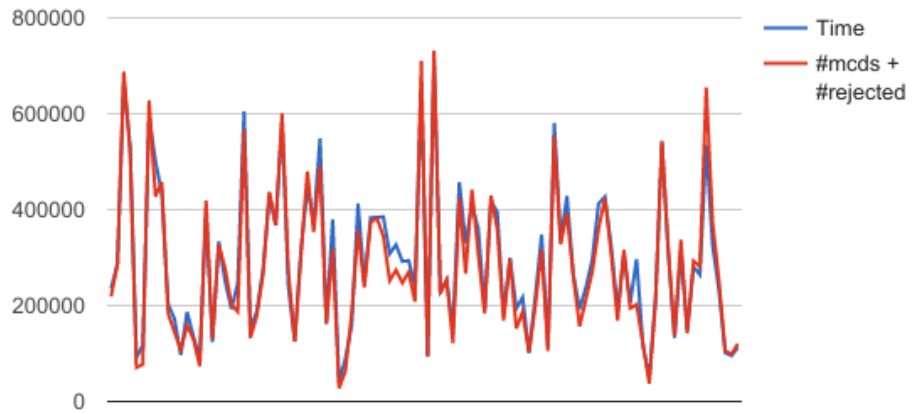

Figure 5.4: A plot comparing the running time and the number of mcds plus the number of subsets that were rejected at the base case found by the algorithm.

If the running time matches the number of leaves pretty well, but does not match the number of mcds, it means that the algorithm discards a lot of leaves. This implies that the algorithm finds a lot of cds that are not minimal. If this is the case then the algorithm might need some improvements to be optimal when it comes to running time, or we might even need to design a new algorithm that does not spend as much time on leaves that are discarded by the algorithm to get a better running time.

Figure 5.5 compares the running time of the algorithm with the number of mcds found by the algorithm. As we can see in Figure 5.5 the number of mcds found by the algorithm matches the running time of the algorithm pretty well, but as we can see in Figure 5.4 the running time of the algorithm matches the number of leaves better. This might be an indicator of the fact that the algorithm discards a lot of leaves, which causes the running time of the algorithm to be unnecessarily large in the cases where this happens.

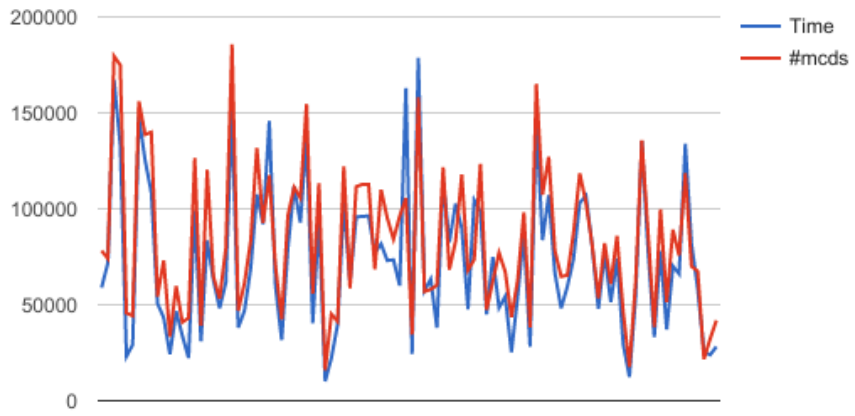

Figure 5.5: A plot comparing the number of mcds found by the algorithm, and the running time of the algorithm.

The problem with analyzing results like these is that it can be hard to interpret something about them. In the graphs that we generated the algorithm discards most of the leaves, and therefore uses most of its time on sets that are not mcds. Even though this might seem like an indicator that the algorithm is not optimal, we cannot draw that conclusion. In fact, any optimal algorithm can discard almost all of the sets it looks at, unless it is run on a graph with a high number of mcds. So the algorithm can still be optimal. We know that when an optimal algorithm runs on a graph containing the highest possible number of mcds it should not discard any leaves. Since we do not know if the upper bound is too high or the lower bound is too low we do not know the highest number of mcds a split graph can contain. The highest number of mcds that we have been able to find in any split graph is given by our two lower bound examples. So we decided to run the algorithm on these graphs, and the expanded version of these graphs and look at the results.

When we ran the algorithm on the lower bound examples and the expanded versions of the lower bound examples, the algorithm did not discard any set, so the running time, and the number of leaves in the search tree produced by running the algorithm in these cases was only dependent on the number mcds found by the algorithm. This is by itself an interesting fact, that the algorithm does not discard any of the leaves when run on the lower bound examples, even though the upper and lower bounds are not equal. The fact that the algorithm did not discard any set when we ran it on the lower bound examples made us belive that the upper bound might be too high.

Thus, we now take a close look at the rules with the highest running time.

5.3 The rules of the algorithm

In the previous two sections, we concentrated on analysis of lower bound examples. Let us now attack the gap from above, and analyze the situations which correspond to the cases of the algorithm giving the upper bound. As mentioned there are three branching rules that dominate the running time of the algorithm for enumerating mcds in split graphs, which gives the current upper bound. We now look at these branching rules with the worst running time and discover the structures in some of the graphs that would use these branching rules. The rules that give the worst running time in the branching algorithm are rules 5.2, 6 and 9.

Figure 5.6 is the graph we get when we try to draw the situations handled by rule 5.2 of the algorithm. It consists of a node x from the clique, with only one neighbor y from the independent set, where y has n neighbors in the clique, and n is at least 3. Every node in the clique except x can have any number of neighbors in the independent set. As we can see Figure 5.6 looks a lot like our first lower bound example, L_1. When $n = 4$ and all of the nodes x_2, x_3, x_4 only have one neighbor in the independent set, we get exactly L_1.

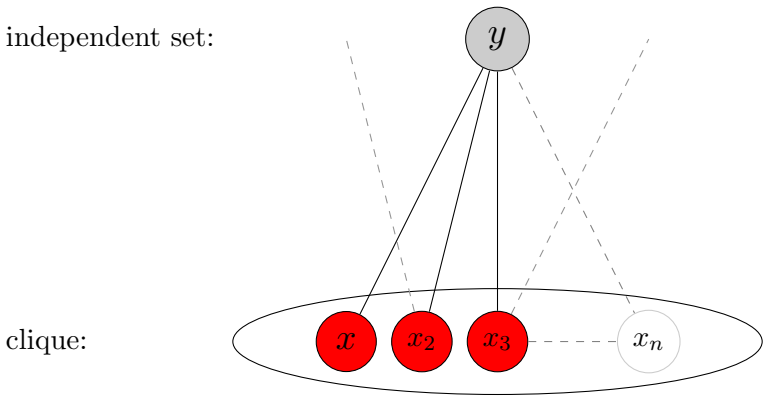

Figure 5.6: Branching rule 5.2.

Figure 5.7 is the graph we get when we draw what happens in rule 6 of the algorithm. It consists of a node x from the clique, with n neighbors in the independent set, $y_1, y_2, ..., y_n$, where $n \geq 3$. All of the nodes $y_1, y_2, ..., y_n$ have at least 2 neighbors in the clique. In our lower bound examples the nodes in the clique all have degree 1 in the first and all have degree 2 in the

second. So none of the lower bound examples that we know of would use this branching rule. We have not been able to find a lower bound example that looks like this graph, but there might exist one.

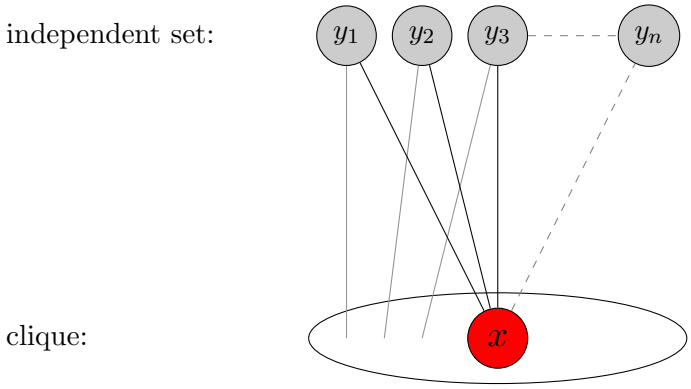

Figure 5.7: Branching rule 6.

Figure 5.8 is the graph we get when we draw rule 9 of the algorithm. It consists of a node y from the independent set with exactly 4 neighbors in the clique, x_1, x_2, x_3, x_4. All of the nodes in the clique, including x_1, x_2, x_3, x_4 have exactly two neighbors in the independent set. This graph looks quite similar to our second lower bound example, L_2. If for example node w_1 and w_2 are merged into one node, and w_3 and w_4 are also merged into one node, then we get exactly L_2.

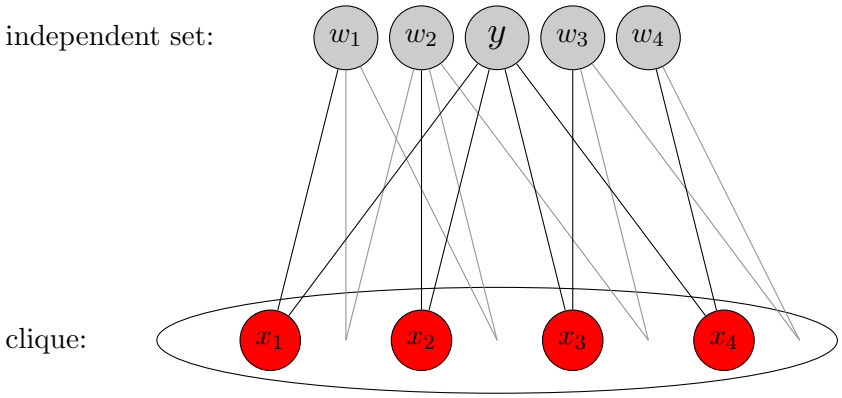

Figure 5.8: Branching rule 9.

The fact that two of the worst rules, 5.2 and 9, seem to concern graphs that give our current lower bound made us think that there is room for improvement on these rules. In fact, studying all worst time rules carefully,

we could see that improvements are possible on all of them. We present these findings in the next sections.

5.4 Proposal for new rules in the algorithm

As mentioned, rules 5.1, 6 and 9 in the enumeration algorithm given by Golovach et al. [11] determine the running time of the algorithm. If we are able to decrease the running time of these rules, or somehow remove all of these rules, then we would end up with an algorithm that has a lower running time, which would provide us with a better upper bound.

We start by looking at rule 5 of Algorithm 3.1, shown here in Algorithm 5.1. It is the test in rule 5.2 that gives the upper bound of the algorithm.

Algorithm 5.1 Rule five of the enumeration algorithm given in Algorithm 3.1

1: **if** there is an $x \in K$ such that $d_S(x) = 1$ and y is the unique neighbor of x in S. Let $N_H(y) = \{x, x_2, ..., x_t\}$ for $t \geq 2$. **then** ▷ 5
2: | **if** t == 2 **then** ▷ 5.1
3: | | ENUMCDS($K \setminus \{x, x_2\}, S \setminus \{y\}, X \cup \{x\}$)
4: | | ENUMCDS($K \setminus \{x, x_2\}, S \setminus \{N_H(x_2) \cap S\}, X \cup \{x_2\}$)
5: | **end if**
6: | **if** $t > 2$ **then** ▷ 5.2
7: | | ENUMCDS($K \setminus \{x, x_2, x_3, ..., x_t\}, S \setminus \{y\}, X \cup \{x\}$)
8: | | ENUMCDS($K \setminus \{x\}, S, X$)
9: | **end if**
10: **end if**

Let us examine how we can change this rule to obtain a lower branching vector for this rule. In rule 5 we check if there is a node in the clique with exactly one neighbor y, in the independent set. Let t be equal to the degree of y. If $t = 2$ we use rule 5.1, else if $t \geq 3$ we use rule 5.2. One way to change this rule is to change rule 5.2 to only be executed if $t = 3$, and then add a rule 5.3 that is executed if $t \geq 4$. We will also need to change the content of the new rule 5.2, and write the content of rule 5.3. Algorithm 5.2 shows one possible way to reformulate rule 5, as discussed above.

Let us argue for the correctness of the new rule 5 shown in Algorithm 5.2. First we can check that all cases are handled. At this point in the algorithm we know that all nodes y in the independent set have $d_K(y) \geq 2$ since otherwise rule 4 would have been applied. Rule 5.1 handles the cases where $d_K(y) = 2$, 5.2 handles all cases where $d_K(y) = 3$ and 5.3 handles all cases where $d_K(y) > 3$. So we handle all possible degrees that y can have.

Algorithm 5.2 New version of rule 5 of the enumeration algorithm

1: **if** If there is an $x \in K$ such that $d_S(x) = 1$ and y is the unique neighbor of x in S. Let $N_H(y) = \{x, x_2, ..., x_t\}$ for $t \geq 2$. **then** ▷ 5
2: | **if** t == 2 **then** ▷ 5.1
3: | | ENUMCDS$(K \setminus \{x, x_2\}, S \setminus \{y\}, X \cup \{x\})$
4: | | ENUMCDS$(K \setminus \{x, x_2\}, S \setminus \{N_H(x_2) \cap S\}, X \cup \{x_2\})$
5: | **end if**
6: | **if** $t == 3$ **then** ▷ 5.2
7: | | **if** $d_S(x_2) = d_S(x_3) = 1$ **then** ▷ 5.2.1
8: | | | ENUMCDS$(K \setminus \{x, x_2, x_3\}, S \setminus \{y\}, X \cup \{x\})$
9: | | | ENUMCDS$(K \setminus \{x, x_2, x_3\}, S \setminus \{y\}, X \cup \{x_1\})$
10: | | | ENUMCDS$(K \setminus \{x, x_2, x_3\}, S \setminus \{y\}, X \cup \{x_2\})$
11: | | **end if**
12: | | **if** $d_S(x_2) = 1$ and $d_S(x_3) \geq 2$ **then** ▷ 5.2.2
13: | | | ENUMCDS$(K \setminus \{x, x_2, x_3\}, S \setminus \{N_H(x_3)\}, X \cup \{x_3\})$
14: | | | ENUMCDS$(K \setminus \{x, x_2, x_3\}, S \setminus \{y\}, X \cup \{x_2\})$
15: | | | ENUMCDS$(K \setminus \{x, x_2, x_3\}, S \setminus \{y\}, X \cup \{x\})$
16: | | **end if**
17: | | **if** $d_S(x_2) \geq 2$ and $d_S(x_3) \geq 2$ **then** ▷ 5.2.3
18: | | | ENUMCDS$(K \setminus \{x, x_3\}, S \setminus \{N_H(x_3)\}, X \cup \{x_3\})$
19: | | | ENUMCDS$(K \setminus \{x, x_2, x_3\}, S \setminus \{N_H(x_2)\}, X \cup \{x_2\})$
20: | | | ENUMCDS$(K \setminus \{x, x_2, x_3\}, S \setminus \{y\}, X \cup \{x\})$
21: | | **end if**
22: | **end if**
23: | **if** $t > 3$ **then** ▷ 5.3
24: | | ENUMCDS$(K \setminus \{x, x_2, x_3, ..., x_t\}, S \setminus \{y\}, X \cup \{x\})$
25: | | ENUMCDS$(K \setminus \{x\}, S, X)$
26: | **end if**
27: **end if**

We also need to check that we handle all cases inside 5.2, since this rule is split into three different cases. We know that $d_S(x) = 1$, but we need to handle every possible combination of degrees for x_2 and x_3. In 5.2.1 we handle the cases where $d_S(x_2) = d_S(x_3) = 1$. In 5.2.2 we handle the cases where one of x_2 and x_3 has degree 1 in S, and the other has degree ≥ 2 in S, and we let x_3 be the one with degree ≥ 2 in S. In 5.2.3 we handle all cases where $d_S(x_2) \geq 2$ and $d_S(x_3) \geq 2$. So inside rule 5.2 we handle all possible degrees of x_2 and x_3.

Let us start by checking the correctness of rule 5.2.1. If this rule is used we have $N_H(y) = \{x, x_2, x_3\}$, and $d_S(x) = 1$, and for $i = 2, 3$ we have $d_S(x_i) = 1$. Since x, x_1 and x_2 in this case dominates the exact same set of nodes we know that none of them can be in the same solution. So if x is added to the solution, we can discard x_2 and x_3. The same applies for x_2 and x_3.

In rule 5.2.2 we know that $N_H(y) = \{x, x_2, x_3\}$ and $d_S(x) = 1$, $d_S(x_2) = 1$, and $d_S(x_3) \geq 2$. Again none of x, x_1 or x_2 can be in the same solution since x and x_2 dominate the exact same set of nodes, which is a subset of the nodes that x_3 dominates. So if one of x, x_2 or x_3, is added to the solution, the others can be discarded.

In rule 5.2.3 we know that $N_H(y) = \{x, x_2, x_3\}$ and $d_S(x) = 1$, and for $i = 2, 3$ we have $d_S(x_i) \geq 2$. Note that in this case x_2 and x_3 can actually be in the same solution, since they might dominate a different set of nodes. This is handled by the fact that when we add x_3 to the solution, we do not discard x_2.

Let us now analyze the running time of the new rule 5 by looking at the branching vectors in each of its steps.

5.1: (3,3), which gives a running time of $\mathcal{O}(1.2600^n)$.
5.2.1: (4,4,4), which gives a running time of $\mathcal{O}(1.3161^n)$.
5.2.2: (5,4,4), which gives a running time of $\mathcal{O}(1.2907^n)$.
5.2.3: (4,5,4), which gives a running time of $\mathcal{O}(1.2907^n)$.
5.3: (5,1), which gives a running time of $\mathcal{O}(1.3248^n)$.

As we can see, all of the steps in the new formulation of rule 5 have a lower running time than the old rule 5, which has a running time of $\mathcal{O}(1.3803^n)$. The step in the new rule 5 with the highest running time is step 5.3, with a running time of $\mathcal{O}(1.3248^n)$, so if we replace rule 5 in the main algorithm with the new rule 5, we decrease the running time of rule 5 from $\mathcal{O}(1.3803^n)$ to $\mathcal{O}(1.3248^n)$, and this rule no longer gives the highest running time of the algorithm. Note that the new running time of rule 5 is still higher than the lower bound, which is 1.3195^n, but the gap is significantly smaller.

Algorithm 5.3 Rule 6 of the enumeration algorithm given in Algorithm 3.1

```
1: if there is an x ∈ K such that d_S(x) ≥ 3. Let N_H(x)∩S = {y_1, y_2, ..., y_t}
   and t ≥ 3 then                                                    ▷ 6
2:     ENUMCDS(K \ {x}, S \ {y_1, y_2, ...y_t}, X ∪ {x})
3:     ENUMCDS(K \ {x}, S, X)
4: end if
```

We will now take a closer look at rule 6. It is displayed in Algorithm 5.3. This rule is one of the three rules that dictate the upper bound of Algorithm 3.1. We were able to change rule 5 to get a better running time, so with these changes added to the algorithm it is only rule 6 and rule 9 that determine the upper bound of Algorithm 3.1. As we can see in Algorithm 5.3, rule 6 is used if there is a node in the clique with a degree higher than 2. One way to change this rule is displayed in Algorithm 5.4.

Algorithm 5.4 New version of rule 6 of the enumeration algorithm

1: **if** there is an $x \in K$ such that $d_S(x) \geq 4$. Let $N_H(x) \cap S = \{y_1, y_2, ..., y_t\}$ and $t \geq 3$ **then** ▷ 6
2: | ENUMCDS$(K \setminus \{x\}, S \setminus \{y_1, y_2, ...y_t\}, X \cup \{x\})$
3: | ENUMCDS$(K \setminus \{x\}, S, X)$
4: **end if**

Let us look at the branching vector of the new formulation of rule 6:

6: (5,1), which gives a running time of $\mathcal{O}(1.3248^n)$.

As we can see this new rule 6 has a lower running time than the old rule 6. So if we can make this change to rule 6 we have decreased the running time of this rule. Note that when we changed rule 5 we did not change which cases it handled. In our new formulation of rule 6 we actually also change which cases it handles. Instead of using this rule when there is a node in the clique with degree more than two, the new rule 6 is used if there is a node in the clique with degree more than three. This implies that we also need to rewrite all rules after rule 6 to not only handle the cases when a node in the clique has 2 neighbors in S, but also when a node in the clique has 3 neighbors in S. Thus if we are going to replace the old rule 6 with the new rule 6, we have to analyze and possibly change all of the rules in Algorithm 3.1 that are after rule 6.

The rules that are after rule 6 in Algorithm 3.1 are rules 7, 8, 9 and 10. Note that what we need to do is to make sure that we handle every case where the nodes in K have degree 2 or 3 in S, instead of just degree 2 in S as these rules currently do. We know that for each node $c \in K$, $d_S(c) = 2$ or $d_S(c) = 3$, since otherwise rule 3, 5 or 6 would have been applied.

Algorithm 5.5 Rule 7 of the enumeration algorithm given in Algorithm 3.1

1: **if** there is a $y \in S$ such that $d_K(y) = 2$ then let $N_H(y) = \{x_1, x_2\}$ and for all $i = 1, 2$ let w_i be the unique neighbor of x_i in S different from y. **then** ▷ 7
2: | ENUMCDS$(K \setminus \{x_1\}, S \setminus \{y, w_1\}, X \cup \{x_1\})$
3: | ENUMCDS$(K \setminus \{x_1, x_2\}, S \setminus \{y, w_2\}, X \cup \{x_2\})$
4: **end if**

Rule 7 in Algorithm 3.1 is displayed in Algorithm 5.5. We can see that if we change this rule to allow nodes in the clique to have degree two or three instead of just degree two, the branching vector of this rule would be the same as in the current rule 7. This is due to the fact that the worst case is when all nodes in the clique have degree 2. This new rule 7 is displayed in Algorithm 5.6.

Algorithm 5.6 New version of rule 7 of the enumeration algorithm

1: **if** there is a $y \in S$ such that $d_K(y) = 2$ then let $N_H(y) = \{x_1, x_2\}$. **then** ▷ 7
2: | ENUMCDS($K \setminus \{x_1\}, S \setminus \{N_H(x_1) \cap S\}, X \cup \{x_1\}$)
3: | ENUMCDS($K \setminus \{x_1, x_2\}, S \setminus \{N_H(x_2) \cap S\}, X \cup \{x_2\}$)
4: **end if**

The branching vector of this new rule 7 is:

7: (3,4), which gives a running time of $\mathcal{O}(1.2208^n)$.

Algorithm 5.7 Rule 8 and rule 9 of the enumeration algorithm given in Algorithm 3.1

1: **if** If there is a $y \in S$ such that $d_K(y) = 3$ then let $N_H(y) = \{x_1, x_2, x_3\}$ and for all $i = 1, 2, 3$ let w_i be the unique neighbor of x_i in S different from y; **then** ▷ 8
2: | ENUMCDS($K \setminus \{x_1\}, S \setminus \{y, w_1\}, X \cup \{x_1\}$)
3: | ENUMCDS($K \setminus \{x_1, x_2\}, S \setminus \{y, w_2\}, X \cup \{x_2\}$)
4: | ENUMCDS($K \setminus \{x_1, x_2, x_3\}, S \setminus \{y, w_3\}, X \cup \{x_3\}$)
5: **end if**
6: **if** there is a $y \in S$ such that $d_K(y) = 4$ then let $N_H(y) = \{x_1, x_2, x_3, x_4\}$ and for all $i = 1, 2, 3, 4$ let w_i be the unique neighbor of x_i in S different from y; **then** ▷ 9
7: | ENUMCDS($K \setminus \{x_1\}, S \setminus \{y, w_1\}, X \cup \{x_1\}$)
8: | ENUMCDS($K \setminus \{x_1, x_2\}, S \setminus \{y, w_2\}, X \cup \{x_2\}$)
9: | ENUMCDS($K \setminus \{x_1, x_2, x_3\}, S \setminus \{y, w_3\}, X \cup \{x_3\}$)
10: | ENUMCDS($K \setminus \{x_1, x_2, x_3, x_4\}, S \setminus \{y, w_4\}, X \cup \{x_4\}$)
11: **end if**

The same applies for rule 8 and rule 9. We can change these rules to handle the cases when the nodes in the clique have degree two or three without changing the branching vectors of these rules, and thereby keep the same running time in these rules. This is because we again obtain the highest running time when the nodes in the clique have 2 neighbors in S. The current versions of these rules are displayed in Algorithm 5.7, and the new version of these two rules are displayed in Algorithm 5.8.

The branching vectors of these two new rules are given by:

8: (3,4,5), which gives a running time of $\mathcal{O}(1.3248^n)$.
9: (3,4,5,6), which gives a running time of $\mathcal{O}(1.3803^n)$.

Algorithm 5.8 New version of rule 8 and rule 9 of the enumeration algorithm

if If there is a $y \in S$ such that $d_K(y) = 3$ then let $N_H(y) = \{x_1, x_2, x_3\}$. **then** ▷ 8
  ENUMCDS$(K \setminus \{x_1\}, S \setminus \{N_H(x_1) \cap S\}, X \cup \{x_1\})$
  ENUMCDS$(K \setminus \{x_1, x_2\}, S \setminus \{N_H(x_2) \cap S\}, X \cup \{x_2\})$
  ENUMCDS$(K \setminus \{x_1, x_2, x_3\}, S \setminus \{N_H(x_3) \cap S\}, X \cup \{x_3\})$
end if
if there is a $y \in S$ such that $d_K(y) = 4$ then let $N_H(y) = \{x_1, x_2, x_3, x_4\}$. **then** ▷ 9
  ENUMCDS$(K \setminus \{x_1\}, S \setminus \{N_H(x_1) \cap S\}, X \cup \{x_1\})$
  ENUMCDS$(K \setminus \{x_1, x_2\}, S \setminus \{N_H(x_2) \cap S\}, X \cup \{x_2\})$
  ENUMCDS$(K \setminus \{x_1, x_2, x_3\}, S \setminus \{N_H(x_3) \cap S\}, X \cup \{x_3\})$
  ENUMCDS$(K \setminus \{x_1, x_2, x_3, x_4\}, S \setminus \{N_H(x_4) \cap S\}, X \cup \{x_4\})$
end if

Algorithm 5.9 Rule 10 of the enumeration algorithm given in Algorithm 3.1

if there is an $x \in K$ with neighbors y and y' in S then $d_K(y) \geq 5$ and $d_K(y') \geq 5$ **then** ▷ 10
  ENUMCDS$(K \setminus N_H(y), S \setminus \{y, y'\}, X \cup \{x\})$
  ENUMCDS$(K \setminus N_H(y'), S \setminus \{y, y'\}, X \cup \{x\})$
  ENUMCDS$(K \setminus \{x\}, S, X)$
end if

Rule 10 however, is only applicable when the node in the clique has degree 2. We can try to create a similar rule to handle the same cases if the node in the clique has degree 3. Rule 10 of Algorithm 3.1 is displayed in Algorithm 5.9.

The branching vector of this rule is:

10: (7,7,1), which gives a running time of $\mathcal{O}(1.3422^n)$.

Algorithm 5.10 A new rule similar to rule 10 in Algorithm 3.1, except it handles the cases where a node in the clique has three neighbors in S instead of two neighbors in S

if there is an $x \in K$ with neighbors y_1, y_2 and y_3 in S then $d_K(y_1) \geq 5$, $d_K(y_2) \geq 5$ and $d_K(y_3) \geq 5$ **then** ▷ 11
  ENUMCDS$(K \setminus N_H(y_1), S \setminus \{y_1, y_2, y_3\}, X \cup \{x\})$
  ENUMCDS$(K \setminus N_H(y_2), S \setminus \{y_1, y_2, y_3\}, X \cup \{x\})$
  ENUMCDS$(K \setminus N_H(y_3), S \setminus \{y_1, y_2, y_3\}, X \cup \{x\})$
  ENUMCDS$(K \setminus \{x\}, S, X)$
end if

If we try to make a similar rule for a node in the clique with three neighbors in S, we would get the rule displayed in Algorithm 5.10.

This rule has a branching vector of:

11: (8,8,8,1), which gives a running time of $\mathcal{O}(1.3560^n)$.

The running time of this rule is quite high, even though it is not as high as the running time we get from rule 9. At this point in the algorithm we know that all nodes in the clique have three neighbors in S, since otherwise rule 5, 6, 7, 8, 9, or 10 would have been applied. We also know that all nodes in S have degree ≥ 5. We can try to change this rule to get a better running time by doing something similar to what we did in rule 7, 8 and 9. We can change rule 10 and rule 11 to the algorithm displayed in Algorithm 5.11.

These rules have branching vectors of:

10: (7,7,1), which gives a running time of $\mathcal{O}(1.3422^n)$.
11: (4,5,6,7,8), which gives a running time of $\mathcal{O}(1.3248^n)$.
12: (9,9,9,1), which gives a running time of $\mathcal{O}(1.3219^n)$.

Algorithm 5.11 New version of rule 10 in the enumeration algorithm, and new rules labeled 11 and 12 to handle the cases where a node in the clique has 3 neighbors in S

1: **if** there is an $x \in K$ such that $d_{\mathcal{S}}(x) = 2$, with neighbors y and y' in S then $d_K(y) \geq 5$ and $d_K(y') \geq 5$ **then** ▷ 10
2: | ENUMCDS$(K \setminus N_H(y), S \setminus \{y, y'\}, X \cup \{x\})$
3: | ENUMCDS$(K \setminus N_H(y'), S \setminus \{y, y'\}, X \cup \{x\})$
4: | ENUMCDS$(K \setminus \{x\}, S, X)$
5: **end if**
6: **if** there is a $y \in S$ such that $d_K(y) = 5$ then let $N_H(y) = \{x_1, x_2, x_3, x_4, x_5\}$. **then** ▷ 11
7: | ENUMCDS$(K \setminus \{x_1\}, S \setminus \{N_H(x_1) \cap S\}, X \cup \{x_1\})$
8: | ENUMCDS$(K \setminus \{x_1, x_2\}, S \setminus \{N_H(x_2) \cap S\}, X \cup \{x_2\})$
9: | ENUMCDS$(K \setminus \{x_1, x_2, x_3\}, S \setminus \{N_H(x_3) \cap S\}, X \cup \{x_3\})$
10: | ENUMCDS$(K \setminus \{x_1, x_2, x_3, x_4\}, S \setminus \{N_H(x_4) \cap S\}, X \cup \{x_4\})$
11: | ENUMCDS$(K \setminus \{x_1, x_2, x_3, x_4, x_5\}, S \setminus \{N_H(x_5) \cap S\}, X \cup \{x_5\})$
12: **end if**
13: **if** there is an $x \in K$ such that $d_{\mathcal{S}}(x) = 3$, with neighbors y_1, y_2, y_3 in S then for $i = 1, 2, 3$ $d_K(y_i) \geq 6$ **then** ▷ 12
14: | ENUMCDS$(K \setminus N_H(y_1), S \setminus \{y_1, y_2, y_3\}, X \cup \{x\})$
15: | ENUMCDS$(K \setminus N_H(y_2), S \setminus \{y_1, y_2, y_3\}, X \cup \{x\})$
16: | ENUMCDS$(K \setminus N_H(y_3), S \setminus \{y_1, y_2, y_3\}, X \cup \{x\})$
17: | ENUMCDS$(K \setminus \{x\}, S, X)$
18: **end if**

We will now give a short explanation of the correctness of the new steps added to Algorithm 5.11.

In rule 11 we know that all $x \in K$ have $d_S(x) = 3$, since otherwise rule 3, 5, 6, 7, 8, 9 or 10 would have been applied and all $y \in S$ have $d_K(y) \geq 5$ since otherwise rule 4, 7, 8 or 9 would have been applied. To dominate y we have to check all possible solutions containing any combination of the nodes in $N_H(y)$. We do this by first adding x_1 to the solution without discarding any of the nodes in $N_H(y)$, in this way all combinations containing the node x_1 are checked. Therefore when we add x_2 to the solution we can discard the node x_1 from K, since we have already handled every possible solution that contains x_1, and so on.

In rule 12 we know that all $x \in K$ have $d_S(x) = 3$ since otherwise rule 3, 5, 6, 7, 8, 9 or 10 would have been applied, and all $y \in S$ have $d_K(y) \geq 6$ since otherwise rule 5 or 4, 7, 8, 9, 10 or 11 would have been applied. We have to handle all cases where x is in the solution, and all cases where x is not in the solution. We know that if x is in the solution, x has to be the only node that dominates one of its neighbors. For each of the nodes y in $N_H(x) \cap S$ we discard $N_H(y)$, and use x to dominate y. Otherwise we discard x from the solution.

We have now successfully changed both rule 5 and rule 6, which previously were two of the three rules in Algorithm 3.1 that gave the highest running times. The only rule that has the highest running time of $\mathcal{O}(1.3803^n)$ in Algorithm 3.1 after these new changes have been added is rule 9. If we are able to change this rule to get a better running time we will decrease the running time of the whole algorithm, and thereby decrease the upper bound on the maximum number of mcds in split graphs.

Before we go on to analyze rule 9 in detail, we give the new version of the algorithm with the proposed changes so far. This new version of the algorithm is displayed in Algorithm 5.12. Rule 9 is analyzed in a section on its own, because its improvement requires going into many more different cases.

Algorithm 5.12 Algorithm for enumerating mcds in split graphs

function ENUMCDS(K, S, X):
  if X is a minimal connected dominating set of G **then** ▷ 1
    return X and stop
  end if
  if X is a connected dominating set of G but not minimal **then** ▷ 2
    stop
  end if
  if there is an $x \in K$ such that $d_S(x) = 0$ **then** ▷ 3
    ENUMCDS$(K \setminus \{x\}, S, X)$
  end if
  if there is a $y \in S$ such that $d_K(y) = 1$ and x is the unique neighbor of y in H **then** ▷ 4
    ENUMCDS$(K \setminus \{x\}, S \setminus \{N_H(x) \cap S\}, X \cup \{x\})$
  end if
  if If there is an $x \in K$ such that $d_S(x) = 1$ and y is the unique neighbor of x in S. Let $N_H(y) = \{x, x_2, ..., x_t\}$ for $t \geq 2$. **then** ▷ 5
    if t == 2 **then** ▷ 5.1
      ENUMCDS$(K \setminus \{x, x_2\}, S \setminus \{y\}, X \cup \{x\})$
      ENUMCDS$(K \setminus \{x, x_2\}, S \setminus \{N_H(x_2) \cap S\}, X \cup \{x_2\})$
    end if
    if $t == 3$ **then** ▷ 5.2
      if $d_S(x_2) = d_S(x_3) = 1$ **then** ▷ 5.2.1
        ENUMCDS$(K \setminus \{x, x_2, x_3\}, S \setminus \{y\}, X \cup \{x\})$
        ENUMCDS$(K \setminus \{x, x_2, x_3\}, S \setminus \{y\}, X \cup \{x_1\})$
        ENUMCDS$(K \setminus \{x, x_2, x_3\}, S \setminus \{y\}, X \cup \{x_2\})$
      end if
      if $d_S(x_2) = 1$ and $d_S(x_3) \geq 2$ **then** ▷ 5.2.2
        ENUMCDS$(K \setminus \{x, x_2, x_3\}, S \setminus \{N_H(x_3)\}, X \cup \{x_3\})$
        ENUMCDS$(K \setminus \{x, x_2, x_3\}, S \setminus \{y\}, X \cup \{x_2\})$
        ENUMCDS$(K \setminus \{x, x_2, x_3\}, S \setminus \{y\}, X \cup \{x\})$
      end if
      if $d_S(x_2) \geq 2$ and $d_S(x_3) \geq 2$ **then** ▷ 5.2.3
        ENUMCDS$(K \setminus \{x, x_3\}, S \setminus \{N_H(x_3)\}, X \cup \{x_3\})$
        ENUMCDS$(K \setminus \{x, x_2, x_3\}, S \setminus \{N_H(x_2)\}, X \cup \{x_2\})$
        ENUMCDS$(K \setminus \{x, x_2, x_3\}, S \setminus \{y\}, X \cup \{x\})$
      end if
    end if
    if $t > 3$ **then** ▷ 5.3
      ENUMCDS$(K \setminus \{x, x_2, x_3, ..., x_t\}, S \setminus \{y\}, X \cup \{x\})$
      ENUMCDS$(K \setminus \{x\}, S, X)$
    end if
  end if

41: | **if** there is an $x \in K$ such that $d_S(x) \geq 4$. Let $N_H(x) \cap S = \{y_1, y_2, ..., y_t\}$ and $t \geq 3$ **then** ▷ 6
42: | | ENUMCDS$(K \setminus \{x\}, S \setminus \{y_1, y_2, ...y_t\}, X \cup \{x\})$
43: | | ENUMCDS$(K \setminus \{x\}, S, X)$
44: | **end if**
45: | **if** there is a $y \in S$ such that $d_K(y) = 2$ then let $N_H(y) = \{x_1, x_2\}$. **then** ▷ 7
46: | | ENUMCDS$(K \setminus \{x_1\}, S \setminus \{N_H(x_1) \cap S\}, X \cup \{x_1\})$
47: | | ENUMCDS$(K \setminus \{x_1, x_2\}, S \setminus \{N_H(x_2) \cap S\}, X \cup \{x_2\})$
48: | **end if**
49: | **if** If there is a $y \in S$ such that $d_K(y) = 3$ then let $N_H(y) = \{x_1, x_2, x_3\}$. **then** ▷ 8
50: | | ENUMCDS$(K \setminus \{x_1\}, S \setminus \{N_H(x_1) \cap S\}, X \cup \{x_1\})$
51: | | ENUMCDS$(K \setminus \{x_1, x_2\}, S \setminus \{N_H(x_2) \cap S\}, X \cup \{x_2\})$
52: | | ENUMCDS$(K \setminus \{x_1, x_2, x_3\}, S \setminus \{N_H(x_3) \cap S\}, X \cup \{x_3\})$
53: | **end if**
54: | **if** there is a $y \in S$ such that $d_K(y) = 4$ then let $N_H(y) = \{x_1, x_2, x_3, x_4\}$. **then** ▷ 9
55: | | ENUMCDS$(K \setminus \{x_1\}, S \setminus \{N_H(x_1) \cap S\}, X \cup \{x_1\})$
56: | | ENUMCDS$(K \setminus \{x_1, x_2\}, S \setminus \{N_H(x_2) \cap S\}, X \cup \{x_2\})$
57: | | ENUMCDS$(K \setminus \{x_1, x_2, x_3\}, S \setminus \{N_H(x_3) \cap S\}, X \cup \{x_3\})$
58: | | ENUMCDS$(K \setminus \{x_1, x_2, x_3, x_4\}, S \setminus \{N_H(x_4) \cap S\}, X \cup \{x_4\})$
59: | **end if**
60: | **if** there is an $x \in K$ such that $d_{\mathcal{S}}(x) = 2$, with neighbors y and y' in S then $d_K(y) \geq 5$ and $d_K(y') \geq 5$ **then** ▷ 10
61: | | ENUMCDS$(K \setminus N_H(y), S \setminus \{y, y'\}, X \cup \{x\})$
62: | | ENUMCDS$(K \setminus N_H(y'), S \setminus \{y, y'\}, X \cup \{x\})$
63: | | ENUMCDS$(K \setminus \{x\}, S, X)$
64: | **end if**
65: | **if** there is a $y \in S$ such that $d_K(y) = 5$ then let $N_H(y) = \{x_1, x_2, x_3, x_4, x_5\}$. **then** ▷ 11
66: | | ENUMCDS$(K \setminus \{x_1\}, S \setminus \{N_H(x_1) \cap S\}, X \cup \{x_1\})$
67: | | ENUMCDS$(K \setminus \{x_1, x_2\}, S \setminus \{N_H(x_2) \cap S\}, X \cup \{x_2\})$
68: | | ENUMCDS$(K \setminus \{x_1, x_2, x_3\}, S \setminus \{N_H(x_3) \cap S\}, X \cup \{x_3\})$
69: | | ENUMCDS$(K \setminus \{x_1, x_2, x_3, x_4\}, S \setminus \{N_H(x_4) \cap S\}, X \cup \{x_4\})$
70: | | ENUMCDS$(K \setminus \{x_1, x_2, x_3, x_4, x_5\}, S \setminus \{N_H(x_5) \cap S\}, X \cup \{x_5\})$
71: | **end if**
72: | **if** there is an $x \in K$ such that $d_{\mathcal{S}}(x) = 3$, with neighbors y_1, y_2, y_3 in S then for $i = 1, 2, 3$ $d_K(y_i) \geq 6$ **then** ▷ 12
73: | | ENUMCDS$(K \setminus N_H(y_1), S \setminus \{y_1, y_2, y_3\}, X \cup \{x\})$
74: | | ENUMCDS$(K \setminus N_H(y_2), S \setminus \{y_1, y_2, y_3\}, X \cup \{x\})$
75: | | ENUMCDS$(K \setminus N_H(y_3), S \setminus \{y_1, y_2, y_3\}, X \cup \{x\})$
76: | | ENUMCDS$(K \setminus \{x\}, S, X)$
77: | **end if**
78: **end function**

5.5 Improving rule 9

In this section we will propose changes to rule 9 of the algorithm to obtain a better running time for this rule. The rules that we create also replace rule 10 of the algorithm. As mentioned, this would decrease the running time of the algorithm for enumerating mcds in split graphs, and thereby decrease the upper bound on the maximum number of mcds in split graphs. Out of all the rules in Algorithm 5.12 the rules with the highest running time, except from rule 9 and 10, is rule 5.3, 6, 8 and 11 with a running time of $\mathcal{O}(1.3248^n)$. The rules that we will create are very specific, and not very implementation friendly, but they are meant more as a proof of the fact that the upper bound is too high, than to be rules used in our main algorithm.

From here on, for each node c in the clique we will use $N_S(c)$ to denote $N_H(c) \cap S$, as a simplification.

Let us take a closer look at rule 9 of Algorithm 5.12, it is displayed in Algorithm 5.13.

Algorithm 5.13 Rule 9 of the enumeration Algorithm 5.12

1: **if** there is a $y \in S$ such that $d_K(y) = 4$ then let $N_H(y) = \{x_1, x_2, x_3, x_4\}$. **then** ▷ 9
2: ENUMCDS$(K \setminus \{x_1\}, S \setminus \{N_H(x_1) \cap S\}, X \cup \{x_1\})$
3: ENUMCDS$(K \setminus \{x_1, x_2\}, S \setminus \{N_H(x_2) \cap S\}, X \cup \{x_2\})$
4: ENUMCDS$(K \setminus \{x_1, x_2, x_3\}, S \setminus \{N_H(x_3) \cap S\}, X \cup \{x_3\})$
5: ENUMCDS$(K \setminus \{x_1, x_2, x_3, x_4\}, S \setminus \{N_H(x_4) \cap S\}, X \cup \{x_4\})$
6: **end if**

As mentioned rule 9 has a branching vector of:

9: (3,4,5,6), which gives a running time of $\mathcal{O}(1.3803^n)$.

If any of the nodes among y's neighbors have degree 3, let this node be x_1, and the rest of y's neighbors be x_2, x_3, x_4. If rule 9 is used in this case, we get a branching vector of: (4,4,5,6), which gives a running time of $\mathcal{O}(1.3472^n)$.

So, as long as at least one of $x_1, .., x_4$ have degree 3, the running time of rule 9 is $\mathcal{O}(1.3472^n)$. If all of $x_1, ..., x_4$ have degree 2, the running time of rule 9 is $\mathcal{O}(1.3803^n)$. This implies that to decrease the running time of the algorithm, we might need to create some new rules, since the running time of the current rule 9 is too high when all the nodes in $N_H(y)$ have two neighbors in S. Let us keep this rule 9, but only execute it when at least one node in $N_H(y)$ has three neighbors in S. This new version of rule 9 can be

seen in Algorithm 5.14. As mentioned it has a branching vector of (4,4,5,6), which gives a running time of $\mathcal{O}(1.3472^n)$.

Algorithm 5.14 New rule 9, the first of the new rules to replace rule 9 and rule 10 of Algorithm 5.12

1: **if** there is a $y \in S$ such that $d_K(y) = 4$, and if there exist an $x_1 \in N_H(y)$, with $d_S(x_1) = 3$. Let $N_H(y) = \{x_1, x_2, x_3, x_4\}$. **then** ▷ 9.1
2: | ENUMCDS$(K \setminus \{x_1\}, S \setminus \{N_S(x_1)\}, X \cup \{x_1\})$
3: | ENUMCDS$(K \setminus \{x_1, x_2\}, S \setminus \{N_S(x_2)\}, X \cup \{x_2\})$
4: | ENUMCDS$(K \setminus \{x_1, x_2, x_3\}, S \setminus \{N_S(x_3)\}, X \cup \{x_3\})$
5: | ENUMCDS$(K \setminus \{x_1, x_2, x_3, x_4\}, S \setminus \{N_S(x_4)\}, X \cup \{x_4\})$
6: **end if**

To solve the cases where all nodes in $N_H(y)$ have two neighbors in S we need to explore a different approach. Instead of looking for a node $y \in S$, with $d_K(y) = 4$, we look for an $x \in K$ with $d_S(x) = 2$. These new rules that we will try to create will handle all cases where there is a node in the clique with degree 2, so it will partially replace rule 9 and completely replace rule 10.

We check if there exists a node x in the clique with two neighbors in the independent set. If such a node x exists, let $N_S(x) = \{y_1, y_2\}$. We know that $d_K(y_1) \geq 4$ and $d_K(y_2) \geq 4$ otherwise rule 8, 7 or 4 would have been applied. Let us start out by looking at the case where at least one of y_1 and y_2 has degree ≥ 5. If this is the case then x can be needed to dominate y_1, or x can be needed to dominate y_2, otherwise we can discard x.

This branching rule is displayed in Algorithm 5.15. This rule has a branching vector of (7,6,1), which gives a running time of $\mathcal{O}(1.3653^n)$. This running time is quite high, but it is better than the current upper bound, so let's keep it for now. We can revisit this rule, and try to change it if we are able to get a better running time when $d_K(y_1) = d_K(y_2) = 4$.

Algorithm 5.15 The second of the new rules to replace rule 9 and 10 of Algorithm 5.12

1: **if** there is an $x \in K$ such that $d_S(x) = 2$ then let $N_H(x) = \{y_1, y_2\}$. If $d_K(y_1) \geq 5$ and $d_K(y_2) \geq 4$. **then** ▷ 9.2
2: | ENUMCDS$(K \setminus \{N_H(y_1)\}, S \setminus \{y_1, y_2\}, X \cup \{x\})$
3: | ENUMCDS$(K \setminus \{N_H(y2)\}, S \setminus \{y_1, y_2\}, X \cup \{x\})$
4: | ENUMCDS$(K \setminus \{x\}, S, X)$
5: **end if**

Note that when $d_K(y_1) = d_K(y_2) = 4$ we cannot use a rule similar to

Algorithm 5.15, since this would give us a branching vector of (6,6,1) and a corresponding running time of $\mathcal{O}(1.3881^n)$, which is higher than the upper bound.

Now we have handled all of the cases where at least one of y_1 and y_2 has degree more than 4. At this point we know that when $d_K(y_1) = d_K(y_2) = 4$ all of the nodes in $N_H(y_1) \cup N_H(y_2)$ have two neighbors in S, since otherwise rule 5, 6, 7, 8, or the new rule 9 would have been applied. Note that the new rule 9 displayed in Algorithm 5.14 handles all cases where a node in S with degree 4 has a neighbor in K with three neighbors in S. We need to consider all cases where both y_1 and y_2 have degree equal to 4, and all nodes in $N_H(y_1) \cup N_H(y_2)$ have two neighbors in S.

To solve these cases with a lower running time we approach them in a slightly different way. We will do the same as we did in Algorithm 5.15, that is enumerate all solutions where x is needed to dominate y_1, and where x is needed to dominate y_2, and the cases where x is not in the solution. Instead of just making one recursive call with x removed from K when x is not in the solution, we can instead look at which ways both y_1 and y_2 can be dominated when x is not in the solution.

Let us start out by looking at the cases where $N_H(y_1) \cap N_H(y_2) = \{x\}$, that is, the cases where x is the only node in K that is a neighbor of both y_1 and y_2. Since both y_1 and y_2 have degree 4 they can have up to 4 common neighbors, and we will look at all of the cases where they have $1, 2, 3$ and 4 mutual neighbors. However, as mentioned we will start by looking at the cases where they only have one neighbor, x, in common. Remember that all of the nodes in $N_H(y_1) \cup N_H(y_2)$ have 2 neighbors in S at this point in the algorithm.

Let us denote $N_H(y_1)$ by $\{x, c_1, c_2, c_3\}$ and $N_H(y_2)$ by $\{x, k_1, k_2, k_3\}$. Since x is the only node in K that can dominate both y_1 and y_2, we know that if we do not use x to dominate y_1 or y_2, then we have to use at least two nodes to dominate y_1 and y_2. Specifically we have to use at least one node from y_1's neighborhood and at least one node from y_2's neighborhood. We have to check all such combinations with at least one node from y_1's neighborhood and at least one node from y_2's neighborhood.

The way we chose to do this is to start by handling all combinations where x is in the solution. Then we handle all combinations where both c_1 and k_1 are in the solution. When we use these two nodes we do not discard any other nodes from $N_H(y_1) \cup N_H(y_2)$, except for x, and in that way we handle all solutions that contain c_1 and k_1. Next we handle all combinations where c_1 and k_2 are in the solution. At this point we have already handled all

cases where c_1 and k_1 are in the solution, and since c_1 is in this solution we can discard k_1 from this solution, and so on. One algorithm that solves the cases we just described is Algorithm 5.16.

Algorithm 5.16 The third of the new rules to replace rule 9 and 10 of Algorithm 5.12

1: **if** there is an $x \in K$ such that $d_S(x) = 2$ then let $N_H(x) = \{y_1, y_2\}$. If $d_K(y_1) = d_K(y_2) = 4$, and $N_H(y_1) \cap N_H(y_2) = \{x\}$. Let us denote $N_H(y_1)$ by $\{x, c_1, c_2, c_3\}$, and $N_H(y_2)$ by $\{x, k_1, k_2, k_3\}$ **then** ▷ 9.3
2: | ENUMCDS$(K \setminus \{N_H(y_1)\}, S \setminus \{y_1, y_2\}, X \cup \{x\})$
3: | ENUMCDS$(K \setminus \{N_H(y2)\}, S \setminus \{y_1, y_2\}, X \cup \{x\})$
4: | ENUMCDS$(K \setminus \{x, c_1, k_1\}, S \setminus \{N_S(c_1) \cup N_S(k_1)\}, X \cup \{c_1, k_1\})$
5: | ENUMCDS$(K \setminus \{x, c_1, k_1, k_2\}, S \setminus \{N_S(c_1) \cup N_S(k_2)\}, X \cup \{c_1, k_2\})$
6: | ENUMCDS$(K \setminus \{x, c_1, k_1, k_2, k_3\}, S \setminus \{N_S(c_1) \cup N_S(k_3)\}, X \cup \{c_1, k_3\})$
7: | ENUMCDS$(K \setminus \{x, c_1, c_2, k_1\}, S \setminus \{N_S(c_2) \cup N_S(k_1)\}, X \cup \{c_2, k_1\})$
8: | ENUMCDS$(K \setminus \{x, c_1, c_2, k_1, k_2\}, S \setminus \{N_S(c_2) \cup N_S(k_2)\}, X \cup \{c_2, k_2\})$
9: | ENUMCDS$(K \setminus \{x, c_1, c_2, k_1, k_2, k_3\}, S \setminus \{N_S(c_2) \cup N_S(k_3)\}, X \cup \{c_2, k_3\})$
10: | ENUMCDS$(K \setminus \{x, c_1, c_2, c_3, k_1\}, S \setminus \{N_S(c_3) \cup N_S(k_1)\}, X \cup \{c_3, k_1\})$
11: | ENUMCDS$(K \setminus \{x, c_1, c_2, c_3, k_1, k_2\}, S \setminus \{N_S(c_3) \cup N_S(k_2)\}, X \cup \{c_3, k_2\})$
12: | ENUMCDS$(K \setminus \{x, c_1, c_2, c_3, k_1, k_2, k_3\}, S \setminus \{N_S(c_3) \cup N_S(k_3)\}, X \cup \{c_3, k_3\})$
13: **end if**

Let us analyze its running time and check if it is lower than the current upper bound. If for all c_i and all k_i, $|N_S(c_i) \cup N_S(k_j)| \geq 4$, we will get a branching vector of (6,6,7,8,9,8,9,10,9,10,11), and a corresponding running time of $\mathcal{O}(1.3451^n)$.

The problem is that we do not know if $N_S(c_i) \cup N_S(k_j)| \geq 4$ for all c_i and all k_j. We know that c_i has y_1 in its set of neighbors for $i = 1, 2, 3$, and k_j has y_2 in its set of neighbors for $j = 1, 2, 3$. Since both c_i and k_j have degree ≥ 2 we know that $|N_S(c_i) \cup N_S(k_j)| \geq 3$, but if there exists one $i = 1, 2, 3$ and one $j = 1, 2, 3$ where c_i and k_j have a neighbor in common, then $|N_S(c_i) \cup N_S(k_j)| = 3$. Since we know that both y_1 and y_2 have to be in $N_S(c_i) \cup N_S(k_j)$. If all pairs of k_j and c_i have $|N_S(c_i) \cup N_S(k_j)| = 3$, the running time we would get by using Algorithm 5.16 is too high.

Luckily there is another way to solve the cases where for at least one c_i and one k_j, $|N_S(c_i) \cup N_S(k_j)| = 3$. Let $i = 1$ and $j = 1$, be one of the pairs with $|N_S(c_i) \cup N_S(k_j)| = 3$, and let us denote $N_S(c_i) \cup N_S(k_j)$ by $\{y_1, y_2, y_3\}$. If this is the case then we know that c_1 dominates y_1 and y_3, and k_1 dominates y_2 and y_3. If we use these nodes, k_1 and c_1, to dominate y_1 and y_2 we know that we can discard all other nodes in $N_H(y_1) \cup N_H(y_2)$, because if any of these nodes are added to the solution, the solution cannot be minimal. We

can use Algorithm 5.17 to solve the cases just described.

Algorithm 5.17 The fourth of the new rules to replace rule 9 and 10 of Algorithm 5.12

1: **if** there is an $x \in K$ such that $d_S(x) = 2$ then let $N_H(x) = \{y_1, y_2\}$. If $d_K(y_1) = d_K(y_2) = 4$, and $N_H(y_1) \cap N_H(y_2) = \{x\}$. Let us denote $N_H(y_1)$ by $\{x, c_1, c_2, c_3\}$, and $N_H(y_2)$ by $\{x, k_1, k_2, k_3\}$ **then** ▷ 9.4
2: | ENUMCDS$(K \setminus \{N_H(y_1)\}, S \setminus \{y_1, y_2\}, X \cup \{x\})$
3: | ENUMCDS$(K \setminus \{N_H(y2)\}, S \setminus \{y_1, y_2\}, X \cup \{x\})$
4: | ENUMCDS$(K \setminus \{x, c_1, c_2, c_3, k_1, k_2, k_3\}, S \setminus \{N_S(c_1) \cup N_S(k_1)\}, X \cup \{c_1, k_1\})$
5: | ENUMCDS$(K \setminus \{x, c_1, k_1, k_2\}, S \setminus \{N_S(c_1) \cup N_S(k_2)\}, X \cup \{c_1, k_2\})$
6: | ENUMCDS$(K \setminus \{x, c_1, k_1, k_2, k_3\}, S \setminus \{N_S(c_1) \cup N_S(k_3)\}, X \cup \{c_1, k_3\})$
7: | ENUMCDS$(K \setminus \{x, c_1, c_2, k_1\}, S \setminus \{N_S(c_2) \cup N_S(k_1)\}, X \cup \{c_2, k_1\})$
8: | ENUMCDS$(K \setminus \{x, c_1, c_2, k_1, k_2\}, S \setminus \{N_S(c_2) \cup N_S(k_2)\}, X \cup \{c_2, k_2\})$
9: | ENUMCDS$(K \setminus \{x, c_1, c_2, k_1, k_2, k_3\}, S \setminus \{N_S(c_2) \cup N_S(k_3)\}, X \cup \{c_2, k_3\})$
10: | ENUMCDS$(K \setminus \{x, c_1, c_2, c_3, k_1\}, S \setminus \{N_S(c_3) \cup N_S(k_1)\}, X \cup \{c_3, k_1\})$
11: | ENUMCDS$(K \setminus \{x, c_1, c_2, c_3, k_1, k_2\}, S \setminus \{N_S(c_3) \cup N_S(k_2)\}, X \cup \{c_3, k_2\})$
12: | ENUMCDS$(K \setminus \{x, c_1, c_2, c_3, k_1, k_2, k_3\}, S \setminus \{N_S(c_3) \cup N_S(k_3)\}, X \cup \{c_3, k_3\})$
13: **end if**

Let us analyze the running time of Algorithm 5.17. In the worst case scenario all pairs of c_i and k_j for $j = 1, 2, 3$ and $i = 1, 2, 3$ can have $|N_S(c_i) \cup N_S(k_j)| = 3$. In that case we will get a branching vector of (6,6,10,7,8,7,8,9,8,9,10) and a corresponding running time of $\mathcal{O}(1.3642^n)$.

Now we have handled all cases where $N_H(y_1) \cap N_H(y_2) = \{x\}$, and we need to handle the cases where y_1 and y_2 have $2, 3$ or 4 common neighbors. Let us start by looking at the case where $N_H(y_1) \cap N_H(y_2) = \{x, x_1\}$, that is when y_1 and y_2 have 2 common neighbors.

We will try a similar strategy as the one we used when $N_H(y_1) \cap N_H(y_2) = \{x\}$. We will handle all cases where x is needed to dominate y_1, and where x is needed to dominate y_2, and the cases where x is not in the solution. We will also do the same for x_1. If none of x or x_1 are in the solution, we will again have to use at least two nodes to dominate y_1 and y_2, with at least one from $N_H(y_1)$ and at least one from $N_H(y_2)$. Algorithm 5.18 solves these cases.

In the worst case scenario all pairs of $i = 1, 2$ and $j = 1, 2$ will have $|N_S(c_i) \cup N_S(k_j)| = 3$. We will then get a branching vector of (6,6,6,6,7,8,8,9), and a corresponding running time of $\mathcal{O}(1.3564^n)$.

Algorithm 5.18 The fifth of the new rules to replace rule 9 and 10 of Algorithm 5.12

1: **if** there is an $x \in K$ such that $d_S(x) = 2$ then let $N_H(x) = \{y_1, y_2\}$. If $d_K(y_1) = d_K(y_2) = 4$, and $N_H(y_1) \cap N_H(y_2) = \{x, x_1\}$. If $d_S(x_1) = 2$. Let us denote $N_H(y_1)$ by $\{x, x_1, c_1, c_2\}$, and $N_H(y_2)$ by $\{x, x_1, k_1, k_2\}$. **then** ▷ 9.5
2: | ENUMCDS$(K \setminus \{N_H(y_1)\}, S \setminus \{y_1, y_2\}, X \cup \{x\})$
3: | ENUMCDS$(K \setminus \{N_H(y2)\}, S \setminus \{y_1, y_2\}, X \cup \{x\})$
4: | ENUMCDS$(K \setminus \{N_H(y_1)\}, S \setminus \{y_1, y_2\}, X \cup \{x_1\})$
5: | ENUMCDS$(K \setminus \{N_H(y2)\}, S \setminus \{y_1, y_2\}, X \cup \{x_1\})$
6: | ENUMCDS$(K \setminus \{x, x_1, c_1, k_1\}, S \setminus \{N_S(c_1) \cup N_S(k_1)\}, X \cup \{c_1, k_1\})$
7: | ENUMCDS$(K \setminus \{x, x_1, c_1, k_1, k_2\}, S \setminus \{N_S(c_1) \cup N_S(k_2)\}, X \cup \{c_1, k_2\})$
8: | ENUMCDS$(K \setminus \{x, x_1, c_1, c_2, k_1\}, S \setminus \{N_S(c_2) \cup N_S(k_1)\}, X \cup \{c_2, k_1\})$
9: | ENUMCDS$(K \setminus \{x, x_1, c_1, c_2, k_1, k_2\}, S \setminus \{N_S(c_2) \cup N_S(k_2)\}, X \cup \{c_2, k_2\})$
10: **end if**

At this point we have handled all the cases where $N_H(y_1) \cap N_H(y_2) = \{x\}$ and the cases where $N_H(y_1) \cap N_H(y_2) = \{x, x_1\}$. We also have to handle the cases where y_1 and y_2 has 3 or 4 common neighbors. Let us start by looking at the cases where $N_H(y_1) \cap N_H(y_2) = \{x, x_1, x_2\}$, that is the cases where y_1 and y_2 have 3 common neighbors.

As before we will handle all cases where x is needed to dominate y_1, and where x is needed to dominate y_2, or where x is not in the solution. We will do the same for x_1 and x_2. In the cases where none of x, x_1 or x_2 is in the solution we will again have to use at least two nodes to dominate both y_1 and y_2, with at least one node from $N_H(y_1)$ and at least one node from $N_H(y_2)$.

Algorithm 5.19 The sixth of the new rules to replace rule 9 and 10 of Algorithm 5.12

1: **if** there is an $x \in K$ such that $d_S(x) = 2$ then let $N_H(x) = \{y_1, y_2\}$. If $d_K(y_1) = d_K(y_2) = 4$, and $N_H(y_1) \cap N_H(y_2) = \{x, x_1, x_2\}$. If $d_S(x_1) = d_S(x_2) = 2$. Let us denote $N_H(y_1)$ by $\{x, x_1, x_2, c_1\}$, and $N_H(y_2)$ by $\{x, x_1, x_2, k_1\}$. **then** ▷ 9.6
2: | ENUMCDS$(K \setminus \{N_H(y_1)\}, S \setminus \{y_1, y_2\}, X \cup \{x\})$
3: | ENUMCDS$(K \setminus \{N_H(y2)\}, S \setminus \{y_1, y_2\}, X \cup \{x\})$
4: | ENUMCDS$(K \setminus \{N_H(y_1)\}, S \setminus \{y_1, y_2\}, X \cup \{x_1\})$
5: | ENUMCDS$(K \setminus \{N_H(y2)\}, S \setminus \{y_1, y_2\}, X \cup \{x_1\})$
6: | ENUMCDS$(K \setminus \{N_H(y_1)\}, S \setminus \{y_1, y_2\}, X \cup \{x_2\})$
7: | ENUMCDS$(K \setminus \{N_H(y2)\}, S \setminus \{y_1, y_2\}, X \cup \{x_2\})$
8: | ENUMCDS$(K \setminus \{x, x_1, x_2, c_1, k_1\}, S \setminus \{N_S(c_1) \cup N_S(k_1)\}, X \cup \{c_1, k_1\})$
9: **end if**

Algorithm 5.19 handles all of these cases. In the worst case scenario $i = 1$ and $j = 1$ will have $|N_S(c_i) \cup N_S(k_j)| = 3$. We will get a branching vector of (6,6,6,6,6,6,8) and a corresponding running time of $\mathcal{O}(1.3674^n)$.

So far we have handled all cases where $N_H(y_1) \cap N_H(y_2) = \{x\}$, $N_H(y_1) \cap N_H(y_2) = \{x, x_1\}$ and where $N_H(y_1) \cap N_H(y_2) = \{x, x_1, x_2\}$. We will now handle the last cases, these are when y_1 and y_2 have 4 neighbors in common. Let us denote $N_H(y_1) \cap N_H(y_2)$ by $\{x, x_1, x_2, x_3\}$.

At this point all of y_1's neighbors are also neighbors of y_2, and vice versa, since $d_K(y_1) = d_K(y_2) = 4$, and $|N_H(y_1) \cap N_H(y_2)| = 4$. We also know that x, x_1, x_2 and x_3 cannot be a part of the same mcds, since they dominate the exact same set of neighbors, that is they all have $\{y_1, y_2\}$ as their set of neighbors in S. We will look at the cases where we use x to dominate both y_1 and y_2, and we will also do the same for x_1, x_2 and x_3.

Algorithm 5.20 handles all of these cases. We will get a branching vector of (6,6,6,6), and a corresponding running time of $\mathcal{O}(1.2600^n)$.

Algorithm 5.20 The seventh of the new rules to replace rule 9 and 10 of Algorithm 5.12

1: **if** there is an $x \in K$ such that $d_S(x) = 2$ then let $N_H(x) = \{y_1, y_2\}$. If $d_K(y_1) = d_K(y_2) = 4$, and $N_H(y_1) \cap N_H(y_2) = \{x, x_1, x_2, x_3\}$, and $d_S(x_1) = d_S(x_2) = d_S(x_3) = 2$. **then** ▷ 9.7
2: | ENUMCDS($K \setminus \{x, x_1, x_2, x_3\}, S \setminus \{y_1, y_2\}, X \cup \{x\}$)
3: | ENUMCDS($K \setminus \{x, x_1, x_2, x_3\}, S \setminus \{y_1, y_2\}, X \cup \{x_1\}$)
4: | ENUMCDS($K \setminus \{x, x_1, x_2, x_3\}, S \setminus \{y_1, y_2\}, X \cup \{x_2\}$)
5: | ENUMCDS($K \setminus \{x, x_1, x_2, x_3\}, S \setminus \{y_1, y_2\}, X \cup \{x_3\}$)
6: **end if**

In these new rules that can replace rule 9 and 10 of the algorithm the one with the highest running time is the sixth rule, 9.6, which has a running time of $\mathcal{O}(1.3674^n)$. The highest running time amongst the other rules is $\mathcal{O}(1.3248^n)$, given by rule 5.3, 6, 8 and 11. We have thus proved the following new and better upper bound for the maximum number of mcds in split graphs.

Theorem 5.1. *A split graph on n nodes has at most 1.3674^n mcds and these can be enumerated in time $\mathcal{O}(1.3674^n)$.*

Consequently, we immediately also obtain the following result:

Corollary 5.1. *A minimum connected dominating set on an input split graph on n nodes can be found in time $\mathcal{O}(1.3674^n)$.*

Due to the fact that a minimum dominating set of a split graph can always be assumed to be connected [9], we also get the following corollary:

Corollary 5.2. *A minimum dominating set on an input split graph on n nodes can be found in time $\mathcal{O}(1.3674^n)$.*

Chapter 6

Conclusion

The goal of the work leading to this thesis was to try to narrow the gap between the upper and lower bounds on the maximum number of mcds in split graphs. In principle, the main idea was to test many split graphs with a practical implementation of the algorithm, and see if this lead to new lower bound examples. Thus, we believed that the chances of narrowing the gap would be by obtaining a higher lower bound. We did not find a graph example that could give us a higher lower bound, but our implementation and our test results convinced us that the upper bound was too high. We therefore did a deeper theoretical analysis of the rules in the algorithm and were actually able to prove a lower upper bound. We achieved our goal of narrowing the gap, but surprisingly in a different way than we expected. Instead of improving the lower bound by practical tests, we found a better upper bound by theoretical analysis.

6.1 Summary

The main contributions of this thesis is Theorem 5.1. We proved that the number of mcds in split graphs is at most 1.3674^n, and the mcds of an input split graph can be enumerated in time $\mathcal{O}(1.3674^n)$. In Chapter 1 we gave an essential introduction to the subject of this thesis. In Chapter 2 we explained the concept of lower bounds. We presented the current lower bound example and our new lower bound example. In Chapter 3 we discussed branching algorithms and presented the branching algorithm that gave the best known upper bound on mcds in split graphs. We showed the importance of actually implementing an algorithm by the two errors we found in the algorithm for enumerating mcds. Crucial errors can be neglected when an algorithm is

not implemented, and these errors might influence the running time of the algorithm or the number of objects found by the algorithm in an incorrect way. In Chapter 4 we went through the implementation details. In Chapter 5 we present the main results and contributions of this thesis.

6.2 Co-bipartite graphs

A class of graphs very similar to split graphs is the class of co-bipartite graphs. Indeed many of our findings can also be used for co-bipartite graphs. Let us take a closer look at the graph class of co-bipartite graphs. We will first define the graph class of co-bipartite graphs, and then we will show how enumerating mcds in co-bipartite graphs is quite similar to enumerating mcds in split graphs.

Definition 6.1. *A co-bipartite graph G is a graph where the nodes can be partitioned into two cliques.*

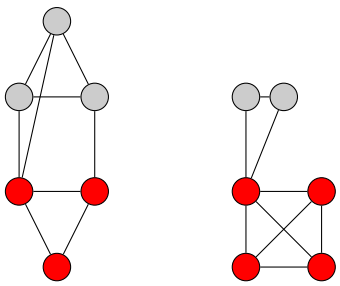

Figure 6.1: Examples of co-bipartite graphs, where the red nodes represent the first clique, and the gray nodes represent the second clique.

In Figure 6.1 we see some examples of co-bipartite graphs where for each of the graphs the first clique in the graph consists of the red nodes, and the second clique in the graph consists of the gray nodes. Any number of edges can be present between the two cliques.

In a co-bipartite graph we know that all mcds have to be a set of nodes that are in the same clique, or a set of exactly two nodes, with one node from each of the two cliques in the graph. This implies that if we have a mcds D of a co-partite graph, with $|D| > 2$, we know that all of the nodes in D have to be members of the same clique. We summarize this in the following lemma:

Lemma 6.1. *For every co-bipartite graph G all mcds D of G with $|D| > 2$, all nodes $n \in D$ have to be from the same clique in G.*

When we look for mcds in co-bipartite graphs we first check all possible subsets of exactly two nodes, with one node from each of the cliques, this would be an $\mathcal{O}(n^2)$ time algorithm. Then we have to check all the subsets of the graph that only consists of nodes from one of the cliques. We can actually use the algorithm described earlier for split graphs given by Golovach et al. [11] to solve this. When we look for mcds that are subsets of one clique we can view the other clique as the independent set in a split graph, since none of these nodes can be a part of the mcds, but they have to be dominated by the mcds. So if we run the algorithm given for split graphs two times, one time for each of the cliques, we will find all mcds that only contain nodes from one of the cliques.

The algorithm given for split graphs runs in time $\mathcal{O}(1.3803^n)$, and we have to run this algorithm twice. We also have to run the $\mathcal{O}(n^2)$ time algorithm to find all mcds consisting of one node from each clique. This results in a total running time of $\mathcal{O}(n^2 + 2 \cdot 1.3803^n)$. This is actually the best enumeration algorithm known for mcds in co-bipartite graphs, and gives the current upper bound on the maximum number of mcds in co-bipartite graphs and was given by Golovach et al. [11].

Our new upper bound of 1.3674^n on the maximum number of mcds in split graphs also gives a new upper bound on the maximum number of mcds in co-bipartite graphs. All mcds in a co-bipartite graph can be found by running the algorithm by Golovach et al. [11] twice, in addition to running an n^2 time algorithm. The adjustments we made in the algorithm for split graphs give us a better running time on enumerating mcds in co-bipartite graphs, and give a new upper bound on the maximum number of mcds in co-bipartite graphs.

Theorem 6.1. *A co-bipartite graph on n nodes has at most $n^2 + 2 \cdot 1.3674^n$ mcds and these can be enumerated in time $\mathcal{O}(n^2 + 2 \cdot 1.3674^n)$.*

Corollary 6.1. *A minimum connected dominating set on an input co-bipartite graph on n nodes can be found in time $\mathcal{O}(n^2 + 2 \cdot 1.3674^n)$.*

6.3 Further work

In this thesis we were able to prove a better upper bound of 1.3674^n on the maximum number of mcds in split graphs. Considering how small the gap between the upper and lower bound on the maximum number of mcds in split graphs, this is a significant improvement in the upper bound. However, we think the upper bound can be decreased even further. In fact we think that the lower bound of 1.3195^n might actually be the correct maximum

number of mcds that any split graph can have.

Even though we were not able to decrease the upper bound to match the lower bound in this thesis, we think that it might be possible. This would require even more tedious analysis, and because of the natural time limitation on a master project, we did not have time to go into even deeper analysis of the rules in the algorithm. Even if it is not possible to improve the rules of the algorithm to obtain a matching upper and lower bound of 1.3195^n, we definitively think that the upper bound can be decreased further down.

If it is not possible to obtain matching upper and lower bounds by theoretical analysis of the rules in the algorithm, then there might exist a higher lower bound example. If this is the case then we think that it might be reasonable to search for such a new lower bound example among more specific graphs than we did. As mentioned we generated and tested all split graphs of size up to 11 nodes, and it will take significantly longer time to test all graphs with a higher number of nodes, even with a faster generation method than the one we used.

As mentioned, we tested a lot of split graphs of different sizes, and noticed that there was a higher number of mcds in graphs containing approximately 30 percent of its nodes in the independent set. Thus, it can be reasonable to search for a possible new lower bound example among such graphs. One can also test properties such as the average degree of the nodes in the clique and independent set, and other properties of split graphs to further restrict the space to search for a new lower bound example in.

This discussion leaves us with two open questions which results in two different approaches to continue with the work from this thesis:

- Can the running time of the branching rules in the main algorithm of this thesis be decreased further?
- Are there split graphs with more than 1.3195^n mcds?

Other interesting questions around this topic could be:

- Is there an algorithm for (Connected) Dominating Set for split graphs that is faster than $\mathcal{O}(1.3674^n)$?
- What is the maximum number of mcds in general graphs? Any improvement over the trivial bound of 2^n would be very interesting.

Although it was not in the scope of our work, the paper by Golovach et al. [11] also studies upper and lower bounds on the maximum number of mcds in other graph classes. For example for the class of chordal graphs, there is again a gap between the lower bound which is $3^{(n-2)/3} \sim 1.4423^n$ and the upper bound which is 1.7159^n. Similar work to that we did in this thesis can also be conducted for this graph class.

References

[1] Faisal N. Abu-Khzam and Pinar Heggernes. Enumerating minimal dominating sets in chordal graphs. *Information Processing Letters*, 116(12):739 – 743, 2016.

[2] Faisal N. Abu-Khzam, Amer E. Mouawad, and Mathieu Liedloff. An exact algorithm for connected red–blue dominating set. *Journal of Discrete Algorithms*, 9(3):252 – 262, 2011. Selected papers from the 7th International Conference on Algorithms and Complexity (CIAC 2010).

[3] László Babai. Graph isomorphism in quasipolynomial time. *CoRR*, abs/1512.03547, 2015.

[4] Alan A. Bertossi. Dominating sets for split and bipartite graphs. *Information Processing Letters*, 19(1):37 – 40, 1984.

[5] Jean-François Couturier, Pinar Heggernes, Pim van 't Hof, and Dieter Kratsch. Minimal dominating sets in graph classes: Combinatorial bounds and enumeration. *Theoretical Computer Science*, 487:82 – 94, 2013.

[6] Jean-François Couturier, Romain Letourneur, and Mathieu Liedloff. On the number of minimal dominating sets on some graph classes. *Theoretical Computer Science*, 562:634 – 642, 2015.

[7] Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch. Solving connected dominating set faster than 2^n. In S. Arun-Kumar and Naveen Garg, editors, *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science: 26th International Conference, Kolkata, India, December 13-15.*, pages 152 – 163. Springer, 2006.

[8] Fedor V. Fomin and Dieter Kratsch. *Exact Exponential Algorithms*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010.

[9] Fedor V. Fomin, Dieter Kratsch, and Gerhard J. Woeginger. Exact (exponential) algorithms for the dominating set problem. In Juraj

Hromkovič, Manfred Nagl, and Bernhard Westfechtel, editors, *Graph-Theoretic Concepts in Computer Science: 30th International Workshop, WG 2004, Bad Honnef, Germany, June 21-23.*, pages 245 – 256. Springer, 2005.

[10] Petr A. Golovach, Pinar Heggernes, Mamadou Moustapha Kanté, Dieter Kratsch, and Yngve Villanger. Minimal dominating sets in interval graphs and trees. *Discrete Applied Mathematics*, 216, Part 1:162 – 170, 2017.

[11] Petr A. Golovach, Pinar Heggernes, and Dieter Kratsch. Enumerating minimal connected dominating sets in graphs of bounded chordality. *Theoretical Computer Science*, 630:63 – 75, 2016.

[12] Petr A. Golovach, Pinar Heggernes, Dieter Kratsch, and Reza Saei. Subset feedback vertex sets in chordal graphs. *Journal of Discrete Algorithms*, 26:7 – 15, 2014.

[13] T. Hearne and C. Wagner. Minimal covers of finite sets. *Discrete Math.*, 5(3):247 – 251, July 1973.

[14] Gordon F. Royle. Counting set covers and split graphs. *Journal of Integer Sequences [electronic]*, 3(2):Art. 00.2.6, 5 p, 2000.

[15] Ryuhei Uehara, Seinosuke Toda, and Takayuki Nagoya. Graph isomorphism completeness for chordal bipartite graphs and strongly chordal graphs. *Discrete Applied Mathematics*, 145(3):479 – 482, 2005.

[16] Douglas B. West. *Introduction to Graph Theory.* Prentice Hall, 2 edition, September 2000.

[17] Jie Wu and Hailan Li. On calculating connected dominating set for efficient routing in ad hoc wireless networks. In *Proceedings of the 3rd International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, DIALM '99, pages 7–14, New York, NY, USA, 1999. ACM.