University of Bergen

Department of Informatics

Algorithms

Maximum Number of Objects in Graph Classes.

Student:
Marit Hellestø

Supervisor:
Professor Pinar Heggernes

Master Thesis

June 2015

Acknowledgement

I would like to thank my thesis supervisor Pinar Heggernes for all the help, time and guidance I have received while working on this thesis. In addition I would like to thank my fellow master students who helped motivate me to work harder.

Contents

Chapter 1

Introduction

To find the largest number of occurrences of a set of vertices with given constraints, is an important field of study in graph theory and graph algorithms.

An upper bound on the number of vertex sets satisfying some property is the proved maximum number of these vertex sets that can be found in any graph. This bound is often proved with the help of branching algorithms. A lower bound is given by the maximum number of vertex sets we have been able to find in a graph, and is proved by giving an example graph with this number of vertex sets.

A classical example is Moon and Moser's theorem from 1965 which states that the number of maximal cliques and maximal independent sets in any graph on n vertices is at most $3^{n/3}$ [20]. This is a tight upper bound as there exist graphs with $3^{n/3}$ maximal independent sets. With a tight bound we mean that the upper bound is equal to the lower bound. We give the lower bound example for the number of maximal independent sets in general graphs in Figure 1.1. This example consists of $n/3$ disjoint triangles.

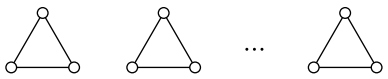


Figure 1.1: A graph having $3^{n/3}$ maximal independent sets

Sometimes we are able to give an upper bound on the number of vertex sets satisfying a given property, but we are not able to find a lower bound example that reaches this bound. For instance the number of minimal subset feedback vertex sets in general graphs, where the upper bound is known to be 1.8638^n [12], but the largest number of minimal subset feedback vertex sets that have been found in a graph gives the lower bound 1.5927^n. Thus these bounds are not tight. In such cases it is not known whether the upper bound is too high, or whether there exists a better lower bound. One way

of trying to tighten these bounds is to restrict the input to certain types of graphs. An example where this has been effective is when finding all minimal subset feedback vertex sets in chordal graphs. In this case the upper bound is 1.6708^n and the lower bound is 1.5848^n [3], making the bounds closer to each other than in general graphs.

In this thesis we will study such upper and lower bounds. More specifically we will use an algorithm given by Couturier et al. in [2] to study the maximum number of minimal dominating sets in chordal graphs. And we will use an algorithm given by Golovach et al. in [12] to study the maximum number of minimal subset feedback vertex sets in chordal and split graphs. We will implement these algorithms and study how the proven bounds compare to practical experiments. The implementation of these algorithms have generated some interesting test results. Especially for the number of minimal subset feedback vertex sets in split graphs. These results will be presented in later chapters.

1.1 Notation

In this thesis we work with simple undirected graphs. We denote a graph by $G = (V, E)$, where V is the set of vertices and E is the set of edges in G. The *neighbourhood* of a vertex $v \in V(G)$ is the set of vertices adjacent to v, and it is denoted by $N_G(v)$. The *closed neighbourhood* of v is $N_G[v] = N(v) \cup \{v\}$. The *degree* of a vertex v is $|N_G(v)|$, and it is denoted $d_G(v)$. For a set $S \subseteq V$ we define the neighbourhood and the closed neighbourhood of S as $N_G(S) = \cup_{v \in S} N_G(v) \setminus S$ and $N_G[S] = N_G(S) \cup S$ respectively. We say that a vertex v is *isolated* if it has no neighbours, that is if $N_G(v) = \emptyset$.

A set $D \subseteq V$ is a *dominating set* in G if for all vertices v in G, either $v \in D$ or $v \in N_G(D)$. A dominating set is *minimal* if no proper subset of D is a dominating set. That is for any vertex v in D, $D \setminus \{v\}$ is not a dominating set. In the DOMINATING SET problem we are given a graph G, and an integer k and we are asked to find a dominating set of size k.

The sub graph of G *induced* by S is denoted by $G[S]$. We use $G - v$ to denote the graph $G[V \setminus \{v\}]$, and $G - S$ to denote the graph $G[V \setminus S]$.

A *path* in G is a sequence of distinct vertices such that the next vertex in the sequence is adjacent to the previous vertex. A *cycle* is a path with at least three vertices such that the first vertex is adjacent to the last vertex.

Given a subset $S \subseteq V$, we call a cycle an *S-cycle* if it contains a vertex of S. For a cycle or S-cycle C we denote the set of vertices in C by $V(C)$. A subset $F \subseteq V$ is a *forest* if $G[F]$ contains no cycle. F is an *S-forest* if no cycle in $G[F]$ contains a vertex of S.

A graph is *connected* if there is a path between every pair of its vertices. A maximal connected sub graph of G is called a *connected component* of G.

A set $X \subseteq V$ is a *clique* if $uv \in E(G)$ for every pair of vertices $u, v \in X$.

A clique is *maximal* if no proper superset of it is a clique. $X \subseteq V$ is an *independent set* if $uv \notin E(G)$ for every pair of vertices $u, v \in X$.

$X \subseteq V$ is a *feedback vertex set* of G, if $G - X$ contains no cycles. Given a graph G and a set $S \subseteq V$, then a set $U \subseteq V$ is a *subset feedback vertex set* of G if no cycle in $G - U$ contains a vertex of S. A subset feedback vertex set U is *minimal* if no proper subset of U is a subset feedback vertex set of G. In the SUBSET FEEDBACK VERTEX SET problem we are given a graph G, a set $S \subseteq V$ and an integer k and asked to find a subset feedback vertex set of size k such that $G - U$ has no cycles containing a vertex of S.

A *simplicial vertex* is a vertex v such that $N_G(v)$ is a clique. A *Perfect Elimination Ordering* (PEO) in a graph $G = (V, E)$ is an ordering $[v_1, v_2, ..., v_n]$ of the vertices in G such that each v_i is a simplicial vertex in the induced sub graph $G[v_i, ..., v_n]$.

1.2 Graph classes

A *graph class* is an infinite set of graphs that satisfy some common property. There are many important graph problems that are NP-hard on the class of general graphs. Meaning the set of graphs without any particular restrictions. Since these graph problems are NP-hard, we do not expect that they can be solved in polynomial time for general graphs. However most of the instances of these problems have nice properties that make them solvable in polynomial time. Therefore a very common way of dealing with hard problems is to restrict the input to various graph classes. In this way some NP-hard problems become polynomial time solvable on these graph classes.

An example of a NP-hard problem that become polynomial, when restricting the input to a certain graph class, is the famous problem CLIQUE which asks for a clique of maximum size in a graph. This problem is solvable in polynomial time on chordal graphs [11]. We will explain what a chordal graph is and different properties of chordal graphs in the next section.

When studying the maximum number of vertex subsets with various properties in graphs, it is also interesting to consider input graphs that belong to different graph classes. The number vertex sets may be exponential in general graphs but could be polynomial when restricting the input to specific graph classes. An example of this is the number of minimal feedback vertex sets which on general graphs is exponential, but that is polynomial on split graphs [12]. It could also be that upper and lower bounds get closer in restricted graph classes. For instance the currently best known upper and lower bounds on the number of minimal subset feedback vertex sets for general graphs are 1.8638^n and 1.5927^n, but for chordal graphs these bounds are 1.6708^n and 1.5848^n respectively [12].

Both the DOMINATING SET problem and the SUBSET FEEDBACK VERTEX SET problem stay NP-hard in chordal and split graphs, but this is not

what we will study in this thesis. We will study the number of minimal dominating sets in chordal graphs and the number of minimal subset feedback vertex sets in chordal and split graphs. The number of these vertex subsets is exponential on general graphs and, more importantly for us, exponential on both chordal and split graphs.

1.2.1 Chordal graphs

In this section we will present properties of chordal graphs in addition to presenting two algorithms that combined are used to recognize chordal graphs. These algorithms will later be used as part of the preprocessing step for the algorithms that will be used to find and list all minimal dominating sets and all minimal subset feedback vertex sets in chordal graphs found in [2] and [12] respectively. These algorithms will be described in Chapters 3 and 4.

Chordal graphs have been found interesting in many areas. They have applications in sparse matrix computations [21] and computational biology and phylogenetics [22], among many other areas. In addition they form one of the first graph classes to be recognized as perfect [13]. It was mentioned that some problems become polynomial-time solvable on different graph classes. A number of problems that are NP-complete on general graphs are polynomial-time solvable in chordal graphs [15]. Examples of such problems are CLIQUE and COLOURING.

Definition 1. *A graph is chordal if every cycle of length* > 3 *has a chord.*

A chord is an edge between two non-consecutive vertices of a cycle. We show the difference between a chordal and a non-chordal graph on four vertices in Figure 1.2. The red edge in the rightmost square of the figure is a chord.

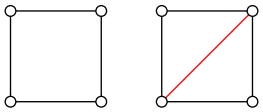


Figure 1.2: To the left a non-chordal graph, to the right a chordal graph.

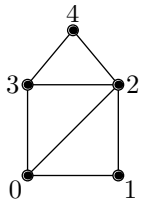
Theorem 1. *[4] Every induced sub graph of a chordal graph is chordal.*

Theorem 2. *[4] Every chordal graph is either a clique or it contains two non-adjacent simplicial vertices.*

Theorem 3. *[10] A graph is chordal if and only if it has a PEO.*

Theorem 2 can be used to recognize whether a graph is chordal. This can be done by repeatedly finding a simplicial vertex in the graph and deleting

it, until either there are no more vertices in the graph, meaning that the graph is chordal, or until there are no more simplicial vertices in the graph, meaning that it is not chordal. Note that if the graph is chordal, the order that these vertices have been deleted in form a PEO of the graph. This is not a very efficient algorithm in terms of running time. In fact it has been shown by Tarjan that we can recognize whether a graph is chordal in linear time [23]. This is done by finding an ordering of the vertices in linear time and checking that this ordering is a PEO.



α_1: [4, 3, 2, 1, 0]

α_2: [1, 4, 2, 0, 3]

Figure 1.3: A chordal graph, and two of its PEOs

Theorem 3 states that a graph is chordal if and only if it has a PEO, but the PEO of a graph is not necessarily unique. A chordal graph can have many different PEOs. This is illustrated in Figure 1.3, where both orderings α_1 and α_2 are PEOs of the graph in the figure.

Tarjan and Yannakakis gives an algorithm called *maximum cardinality search* (MCS) in [23]. MCS gives an ordering α of the vertices in a graph. It is proved in [23] that this ordering is a PEO, if the graph is chordal. We use the MCS algorithm in combination with the CHECKPEO algorithm that is described in Algorithm 1.2 to verify whether a graph is chordal and to generate PEOs.

The pseudo code for finding an ordering of the vertices with MCS, is presented in Algorithm 1.1, and the pseudo code for checking that this ordering is in fact a PEO, is presented in Algorithm 1.2. Both these algorithms were given by Tarjan and Yannakakis in [23].

The MCS algorithm works as follows: It takes as input a graph G and outputs an ordering α. It starts by choosing a vertex v, removing it from the graph G and putting it first in the ordering α. After which it marks all its neighbours. This is illustrated in Figure 1.4 (b). When the algorithm chooses the next vertex v it chooses any of the vertices with the highest number of neighbours in α. And puts v first in α, as shown in Figure 1.4 (c). It continues in this way, choosing a vertex with the largest number of neighbours in α, putting it first in α and marking all its neighbours, before removing it from G. The algorithm terminates when there are no more vertices in G. Figure 1.4 illustrates how the algorithm would proceed in a graph with 5 vertices. In this example the first vertex chosen is 0. The numbers in parenthesis gives the number of neighbours already placed in the

ordering. An implementation of Algorithm 1.1 can be found in Appendix A.

Algorithm 1.1 MCS algorithm

Input: Graph G
Output: An ordering α of the vertices of G

```
1: i:= Number of vertices of G
2: j:=0
3: for All vertices do
4: |  level(v) := 0
5: |  add v to set(0)
6: end for
7: while i ≥ 0 do
8: |  v:= vertex from set(j)
9: |  delete v from set(j)
10: |  add v to α at position i
11: |  level(v) := −1
12: |  for All neighbours w of v where level(w) ≥ 0 do
13: |  |  delete w from set(level(w))
14: |  |  level(w) := level(w)+1
15: |  |  add w to set(level(w))
16: |  end for
17: |  i := i − 1
18: |  j := j + 1
19: |  while j ≥ 0 and set(j)= ∅ do
20: |  |  j := j − 1
21: |  end while
22: end while
```

To check that the ordering α is in fact a PEO, we use the CHECKPEO algorithm described in Algorithm 1.2. This algorithm is given by Tarjan and Yannakakis in [23] as `Test for zero fill-in`. We know that any graph can be made chordal by adding edges to the graph. Algorithm 1.2 verifies that no edges have to be added to make the graph chordal.

Note that we abuse the notation somewhat in Algorithm 1.2, such that $\alpha(i)$ returns element at position i in α while $\alpha(v)$ returns the index of v in α. Before we start to describe the algorithm, we define $f(v)$, the follower of v, to be the vertex of largest index, in α given by the MCS algorithm, that is both a neighbour of v and that comes before v in α.

The algorithm works as follows: For all vertices v, compute $f(v)$. For every $\{v, w\}$ in $E(G)$, where v comes before w in α, verify that either $\{f(v), w\}$ is in $E(G)$ or that $f(v) = w$. In this way we check that all cycles > 3 has a chord. An implementation of this algorithm is given in Appendix A.

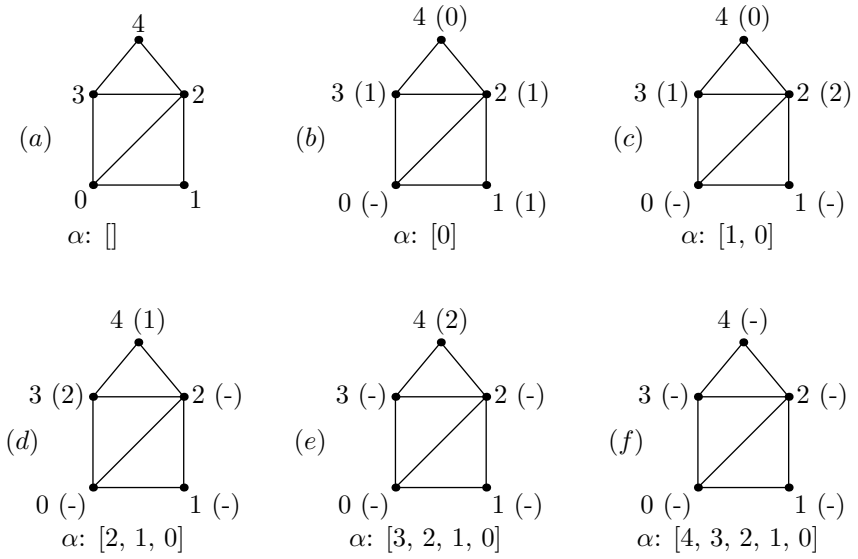


Figure 1.4: Figure showing how an ordering α is found with MCS algorithm

Algorithm 1.2 CheckPEO algorithm

Input: Graph G, Ordering α
Output: if G is chordal **return true** , else **return false**

```
1:  for i := 0 to n do
2:      w := α(i) in α
3:      f(w) := w
4:      index(w) := i
5:      for all neighbours v of w where α(v) < i do
6:          index(v) = i
7:          if f(v) = v then
8:              f(v) = w
9:          end if
10:     end for
11:     for all neighbours v of w where α(v) < i do
12:         if index(f(v)) < i then
13:             return false
14:         end if
15:     end for
16: end for
17: return true
```

The algorithms that have been described above will be used both to test for chordality in graphs and to generate PEOs for the graphs that we generate. The PEOs are important for achieving the correct running time for the algorithms in [2] and [12]. The reason for this will be explained in

later chapters.

1.2.2 Split graphs

In this section we will first present a few properties of split graphs before we present how we can recognize a split graph.

Definition 2. *A graph $G = (V, E)$ is a split graph if V can be partitioned into a clique and an independent set.*

Split graphs are a sub-class of chordal graphs, which is why the algorithm given by Golovach et al. in [12] also gives an upper bound on the number of minimal subset feedback vertex sets in split graphs.

It was mentioned in the previous section that upper and lower bounds on the number of vertex subsets in graphs can be closer when studied in restricted graph classes. An example of this is the bounds on the number of minimal feedback vertex sets. For general graphs the best known upper and lower bounds are 1.8638^n and 1.5927^n respectively. But for split graphs these bounds are not only tight but also polynomial, the upper and lower bounds for this graph class has been shown to be n^2 [12]. This is due to the partition of the vertices in the graph that form a clique. There can be at most two vertices of the clique left in the graph after the feedback vertex set has been removed which results in n^2 possible minimal feedback vertex sets.

We define a degree sequence $d_1 \geq d_2 \geq ... \geq d_n$ of an undirected graph G, to be an ordering of the vertices $v_1, v_2, ..., v_n$ of G such that $d_G(v_i) = d_i$ and $d_i \geq d_{i+1}$.

Theorem 4. *[1] Given a degree sequence $d_1 \geq d_2 \geq ... \geq d_n$ of an undirected graph G, let $w = max\{i \mid d_i \geq i - 1\}$. Then G is a split graph if and only if*

$$\sum_{i=1}^{w} d_i = w(w-1) + \sum_{i=m+1}^{n} d_i \tag{1.1}$$

Corollary 1. *[13] If G is a split graph, then every graph with the same degree sequence as G is also a split graph.*

Theorem 4 is important for recognizing whether a graph is a split graph. In fact it can be used to recognize whether a graph is a split graph in linear time. The equation works by identifying what index the last vertex of the clique has in the degree sequence and assigning that value to w. Then it computes the sums as given by (1.1). Where the left hand side sums up the degree of all the vertices in the clique, this will necessarily also count edges between the clique and the independent set. The right hand side first summarizes the edges in the clique, and then adding all edges that have an

endpoint in the independent set. When these two sides are equal G is a split graph. An implementation of this equation can be found in Appendix A.

One of the goals of this thesis is to study the bounds for the number of minimal subset feedback vertex sets in split graphs. Therefore we need to be able to recognize the graphs that are split graphs in our data sets.

1.3 Branching Algorithms

Branching algorithms can be used to find and list all feasible solutions of a problem. One goal of this thesis is to generate all minimal dominating sets for a selection of chordal graphs, and all minimal subset feedback vertex sets for a selection of chordal and split graphs. The two algorithms that we implement as part of this thesis are both branching algorithms.

A basic branching algorithm for a graph problem typically works as follows: For every vertex v we have one branch where we generate all solutions containing v, and one branch where we generate all solutions not containing v. Every time we branch we generate new sub problems, that is we generate new instances not containing the vertices that we have already branched on. As a consequence, the deeper we get in the branching tree, the smaller instances we have to consider, which in turn makes it faster to solve the sub-problems. The tree generated by the branching of our algorithm is called a computation tree. The leaves of the computation tree represent all feasible solutions to our problem.

In graphs we often say that the size of an instance is equal to the number of vertices in the graph. By generating new sub-problems with a *branching rule* or a *reduction rule* we can reduce the size of the instance. A branching rule decides how we branch on vertices in different circumstances, while a reduction rule removes or forbid vertices without branching. A reduction rule is used if there are vertices that are either always in the solution or if they are never in the solution.

To forbid a vertex means that it is still part of the graph, but we do not ever include this vertex in a solution in the subsequent sub problems. An example of a situation where we would forbid a vertex is when finding all minimal dominating sets in a graph. Let us assume that in a branch we decide that a vertex v should never be in any of the minimal dominating sets generated by this branch. If v is not yet dominated, we can not remove it as we have no guarantee that it will be dominated at a later stage, so we forbid it.

Generating all vertex subsets of a graph is often done with recursion. An example of a recursion algorithm generating all vertex subsets of a graph is given in Algorithm 1.3. This algorithm generates all vertex subsets containing X. To generate all 2^n vertex subsets of a graph we call the algorithm with $X = \emptyset$, that is we call GenerateAllSubsets$(G, \emptyset)$.

The leaves in the computation tree generated by Algorithm 1.3 represent all subsets of vertices of the graph G. In this case we have no need to forbid any vertices as we want to generate all 2^n vertex subsets.

Algorithm 1.3 Algorithm generating all vertex subsets containing X of a graph G

```
1: procedure GenerateAllSubsets(G,X)
2:     if G is empty then
3:         Output X
4:     end if
5:     while G is nonempty do
6:         v := any vertex from G
7:         GenerateAllSubsets(G − v, X ∪ v)
8:         GenerateAllSubsets(G − v, X)
9:     end while
10: end procedure
```

The maximum number of leaves in the computation tree is analysed by looking at the branching steps of the algorithm. If at each branching step we make t new sub-problems where the size of the instance is decreased by $c_1, c_2, ..., c_t$ in each respective sub-problem. Then we obtain the recurrence $T(n) \leq T(n-c_1)+T(n-c_2)+...+T(n-c_t)$ for the number of leaves, assuming that $T(1) = 1$. This recurrence has the *branching vector* $(c_1, c_2...c_t)$. We use this information to give the equation $x^n - x^{n-c_1} - ... - x^{n-c_t} = 0$. Let α be the unique positive real root of this equation. The number α is called the *branching number* of the branching vector. The maximum number of leaves is at most α^n, which gives us an upper bound on the number of objects we want to generate. If the algorithm performs a polynomial number of operations at each branching and reduction step then the total running time is given by $O^*(\alpha^n)$, where O^*-notation suppresses polynomial factors. When we branch in different ways, the running time is given by the worst case branching. Meaning that the branching vector with the least progress is used to find the running time of the algorithm. For Algorithm 1.3 we get the branching vector $(1, 1)$ which gives the equation $x^n - x^{n-1} - x^{n-1} = 0$. The unique positive real root of this equation gives the branching number 2. Therefore the maximum number of leaves produced by Algorithm 1.3 is 2^n giving a total running time of $O^*(2^n)$.

The approach to branching that is described above is often called *branch and reduce*. In branch and reduce the size of the sub-problem is typically the number of vertices in the sub-graph corresponding to it. A different approach to branching called *measure and conquer* has been showed to be very beneficial. The branching algorithms with the currently best known

running times for solving particular NP-hard problems, that have been found in the last decade, have been found using the measure and conquer method or related approaches [8].

The main difference between the measure and conquer method and the branch and reduce approach is that the measure of how much we can reduce the size, or the measure, of the graph in each branch can be chosen with more freedom with measure and conquer [8]. With this approach it is possible to reduce the size of the graph by a number between 0 and 1, when we delete and forbid vertices. This makes it possible to lower bound the progress made by the algorithm at each branching step.

The measure of a vertex is chosen carefully, exploiting how the algorithm works. A basic example is for the MAXIMUM INDEPENDENT SET problem where the measure of vertices with degree ≤ 1 is set to 0, vertices with degree 2 is set to 0.5 and vertices with degree ≥ 3 is set to 1. This improves the running time of the algorithm from $O(1.3803^n)$ to $O(1.3248^n)$ [8]. It is possible to improve this running time more by choosing the measure even more carefully, giving a running time of $O(1.2905^n)$ [8].

1.4 Overview of This Thesis

In this thesis we will study:

- Maximum number of minimal dominating sets in chordal graphs.
- Maximum number of minimal subset feedback vertex sets in chordal graphs.
- Maximum number of minimal subset feedback vertex sets in split graphs.

To study the maximum number of minimal dominating sets in chordal graphs, we will implement an algorithm that finds and list all minimal dominating sets in chordal graphs. This algorithm was given by Couturier et al. in [2], and it gives the best known upper bound on the number of minimal dominating sets in chordal graphs: 1.6181^n. One goal is to see whether we can find a graph that gives a better lower bound than 1.4422^n for the number of minimal dominating sets in chordal graphs. A second goal is to study how the algorithm behaves in practice, and analyse this behaviour to see whether it is possible to improve the upper bound in some way.

For the study of the number of minimal subset feedback vertex sets in chordal and split graphs we will use an algorithm given by Golovach et al. that was given in [12]. This algorithm gives the best known upper bound on the number of minimal subset feedback vertex sets in chordal and split graphs: 1.6708^n. We will use this algorithm to find and list all minimal subset feedback vertex sets in a selection of chordal and split graphs. This

data will then be used to see whether it is possible to improve the current upper bound, or to see if we can find a better lower bound example than the currently best known lower bounds. In fact for split graphs we will see that there exists a graph that gives a better lower bound than the currently best known lower bound.

The algorithms that we implement are based on branching. To test the algorithms we will generate a large number of graphs, this will be done with the help of a graph generating tool called `geng` given by [18]. All graphs with upto 11 vertices, and a selection of both dense and sparse graphs with upto 15 vertices, will be generated.

This thesis is structured as follows:

In the first chapter we have given some general background information on graph classes, branching algorithms and notation that is used in the thesis. This will be important for the next chapters.

In Chapter 2 we will explain both the Dominating Set problem and the Subset Feedback Vertex Set problem. And what results have been found in terms of current upper and lower bounds for the number of minimal dominating sets in chordal graphs and for the number of minimal subset feedback vertex sets in chordal and split graphs.

In Chapter 3 we will first explain how we implemented the algorithm for finding and listing all minimal dominating sets that was given by Couturier et al in [2]. We will then present and analyse the test results given by this algorithm.

In Chapter 4 we will explain how we implemented the enumeration algorithm given by Golovach et al. in [12] for finding an listing all minimal subset feedback vertex sets in chordal and split graphs. Then we will present the test results we have achieved with this implementation for chordal and split graphs.

In Chapter 5, the last chapter, we will present the conclusion of this thesis. We will summarize our work and propose a few questions for further study.

Chapter 2

Listing all Minimal Dominating Sets and Minimal Subset Feedback Vertex Sets

2.1 Minimal Dominating Sets

Dominating set is one of the most classical and important NP-complete problems. It has been studied in various forms over the years and has many practical applications [14]. One of the earliest studies of domination dates back to 1862 and considered how many queens were necessary to cover, or dominate, an $n \times n$ chessboard [14]. But it was not until the late 1950s, early 1960s, that the mathematical study of the problem appeared. When the Dominating Set problem was shown to be NP-complete in 1979 by Garey and Johnson [19] it inspired many to study this problem and now the number of papers on the topic, is in the thousands.

In this thesis we will study an algorithm for finding and listing all minimal dominating sets in chordal graphs given by Couturier et al. in [2]. Note that finding a single minimal dominating set in a graph can easily be done in polynomial time, but a graph normally has an exponential number of minimal dominating sets. The algorithm that we study is an enumeration algorithm, meaning that it finds and lists all minimal dominating sets in an n-vertex graph. This algorithm works specifically for chordal graphs.

Finding a minimum dominating set when we have listed all dominating sets is trivial, we can simply pick the smallest set. Observe that we do not even have to list all dominating sets, but simply all the minimal dominating sets as a minimum dominating set is by definition also minimal.

The fastest known algorithm for finding and listing all minimal dominating sets in general graphs was given by Fomin et al. in 2008, and runs in

$O(1.7159^n)$ time [7]. Note that this is not the fastest algorithm for solving the Dominating Set problem. The currently fastest known algorithm for this was given by Iwata and runs in $O(1.4864^n)$ time and polynomial space, or $O(1.4689^n)$ time and space [16].

The algorithm by Fomin et al., on general graphs, is not the fastest for solving the Dominating Set problem, but it is the fastest known algorithm for enumerating all minimal dominating sets in general graphs. Specifically it gives the best known upper bound on the number of minimal dominating sets in general graphs.

When we mention upper bounds on the number of minimal dominating sets in graph classes we mean how many minimal dominating sets there could at most be in a graph. This upper bound is often given by a branching algorithm. Specifically it is given by how many vertex subsets the algorithm can theoretically find and list. The lower bound on the number of minimal dominating sets in a graph is given by an example of a graph with the largest number of minimal dominating sets that we know of. We give a lower bound example by finding a graph G' with a high number of minimal dominating sets. We then construct a graph G consisting of $|G|/|G'|$ copies of G'. The motivation for this is that the number of minimal dominating sets in G is then the product of the number of minimal dominating sets of all its components. More specifically, assume that G is a disconnected graph, and $G_1, G_2, ..., G_k$ are its components. Let $t_1, t_2, ..., t_k$ be the number of minimal dominating sets in $G_1, G_2, ..., G_k$ respectively. Then the number of minimal dominating sets in G is $t_1 \cdot t_2 \cdot ... \cdot t_k$.

We want the known upper and lower bounds to be equal, but this is not always the case. We can see in Table 2.1 that the known bounds for the number of minimal dominating sets are not equal. Not for general graphs, and not for chordal graphs.

Graph class	Lower bound	Upper bound
General	1.5704^n	1.7158^n
Chordal	1.4422^n	1.6181^n

Table 2.1: Table giving the upper and lower bounds for the number of dominating sets in graph classes

Notice that the gap between the known bounds on chordal graphs is actually bigger than the gap between the known bounds on general graphs. In these cases we want to find either, an example of a graph with a larger number of minimal dominating sets, or we want to improve the algorithm, to achieve a better upper bound.

Observe that having tight bounds can result in faster algorithms for seemingly unrelated problems. An example of this is an algorithm by Lawler, that was the fastest known algorithm for Graph Colouring for 25 years.

It used the bounds on the number of independent sets in general graphs to solve the problem [17].

The algorithm by Couturier et al. [3], gives us the currently best known upper bound on the number of minimal dominating sets in chordal graphs. As mentioned, the gap between the known upper and lower bounds on chordal graphs is actually larger than the gap between the known bounds on general graphs. Therefore we want to see whether it is possible to improve these bounds.

A graph achieving the currently best known number of minimal dominating sets for general graphs, is given in Figure 2.1. It has $15^{n/6} \approx 1.5704^n$ minimal dominating sets. This graph demonstrates the method of achieving a high lower bound as described above. Note that this graph is not chordal, as it contains a cycle of length 4 without a chord.

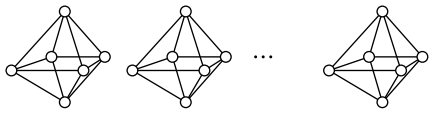


Figure 2.1: A graph with $1 - 5704^n$ minimal dominating sets.

For chordal graphs we find the lower bound in the same way. An example of a graph giving the currently largest number of minimal dominating sets for chordal graphs is given by Figure 2.2 and has $3^{n/3} \approx 1.4422^n$ minimal dominating sets.

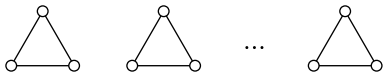


Figure 2.2: A graph with 1.4422^n minimal dominating sets.

In this thesis we will study the algorithm by Couturier et al., and analyse how it works on chordal graphs. Specifically we want to see whether it is possible to find a graph that gives a higher lower bound for chordal graphs than the bound we already have. We also want to see what output the algorithm gives and what information this will give us about the upper bound.

There is a possibility that the given upper bound is too high, which motivates us to study the behaviour of the algorithm. This is to see whether there is some behaviour that is not captured by the algorithm, and that could help us improve the upper bound. It could also be that the lower bound is too low, meaning there could exist an example of a graph that has more minimal dominating sets than the currently best known lower bound example. This motivates us to look for a graph that achieves a better lower

bound.

Table 2.2 gives an overview over how many minimal dominating sets there would have to be in a chordal graph to achieve at least the current lower bound. In the table we let s denote the number of vertices in a graph, t is the number of dominating sets needed in the graph to achieve at least the current lower bound. And n is the total number of vertices in a lower bound example.

s	t	lower bounds
3	3	$3^{n/3} \approx 1.4422^n$
4	5	$5^{n/4} \approx 1.4953^n$
5	7	$7^{n/5} \approx 1.4758^n$
6	9	$9^{n/6} \approx 1.4422^n$
7	13	$13^{n/7} \approx 1.4425^n$
8	19	$19^{n/8} \approx 1.4449^n$
9	27	$27^{n/9} \approx 1.4422^n$
10	39	$39^{n/10} \approx 1.4424^n$

Table 2.2: Table showing the number of minimal dominating sets needed on chordal graphs of s vertices to achieve at least the current lower bound.

2.2 Minimal Subset Feedback Vertex Sets

The Subset Feedback Vertex Set problem was first introduced by Even et al. in a paper where they find a polynomial time approximation algorithm for finding a minimum subset feedback vertex set [5]. The motivation for working on this was the many important and practical uses of solving Subset Feedback Vertex Set, such as its applications in genetics, circuit testing and artificial intelligence [5].

Subset Feedback Vertex Set is an interesting problem, even more so as it is a generalization of two NP-complete problems [6]. If we set $S = V$ we get an instance of the Feedback Vertex Set problem [19], where given a graph G we are asked to find a minimum sized set of vertices such that when these are removed from G, the remaining graph is acyclic. If we set $|S| = 1$ we get an instance of the Multiway Cut problem [9], that given a graph G and a set $T \subseteq G$ called *terminals*, asks for a set of edges of minimum weight that disconnects every pair of terminals in G.

We mentioned in the previous section that to find a minimum dominating set, it is sufficient to list all minimal dominating sets. This is also true for finding a minimum subset feedback vertex set.

The fastest known algorithm for the Subset Feedback Vertex Set problem on general graphs uses time $O(1.8638^n)$ [9]. This is not only the

fastest known algorithm for solving Subset Feedback Vertex Set, but also the fastest known algorithm for finding and listing all minimal subset feedback vertex sets in general graphs. In consequence this algorithm gives us the currently best known upper bound on the number of minimal subset feedback vertex sets in general graphs.

Though this is the best upper bound that have been found there is still a significant gap between the upper and lower bounds on general graphs. Golovach et al. [12] gave an algorithm for enumerating all minimal subset feedback vertex sets in chordal graphs that significantly lowers the upper bound for chordal graphs. Though this is a great improvement for chordal graphs, these techniques do not transfer to the general case, as the algorithm relies heavily on the structural properties of chordal graphs [12]. The algorithm also gives a better upper bound for split graphs, but here the gap between the upper and lower bounds is greater than for chordal graphs. This is because the currently best known lower bound for split graphs is significantly lower than the currently best known lower bound given for chordal graphs.

In Table 2.3 we can see the currently best known bounds on the number of subset feedback vertex sets in general, chordal and split graphs.

Graph class	Lower bound	Upper bound
General	1.5927^n	1.8638^n
Chordal	1.5848^n	1.6708^n
Split	1.4422^n	1.6708^n

Table 2.3: Table giving the upper and lower bounds for the number of subset feedback vertex sets in graph classes

When constructing a lower bound example for the number of minimal subset feedback vertex sets on chordal graphs, we do it the same way as described for minimal dominating sets. Find a graph G with a large number of minimal subset feedback vertex sets, then copy it $n/|G|$ times. We find the number of sets in the constructed graph as follows: Assume we have an unconnected graph G such that $G_1, G_2, ..., G_k$ are the components of G. We let $t_1, t_2, ..., t_k$ be the number of minimal subset feedback vertex sets in $G_1, G_2, ..., G_k$ respectively. Then the number of minimal subset feedback vertex sets in G is $t_1 \cdot t_2 \cdot ... \cdot t_k$. This way of finding a better lower bound example is not possible for split graphs, as a split graph has to be connected and maintain a partition into a clique and an independent set. We will explain how to achieve an exponential lower bound for split graphs later in this section.

As mentioned, for chordal graphs, we can find an exponential number of minimal subset feedback vertex sets in the same way as we did for finding an exponential number of minimal dominating sets. An example of such a

graph that achieves the currently highest known lower bound, is the disjoint union of complete graphs on 5 vertices. We choose any three of the vertices in every connected component to be in S. Three vertices are chosen to be sure that there can be no more cycles in the component. This is depicted in Figure 2.3, where the red vertices are vertices in S. The graph in Figure 2.3 has $10^{\frac{n}{5}} \approx 1.5848^n$ minimal subset feedback vertex sets. To make a component of the graph acyclic only two vertices can remain as part of the maximal S-forest. Giving $\binom{5}{2} = 10$ possible minimal subset feedback vertex sets. Since we have $n/5$ copies of this connected component. This gives the lower bound $10^{\frac{n}{5}} \approx 1.5848^n$.

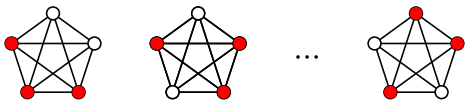


Figure 2.3: A chordal graph with the currently highest number of minimal subset feedback vertex sets.

An example of a split graph achieving an exponential number of minimal subset feedback vertex sets is given in Figure 2.4. The red vertices are the vertices contained in S. This lower bound example is constructed by copying triangles and connecting them in a certain way. We choose one of the vertices in each triangle to be in S. This way each triangle has 3 minimal subset feedback vertex sets. As we will explain below, no vertex in S can be part of the clique of the split graph. As such all vertices in S are in the independent set of the graph. The remaining vertices of each triangle is connected into a large clique. Observe that removing an S-cycle from one triangle will not affect the S-cycles in the other triangles, as these are not dependent on each other. This method can be used for all split graphs. As long as all vertices of S is in the independent set, we can connect all vertices that are in the clique of the graphs to make an even larger clique. And the number of minimal subset feedback vertex sets is the product of all the sets in all these connected graphs. That is given a split graph G with components $G_1, G_2, ..., G_k$ where all vertices in the cliques of these components are connected. Each component has $t_1, t_2, ..., t_k$ minimal subset feedback vertex sets respectively. Then the number of minimal subset feedback vertex sets in G is $t_1 \cdot t_2 \cdot ... \cdot t_k$. In Figure 2.4 each triangle has 3 subset feedback vertex sets and we have $n/3$ triangles in the graph, we get the bound $3^{n/3} \approx 1.4422^n$.

When looking for a better lower bound on split graphs, we cannot choose a vertex from the clique to be part of S. If we did we would get a polynomial number of minimal subset feedback vertex sets as we would have to remove the entire clique except for two vertices to be able to remove all cycles containing a vertex of S. This is because all vertices in the clique are part of all cycles in the clique.

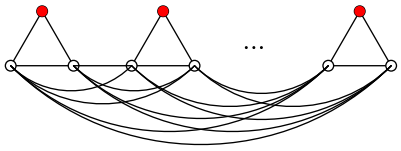


Figure 2.4: A split graph with the currently highest number of minimal subset feedback vertex sets.

In Table 2.4 we give an overview of how many minimal subset feedback vertex sets would have to be in a graph for it to at least achieve the current lower bounds for chordal and split graphs. Again we let s be the size of a graph, t the number of minimal subset feedback vertex sets needed to achieve at least the current lower bound, and n the number of vertices in a lower bound example.

s	t	bounds for chordal graphs	s	t	bounds for split graphs
3	4	$4^{n/3} \approx 1.5874^n$	3	3	$3^{n/3} \approx 1.4422^n$
4	7	$7^{n/4} \approx 1.6265^n$	4	5	$5^{n/4} \approx 1.4953^n$
5	10	$10^{n/5} \approx 1.5848^n$	5	7	$7^{n/5} \approx 1.4757^n$
6	16	$16^{n/6} \approx 1.5874^n$	6	9	$9^{n/6} \approx 1.4422^n$
7	26	$26^{n/7} \approx 1.5927^n$	7	13	$13^{n/7} \approx 1.4425^n$
8	40	$40^{n/8} \approx 1.5858^n$	8	19	$19^{n/8} \approx 1.4449^n$
9	64	$64^{n/9} \approx 1.5874^n$	9	27	$27^{n/9} \approx 1.4422^n$
10	100	$100^{n/10} \approx 1.5848^n$	10	39	$39^{n/10} \approx 1.4424^n$

Table 2.4: Table showing the number of minimal subset feedback vertex sets needed in chordal and split graphs to achieve at least the currently best known lower bound.

In this thesis we test the algorithm by Golovach et al. on both chordal graphs and split graphs. The goal is to see whether there exists a graph that achieves a better lower bound than the currently best known lower bounds for chordal and split graphs. In fact it will be shown in Chapter 4 that there exists a better lower bound for split graphs than the lower bound given by Figure 2.4. In addition we will, as with dominating set, test and see how the practical running time of the algorithm, corresponds with the theoretical running time. We will compare the results for chordal and split graphs and see whether there are any significant differences in how many leaves the algorithm reaches on these different graph classes.

Chapter 3

Minimal dominating sets: Implementation and test results

In this chapter we will give the details of the algorithm for enumerating all minimal dominating sets in chordal graphs by Couturier et al. [2], how we implemented it and what test results we achieved with the implementation. First we will explain the pseudo code of the algorithm, followed by a section with details on choices we took while implementing the algorithm. Then we will present the test results we obtained when we tested our implementation on chordal graphs. Last we will present some suggested improvements to the algorithm.

3.1 The Algorithm

The algorithm by Couturier et al. [2] consists of two reduction rules and two branching rules. We give a pseudo-code for our implementation of this algorithm in Algorithm 3.1. The rules in the algorithm are marked as (1),(2),(3) and (4), respectively. We give the full implementation of Algorithm 3.1 in Appendix B. We will give an informal explanation of the correctness of the rules when we explain them. For full proof of correctness we refer to [2].

The algorithm works as follows: It takes as input a chordal graph $G = (V, E)$, a set $D \subseteq V$ and a PEO α of G. The goal is to generate all minimal dominating sets of G that D is a subset of. To find all minimal dominating sets of G we initially run the algorithm with $D = \emptyset$. That is we call the algorithm with $\textsc{Alg}(G, \emptyset, \alpha)$.

For the next steps we define the following operations on α:

$(\alpha - x)$: Remove x from α. x is always the first vertex in α

$(\alpha - X)$: Remove the set X from α. The other vertices of α retain their order in α.

The first step of the algorithm is to check whether G is empty. This is the base-case of the algorithm, meaning this is the check that tests whether we are at a leaf in the computation tree. That G is empty, means that all vertices in G have been processed.

When we reach a leaf in the computation tree we check whether D is a minimal dominating set of G. This is done by verifying that $D - v$ is not a dominating set of G, for every vertex v in D.

Algorithm 3.1 Algorithm for enumerating MDS in chordal graphs

```
1:  procedure ALG(G, D, α)
2:      if G is empty then
3:          if D is a MDS of G then
4:              save D
5:          end if
6:      end if
7:      x:= first vertex in α
8:      if x is isolated then
9:          if x is dominated by D then
10:             ALG(G − x, D,α − x)                          ▷ (1)
11:         else
12:             ALG(G − x, D ∪ {x}, α − x)                   ▷ (2)
13:         end if
14:     else if x is dominated by D then                     ▷ (3)
15:         ALG(G − N_G[x], D ∪ {x},α − N_G[x])
16:         ALG(G − x, D, α − x)
17:     else if x is not dominated by D then                 ▷ (4)
18:         y := an arbitrary neighbour of x
19:         ALG(G − {x, y}, D ∪ {y}, α − {x, y})
20:         ALG(G − y, D, α − y)
21:     end if
22: end procedure
```

If G is non-empty, pick a simplicial vertex of G. This can be done in constant time by picking the first vertex in α, a PEO of G. We generate α as part of the preprocessing step. We will explain the preprocessing step in the next section.

Let x be the first vertex of α. If x is isolated, meaning it has no neighbours in G, proceed with one of the two following reduction rules to decide

what happens to x. If x is dominated remove x from G as described in (1). If x is not dominated add x to D and remove it from G as described in (2). (1) is safe because if x is dominated, and it has no neighbours to dominate, adding x to D would make x redundant, and D could not possibly be minimal. (2) is safe because if x is not dominated by D and it has no neighbours that could dominate it then x would have to be in D, if not G would not be dominated by D.

If x is not isolated, proceed with the branching rules. If x is dominated by D the algorithm branches as described in (3), it creates one branch where x is added to D while removing the closed neighbourhood of x from G. And one branch where it removes x from G. The first branch is safe since x is already dominated by D, adding a neighbour of x to D, would make x redundant in D. This is because any neighbour of x would dominate all other neighbours of x, since x is simplicial. The second branch is safe, as x has already been dominated.

If x is not dominated, let y be any neighbour of x, then branch as described in (4). We generate one branch where y is added to D and x and y are deleted from G. And one branch where y is deleted from G. In the first branch, after y has been added to D, x is dominated. It is not possible for x to have a private neighbour, a vertex only dominated by x, in this branch. And since x is simplicial, $N_{G'}[x] \subseteq N_{G'}[y]$. As such it is safe to remove both x and y from G. The first branch is safe. The second branch is safe as x has not been deleted. And any vertex that dominates x would also dominate y, since x is simplicial and any neighbour of x is also a neighbour of y.

The running time of the algorithm is decided by the branching vectors given by the branching rules. For the branching in (3) we know that x has at least one neighbour and as such we remove at least two vertices from G in the first branch, and we remove one vertex, x, in the second branch. This gives the branching vector $(2, 1)$. For every branching of (4), we know that x has at least one neighbour y. In one branch we remove two vertices, $\{x, y\}$, while in the second branch we remove only one vertex, y. Again we get the branching vector $(2, 1)$.

The branching vector $(2, 1)$ has the branching number 1.61804. This implies a running time of $O^*(1.61804^n) = O(1.6181^n)$. For more in-depth analysis of the running time of the algorithm we refer to [2].

3.2 Implementation Details

As mentioned in the previous section we need to do some preprocessing before testing the implementation of the algorithm. In this section we will first present what was done in the preprocessing step, before we present different choices that were made in terms of the implementation of the algorithm.

To test our implementation of Algorithm 3.1 we first generate graphs. This is done with a graph generating tool called `geng`, which is part of a package called `gtools`, that is distributed by [18]. It can generate all non-isomorphic graphs of size n where n is small. The program, `geng`, is implemented in the programming language C, and outputs the graphs in a compressed format called `graph6`. To be able to use this data we use a second tool, also from the `gtools` package, called `showg`. This interprets the `graph6` format and outputs the graphs in a format we can read into our Java implementation. Note that this tool generates all graphs and not just chordal graphs.

We were able to generate all non-isomorphic graphs with upto 11 vertices with no problem. But we wanted to test our implementation of the algorithm on graphs with upto 15 vertices. Since the number of graphs generated grows exponentially the time it would take to generate all these graphs is too long. In fact; to generate all graphs with 12 vertices would take 285 hours according to tests run by [18]. As a consequence only a selection of graphs for the graphs with 12-15 vertices were generated. For graphs with more than 11 vertices we chose to generate very dense graphs and very sparse graphs. We made this choice based on the output from the graphs with upto 11 vertices. We saw that the most interesting results were achieved in either very dense graphs or very sparse graphs. For example we found that complete graphs generated the most leaves as compared to more sparse graphs with the same number of vertices.

As all graphs were generated and not just chordal graphs, the size of the text files containing the largest graphs was too large for a laptop. Even when generating a selection of graphs as we did for graphs with more than 11 vertices the files were too large. As such we ran our program on `Brake`, a supercomputer at the University of Bergen, to be able to test the algorithm on more graphs. We generated all graphs instead of just chordal graphs to be sure that we generated a variety of graphs. And to be sure that we tested all graphs of a certain size.

The program `geng` generates graphs relatively fast, but it generates, as mentioned, all graphs. We only want to test our implementation on chordal graphs, therefore we had to test whether the graphs were chordal before running our implementation of the algorithm. To test for chordality in graphs we implemented the MCS algorithm and the CHECKPEO algorithm that were described in Chapter 1.2.1 on chordal graphs. That is, for every graph generated we find an ordering of the vertices with the MCS algorithm and check that this ordering is in fact a PEO. If it is a PEO this implies that the graph is chordal and we run the implementation of Algorithm 3.1 on the graph. If the ordering is not a PEO meaning that the graph is not chordal we proceed with the next graph. The PEO, that is generated, is used not only when testing for chordality, but also as input to the implementation of Algorithm 3.1.

We chose to represent the graphs we generated with an adjacency-list. This data structure was chosen as opposed to an adjacency-matrix as the graphs generated varied between being very dense and very sparse.

To make our implementation of Algorithm 3.1 as efficient as possible we chose to keep the graph as a constant field variable. That way we did not have to change the actual graph when running the algorithm. Instead we chose to send the set of vertices we have already processed, as input to the algorithm. We implemented the method header as follows:

```
public void finn_mds(HashMap<Integer,Integer> slettaNoder,
  ArrayList<Integer> dSet, LinkedHashSet<Integer> p)
```

Where `slettaNoder` is the set of vertices that has already been processed, `dSet` is the set of vertices that has been added to the dominating set, and `p` is the PEO of the graph. We chose to do this as it is more efficient to copy only the list of vertices already processed as opposed to the whole graph minus the vertices already processed. In addition it made it easier to check whether we have reached a leaf in our computation tree. It was necessary to copy either the graph or the list of processed vertices because of how recursion works in Java. If we did not copy the input then the first branch would change the variables which would result in the second branch recurring on the wrong input. We give the method calls depicting how we branch on a vertex x that has already been dominated:

```
finn_mds(removeNeighbourhood(x, slettaNoder),
         addToDomSet(x, dSet),
         removeNeighbourhoodFromPEO(x, p));
finn_mds(removeNode(x, slettaNoderCopy),
         dSetCopy,
         removeFromPEO(x,pCopy));
```

3.3 Test results of our implementation

In this section we will present the test results that was achieved when running the implementation of Algorithm 3.1 on chordal graphs. We will also present some observations that were made based on the output given by the implementation.

We mentioned that one goal of this thesis was to study the upper and lower bounds on the number of minimal dominating sets in chordal graphs. We have not been able to find a better lower bound than the currently best known lower bound; 1.4422^n. But an observation we have made is that there exist graphs that manage to achieve this lower bound that are not isomorphic to the lower bound example that we gave in Chapter 2.1 on minimal dominating sets. It is interesting to note that for all the graphs we have found that manage to achieve this lower bound the number of

vertices in these graphs is divisible by 3. We show examples of this in Figure 3.1. Here the first example (a) consists of connected components of size 3 where each component has 3 minimal dominating sets, giving the bound $3^{n/3} \approx 1.4422^n$. In (b) each connected component has 6 vertices and 9 minimal dominating sets, achieving the bound $9^{n/6} \approx 1.4422^n$. In (c) each connected component consists of 9 vertices and each component has 27 minimal dominating sets. This gives the bound $27^{n/9} \approx 1.4422^n$. In other words all these examples manage to achieve the same lower bound. This observation supports the hypothesis that 1.4422^n could be the upper bound on the number of minimal dominating sets, in chordal graphs. Note that similar results were achieved for graphs of size 12 and 15.

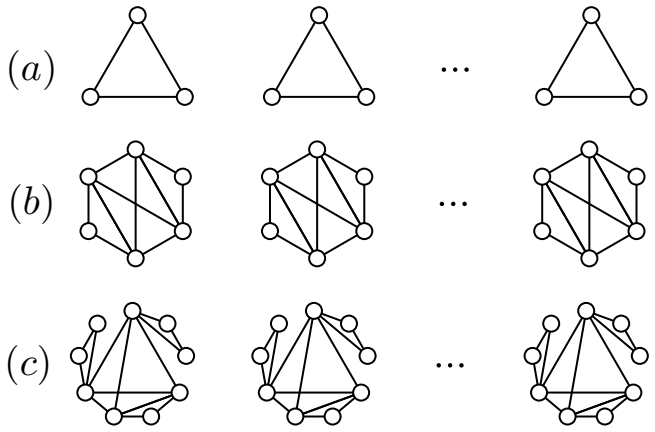


Figure 3.1: Three graphs that achieve the currently best known lower bound for the maximum number of minimal dominating sets in chordal graphs. The graph in (a), has 3 minimal dominating sets in every component. The graph in (b), has 9 minimal dominating sets in every component. The graph in (c), has 27 minimal dominating sets in every component.

We have tested our implementation on more than 5 million graphs. We have generated all graphs with upto 11 vertices, and a selection of graphs with between 12 and 15 vertices. As mentioned not all of these were chordal, and as a consequence we have generated more graphs than we have tested the implementation on. The test results that have been output by our implementation is shown in Table 3.1. This data is also depicted in Figure 3.2. Where the blue line is the running time of the algorithm given by Couturier et al. in [2], meaning the upper bound on the number of minimal dominating sets in chordal graphs. The red line depicts the maximum number of leaves found in graphs of size s. The brown line depicts the maximum number of minimal dominating sets in a graph of s vertices. And the grey line depicts the average number of leaves reached for each graph size. The points on the lines are given by $DS^{n/s}$ where DS is the number of dominating sets and s

is the size of the graph.

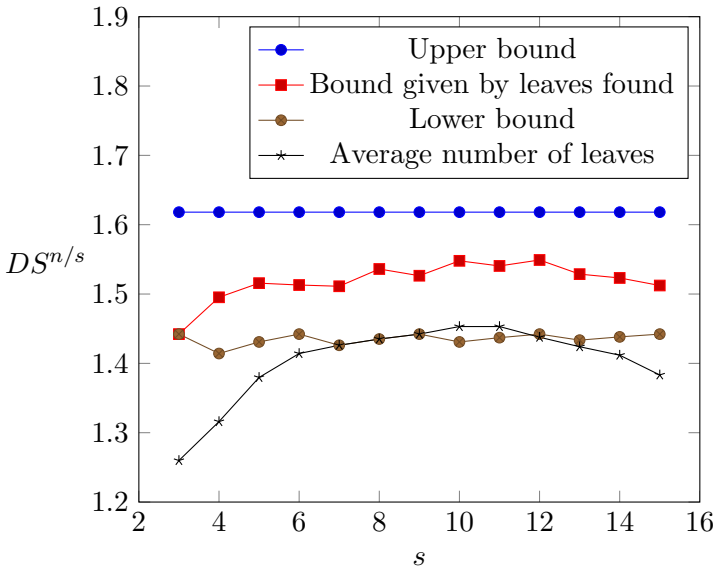


Figure 3.2: A chart depicting the correlation between the upper and lower bounds on the number of minimal dominating sets in chordal graphs, the number of leaves found, and the average number of leaves found in graphs of size s.

In Table 3.1 we give t, the number of minimal dominating sets we would have to find to at least achieve the currently best known lower bound. We give the number of graphs we have tested our implementation on for each graph size, and we give the exact number of minimal dominating sets and the exact number of leaves found. These are given as DS and L respectively.

Notice that the average number of leaves reached for each graph size is generally closer to the maximum number of minimal dominating sets found, as opposed to the maximum number of leaves found. Also the pattern made by the brown line seems to indicate that the lower bound does not grow. In fact it seems as if it stabilizes at 1.4422. There is also a substantial gap between the number of leaves found and the number of minimal dominating sets found. There is even a gap between the number of leaves found and the running time of the algorithm given by the branching vector $(2, 1)$. The gap between the number of minimal dominating sets found and the number of leaves found is especially interesting, as it indicates that the algorithm finds dominating sets that are not minimal. One of the cases where this happens, is when the input graph is a clique. In this case the implementation will branch more than necessary. This is because the algorithm has to process all vertices in the graph, even though a dominating set has already been found. In practice this means that we generate dominating sets that cannot

be minimal.

s	t	L	$L^{1/s}$	DS	$DS^{1/s}$	# graphs	Average # leaves
3	3	3	1.4422	3	1.4422	2	2
4	5	5	1.4953	4	1.4142	5	3
5	7	8	1.5157	6	1.4309	15	5
6	9	12	1.5130	9	1.4422	58	8
7	13	18	1.5112	12	1.4261	272	12
8	19	31	1.5361	18	1.4351	1614	18
9	27	45	1.5264	27	1.4422	11911	27
10	39	79	1.5479	36	1.4309	109539	42
11	57	116	1.5405	54	1.4371	1392387	61
12	81	191	1.5491	81	1.4422	364142	78
13	117	249	1.5287	108	1.4335	1116854	99
14	169	362	1.5232	162	1.4382	2791946	125
15	243	495	1.5123	243	1.4422	2270218	130

Table 3.1: Table showing the number of minimal dominating sets on graphs of s vertices

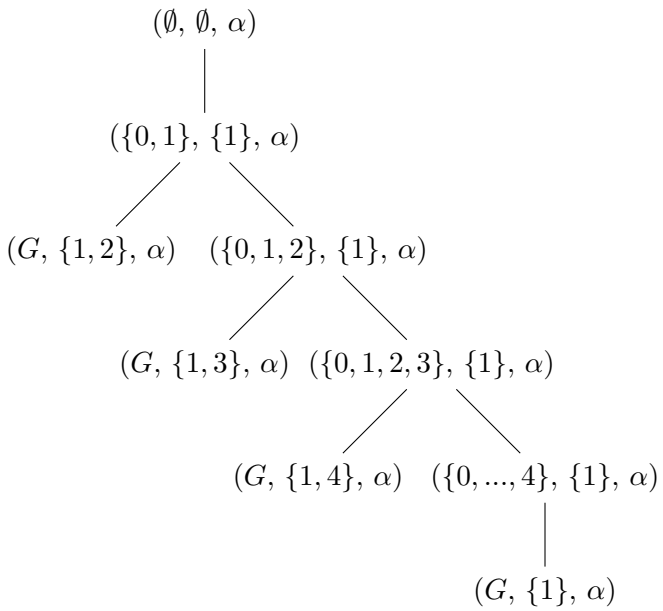


Figure 3.3: Left branch of computation tree, when running our implementation of Algorithm 3.1 on a complete graph of 5 vertices.

In Figure 3.3 we show the left branch of the computation tree built when running Algorithm 3.1 on a complete graph of 5 vertices. The algorithm finds

a dominating set already at the second level. But it continues to look for all possibilities for the remaining vertices. Even though any new vertices added to the dominating set would be redundant. In the figure this branch generates four leaves, but only on leaf gives a minimal dominating set.

These observations support the hypothesis that the upper bound is too high and could be equal to the lower bound of 1.4422^n. They are also what motivated the next section, where we suggest two new reduction rules.

3.4 Suggested improvements

As mentioned in the previous section, one case where the number of leaves in a graph is significantly larger than the number of minimal dominating sets is when the graph is a clique. Therefore we suggest two additional reduction rules for the algorithm:

R3: If D is a dominating set of G, then stop recurring and check whether D is minimal.

The rule checks whether a set D is a dominating set. If the set is a dominating set then stop recurring as a solution has been found. Test whether D is minimal, if it is: save D. This rule is safe because if D is a dominating set of G, then D can never be minimal if we add any other vertices to the set.

R3 will mostly give results in small graphs, but will not be very useful if the graph consists of many connected cliques. Depending on the PEO the algorithm could work its way through one clique at a time and **R3** would not happen. As a consequence of this situation we generalized the rule into reduction rule **R4**.

R4: If $N_G[x]$ is dominated by D, remove x. We call $\textsc{Alg}(G - x, D, \alpha - x)$

The rule checks whether $N_G[x]$ is dominated, and removes x from G if this is true. The rule is safe because if $N_G[x]$ is dominated, then x has no private neighbour in G. This means that x will not dominate any unique vertex in G. As a consequence x would be redundant in D.

In Algorithm 3.2 we show how we would formulate the algorithm to implement these changes.

After running Algorithm 3.2 on the same data as was used when testing Algorithm 3.1, we noticed that the maximum number of leaves reached, for a graph with s vertices, was significantly lowered. This is depicted in Figure 3.4. In addition we noticed that the average number of leaves reached when we run the changed implementation is drastically reduced. For some graph sizes this number is even halved when compared to the number given by the original implementation

Algorithm 3.2 Algorithm for enumerating MDS in chordal graphs with changes

```
1: procedure ALG(G, D, α)
2:     if G is empty or D is dominating set of G then                ▷ (R3)
3:         if D is a MDS of G then
4:             save D
5:         end if
6:     end if
7:     x:= first vertex in α
8:     if x is isolated then
9:         if x is dominated by D then
10:            ALG(G − x, D,α − x)
11:        else
12:            ALG(G − x, D ∪ {x}, α − x)
13:        end if
14:    else if x is dominated by D then
15:        if N_G[x] is dominated by D then                          ▷ (R4)
16:            ALG(G − x, D, α − x)
17:        else
18:            ALG(G − N_G[x], D ∪ {x},α − N_G[x])
19:            ALG(G − x, D, α − x)
20:        end if
21:    else if x is not dominated by D then
22:        y := an arbitrary neighbour of x
23:        ALG(G − {x, y}, D ∪ {y}, α − {x, y})
24:        ALG(G − y, D, α − y)
25:    end if
26: end procedure
```

The changes we have made to the implementation runs in polynomial time. Since we did not change any of the branching rules, we still get the same branching number. Therefore the running time of the changed implementation is the same as for the original implementation. As we run the implementation on small graphs the polynomial factor is not a problem. In terms of practical running time we see a reduction in the number of leaves reached. This supports the hypothesis that the upper bound is in fact too high.

The implemented changes reduced the practical running time of the algorithm. This was observed as the number of leaves found by the algorithm were reduced. This observation in addition to the observations made con-

cerning the lower bound examples, in the previous section, support the hypothesis that the currently best known upper bound could be too high. In fact these observations imply that the upper bound could be equal to the currently best known lower bound; 1.4422^n.

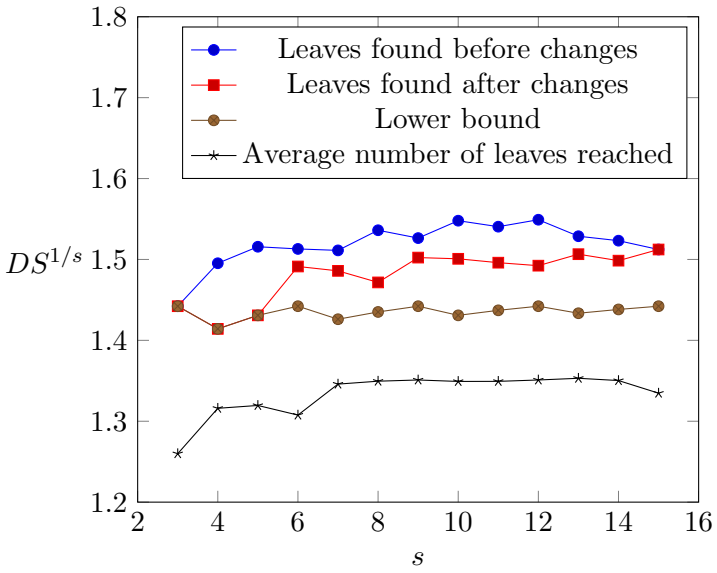


Figure 3.4: Chart showing the result given by Algorithm 3.2 compared with the results given by Algorithm 3.1

s	graphs tested	Minimal dominating sets	Leaves reached before changes	Leaves reached after changes	Average leaves reached
3	2	3	3	3	2
4	5	4	5	4	3
5	15	6	8	6	4
6	58	9	12	11	5
7	272	12	18	16	8
8	1614	18	31	22	11
9	11911	27	45	39	15
10	109539	36	79	58	20
11	1392387	54	116	84	27
12	364142	81	191	122	37
13	1116854	108	249	206	51
14	2791946	162	362	288	67
15	2270218	243	495	495	76

Table 3.2: Table showing test results when implementing Algorithm 3.2.

Chapter 4

Subset Feedback Vertex Sets: Implementation and test results

In this chapter we will first present the algorithm for enumerating all subset feedback vertex sets given by Golovach et al. [12]. Then we will explain details of how we chose to implement the algorithm. At last we will present the test results that we managed to achieve with our implementation of this algorithm. In fact we will present a graph that achieves a better lower bound than the currently best known lower bound for split graphs.

4.1 The Algorithm

In Algorithm 4.1 we give a pseudo-code for the algorithm by Golovach et al. as we implemented it. An implementation of this algorithm is given in Appendix C. Due to the length of the algorithm and for better readability we chose to describe the different cases of the algorithm in separate procedures. These are given in Algorithms 4.2 to 4.8. We will explain each case as given by the branching and reduction rules. As we explain these cases we will also give an informal explanation of the safeness of these rules. For full proof of correctness we refer to [12].

Before we describe Algorithm 4.1 we need to define a few terms that we will use. We let $S \subseteq V$ be the set of vertices in G that can not be part of any cycles when we remove the subset feedback vertex set from G. We let $F \subseteq V$ be a set of vertices that form an S-forest in G. And $U \subseteq V$ is the set of vertices that are deleted from G. The set $R \subseteq F$ is the set of hidden vertices, the vertices that we do not not make any decisions based on. Note that it is necessary to hide vertices to make it possible to always choose a simplicial vertex v, that is either undecided or has a neighbour that is undecided. It is shown in [12] that these vertices have no effect on the

result. We let $G' = V \setminus U \cup R$, and we say that a vertex is undecided if it is not contained in $U \cup F$. The goal is to generate all minimal subset feedback vertex sets that U is a subset of.

Algorithm 4.1 Algorithm for enumerating SFVS in chordal graphs

```
1: procedure ALG(F, U, R)
2:     REDUCE(F,U,R)
3:     if G has no undecided vertices then
4:         if F is maximal S-forest then
5:             Save U
6:             Save F
7:         end if
8:     else
9:         if F is S-forest then
10:            v:= simplicial vertex of G'
11:            if v is undecided then
12:                if v is in S then
13:                    if v has no neighbour in F then
14:                        CASE 1.1
15:                    else
16:                        CASE 1.2
17:                    end if
18:                else
19:                    if v has no neighbours in F then
20:                        CASE 1.3
21:                    else
22:                        CASE 1.4
23:                    end if
24:                end if
25:            else if v is in F then
26:                if v is in S then
27:                    CASE 2.1
28:                else
29:                    CASE 2.2
30:                end if
31:            end if
32:        end if
33:    end if
34: end procedure
```

The algorithm works as follows: We first call the reduce procedure, which is described in Algorithm 4.2. It contains three reduction rules, **Rule A**, **Rule B** and **Rule C**, and it will run until there are no vertices in G' that satisfy these rules. We chose to implement this procedure such that instead of just finding one vertex that satisfy a rule and calling ALG on the changes, we find all vertices that satisfy the rules and remove or decide the vertices as the rules states, without calling ALG on the changes.

Algorithm 4.2 Reduce in Algorithm for enumerating all minimal SFVSs

```
1: procedure REDUCE
2:     while There is change do
3:         for all vertices v in G' do                                  ▷ (Rule A)
4:             if u, w ∈ N(v), u, w ∈ F s.t. uw ∈ E and {u, v, w} ∩ S ≠ ∅
   then
5:                 (F, U ∪ v, R)
6:             end if
7:         end for
8:         for all vertices v in G' do                                  ▷ (Rule B)
9:             if v has d(v) ≤ 1 then
10:                (F ∪ v, U, R ∪ v)
11:            end if
12:        end for
13:        for all vertices v in G' do                                  ▷ (Rule C)
14:            if v is simplicial and N[v] ∩ S = ∅ then
15:                (F ∪ v, U, R ∪ v)
16:            end if
17:        end for
18:    end while
19: end procedure
```

The REDUCE procedure works by first trying to find a vertex that satisfy Rule A, then Rule B and last Rule C. If a vertex is found that satisfy either Rule B or Rule C, then the procedure will act on the vertex as described in the rule before starting again at Rule A. Next we will describe how the different reduction rules work.

Rule A checks whether there exists any vertex v with neighbours u, w contained in F where $\{u, v, w\} \cap S$ is non-empty. If such a vertex v exists, we add v to U. This is safe as we know that $\{u, v, w\}$ form an S-cycle. This means that at least one of them would have to be in U. Since v is the only undecided vertex of the three, it has to be added to U.

Rule B checks whether there exists a vertex v where $d(v) \leq 1$. If v is

undecided, add v to F. When v is in F, hide v by adding it to R. The safeness of this rule comes from the fact that v can never be part of an S-cycle in G. As such it can never be a part of an S-cycle at a later stage when there are fewer vertices to consider either. Since v will never be a part of an S-cycle at a later stage there is no reason to make decisions based on this vertex. As such it is also safe to hide v.

When we try to find a vertex that satisfy Rule C we try to find a simplicial vertex v such that $N_{G'}[v] \cap S$ is empty. If v is undecided, add v to F. When v is in F, add v to R. Due to Rule B, we know that $d(v) \geq 2$. Let F' be a maximal S-forest of G, such that $F \subseteq F'$. If F' contains at most one neighbour of v, then F' must contain v also. In this case v would never be a part of an S-cycle and it must therefore be in F. If F' contains two or more neighbours of v then these neighbours can not be part of any S-cycles in F', and since v is not contained in S, we can safely add v to F. The safeness of hiding v comes from the observation that any S-cycle containing v must contain at least two neighbours u, w of v. Since v is simplicial we know that u and w are adjacent. As such any S-cycle containing v will still be an S-cycle even if we remove v.

After exhaustively running the REDUCE procedure, the algorithm checks whether there are any undecided vertices left in G'. This is done with a simple comparison between the size of $|F \cup U|$ and the size of $|V(G)|$. If there are no more undecided vertices in G', we check whether F is a maximal S-forest. If it is, we save both U and F.

If there are undecided vertices left in G', verify that F is an S-forest and proceed with the branching rules. We let v be a simplicial vertex of G'. v can be found in constant time by using an ordering α of the vertices, and picking the first vertex in this ordering. Note that α is a PEO of G, and it is found in the preprocessing step of the implementation. In the implementation we let α be a field variable. The reason for this and how we maintain a correct PEO will be explained in Chapter 4.2. In the next paragraphs we will explain the different cases that decide how the algorithm branches. Which case used is decided by properties of v.

Case 1.1 If v is undecided, not contained in S and has no neighbours in F, call CASE 1.1, which is described in Algorithm 4.3. We will branch in different ways depending on the size of the neighbourhood of v.

If v has exactly 2 neighbours then we branch in four ways. These branches are given by the different ways we can add v and at most one of its neighbours to F. We also get one branch where we add only the neighbours of v to F. The remaining vertices in $N_G[v]$ are added to U.

If v has exactly 3 neighbours then again we branch on the different ways we can add v and at most one of its neighbours to F while adding the rest to U, or adding v to U. In the latter case there are two possibilities. Either

we add one neighbour to F, or we add one neighbour to U in addition to v and add the remaining neighbours to F.

If v has more than 3 neighbours then the algorithm branches in $d(v)+2$ ways. One branch where v is added to U, one branch where v is added to to F and $N_{G'}(v)$ is added to U, and $d(v)$ branches where it branches on the different ways of adding v and one of its neighbours to F, while adding the rest of the neighbourhood to U.

In each of these branches we add at most two vertices to F and we add the rest to U. If we did not add the remaining vertices to U, or if we added more than two vertices to F we would get an S-cycle in F. Since we generate all the different ways v could or could not be added to F, these branches are safe.

Algorithm 4.3 Case 1.1 in Algorithm for enumerating all minimal SFVSs

```
1: procedure CASE 1.1
2:     if v has 2 neighbours then
3:         Let u_1 and u_2 be the neighbours of v
4:         ALG(F ∪ {u_1, u_2}, U ∪ {v}, R)
5:         ALG(F ∪ {v, u_2}, U ∪ {u_1}, R)
6:         ALG(F ∪ {v, u_1}, U ∪ {u_2}, R)
7:         ALG(F ∪ {v}, U ∪ {u_1, u_2}, R)
8:     else if v has 3 neighbours then
9:         Let u_1, u_2, u_3 be the neighbours of v
10:        ALG(F ∪ {v, u_1}, U ∪ {u_2, u_3}, R)
11:        ALG(F ∪ {v, u_2}, U ∪ {u_1, u_3}, R)
12:        ALG(F ∪ {v, u_3}, U ∪ {u_1, u_2}, R)
13:        ALG(F ∪ {v}, U ∪ {u_1, u_2, u_3}, R)
14:        ALG(F ∪ {u_1}, U ∪ {v}, R)
15:        ALG(F ∪ {u_2, u_3}, U ∪ {v, u_1}, R)
16:    else if v has more than 3 neighbours then
17:        ALG(F, U ∪ {v}, R)
18:        ALG(F ∪ {v}, U ∪ N_{G'}(v), R)
19:        for all Neighbours u_i of v do
20:            ALG(F ∪ {v, u_i}, U ∪ (N_{G'}(v) − u_i), R)
21:        end for
22:    end if
23: end procedure
```

Case 1.2 If v is undecided, contained in S and has at least one neighbour in F, proceed with CASE 1.2, which is described in Algorithm 4.4. In this

case the algorithm branches in two ways. Let u be the neighbour of v that is in F, note that v can only have one neighbour in F or v would have been added to U in Rule A. We generate one branch where v is added to U, and one branch where v is added to F and $N_{G'}(v) - u$ is added to U.

The first branch is safe as v has been added to U, and can therefore never be part of an S-cycle in F. The second branch is safe since v already has one neighbour in F. Adding more neighbours to F after adding v to F would result in an S-cycle in F. Therefore the rest of the neighbours have to be added to U.

Algorithm 4.4 Case 1.2 in Algorithm for enumerating all minimal SFVSs

```
1: procedure CASE 1.2
2:     Let u be neighbour of v in F
3:     ALG(F, U ∪ v, R)
4:     ALG(F ∪ v, U ∪ (N_{G'}(v) − u), R)
5: end procedure
```

Case 1.3 For CASE 1.3 to happen v has to be undecided, not contained in S and it cannot have any neighbours in F. This case is very similar to CASE 1.1. Since v was not handled by any of the reduction rules, v has at least one neighbour contained in S. For $d(v) = 2$ we proceed in the same way as we did in CASE 1.1 when $d(v) = 2$.

When $d(v) = 3$ the algorithm acts in a similar way as it did in CASE 1.1 for $d(v) = 3$. Only instead of branching on v it branches on the different ways it can add a neighbour u_1, that is contained in S, to F.

For the case where $d(v) \leq 4$, let u_1 be a neighbour of v contained in S. The algorithm branches: One branch where u_1 is added to U, one branch where u_1 is added to F and $N_{G'}[v] - u_1$ is added to U. Next it branches for all the combinations u_1 and a vertex from $N_{G'}[v] - u_1$ can be added to F and adding the rest of the closed neighbourhood of v to U.

These branches are safe as we branch on all the different ways to add u_1 to F, at most two vertices are added to F and the rest of the neighbourhood to U. This is safe because adding more than two vertices to F would produce an S-cycle in F so these vertices have to be in U.

Case 1.4 If v is undecided, not contained in S and has at least one neighbour in F we proceed with CASE 1.4. This procedure is described in Algorithm 4.6. As v has not been decided by any of the reduction rules we know that v has exactly one neighbour u in F. We branch in different ways depending on whether u is contained in S or not. If u is contained in S then at most one vertex from $N_{G'}[v] - u$ can be added to F. Therefore the algorithm branches on all the different ways we can add a vertex from $N_{G'}[v] - u$

to F, and deleting the rest of the neighbourhood from G' by adding it to U. This is safe as adding any more of the neighbours to F would result in an S-cycle in F, so these have to be in U.

Algorithm 4.5 Case 1.3 in Algorithm for enumerating all minimal SFVSs

```
1: procedure CASE 1.3
2:     if v has 2 neighbours then
3:         Let u_1, u_2 be neighbours of v
4:         ALG(F ∪ {u_1, u_2}, U ∪ v, R)
5:         ALG(F ∪ {v, u_2}, U ∪ u_1, R)
6:         ALG(F ∪ {v, u_1}, U ∪ u_2, R)
7:         ALG(F ∪ v, U ∪ {u_1, u_2}, R)
8:     else if v has 3 neighbours then
9:         Let u_1, u_2, u_3 be neighbours of v
10:        ALG(F ∪ {v, u_1}, U ∪ {u_2, u_3}, R)
11:        ALG(F ∪ {u_1, u_2}, U ∪ {v, u_3}, R)
12:        ALG(F ∪ {u_1, u_3}, U ∪ {v, u_2}, R)
13:        ALG(F ∪ {u_1}, U ∪ {v, u_2, u_3}, R)
14:        ALG(F ∪ {v}, U ∪ {u_1}, R)
15:        ALG(F ∪ {u_2, u_3}, U ∪ {v, u_1}, R)
16:    else if v has more than 3 neighbours then
17:        Let u_1 be a neighbour of v in S
18:        ALG(F, U ∪ v, R)
19:        ALG(F ∪ u_1, U ∪ (N_G'[v] − u_1), R)
20:        ALG(F ∪ {v, u_1}, U ∪ (N_G'(v) − u_1), R)
21:        for all Neighbours u_i of v except u_1 do
22:            ALG(F ∪ {u_1, u_i}, U ∪ (N_G'[v] − {u_1, u_i}), R)
23:        end for
24:    end if
25: end procedure
```

If u is not contained in S then, due to the reduction rules, v has another neighbour w that is contained in S. The algorithm branches in two ways. One branch where w is added to F and $N_{G'}[v] - \{u, w\}$ is added to U. And one branch where w is added to U. The first branch is safe as adding any more vertices from the closed neighbourhood of v to F would result in an S-cycle in F so these vertices have to be added to U. The second branch is safe since w is in U, and can therefore never be part of an S-cycle in F.

Algorithm 4.6 Case 1.4 in Algorithm for enumerating all minimal SFVSs

```
1: procedure CASE 1.4
2:     Let u be a neighbour of v
3:     if u is in S then
4:         for all Vertices x in N_{G'}[v] − u do
5:             ALG(F ∪ x, U ∪ (N_{G'}[v] − {u, x}), R)
6:         end for
7:     else
8:         Let w be a neighbour of v in S
9:         ALG(F ∪ w, U ∪ (N_{G'}[v] − {u, w}), R)
10:        ALG(F, U ∪ w, R)
11:    end if
12: end procedure
```

Case 2.1 When v is in F and in S, proceed with CASE 2.1. This case is described in Algorithm 4.7. Due to the reduction rules we know that $N_{G'}(v) \cap F$ is empty, and since v is in F we can add at most one neighbour of v to F.

If $d(v) = 2$, let u and w be the neighbours of v. The algorithm branches on whether to add u to F and w to U, or to add u to U. The first branch is safe as w has to be added to U after adding u to F. If not we would get an S-cycle in F. The second branch is safe as u is deleted from G', and thus u can never be part of an S-cycle in F.

Algorithm 4.7 Case 2.1 in Algorithm for enumerating all minimal SFVSs

```
1: procedure CASE 2.1
2:     if v has 2 neighbours then
3:         Let u_1, u_2 be neighbours of v
4:         ALG(F ∪ u_1, U ∪ u_2, R)
5:         ALG(F, U ∪ u_1, R)
6:     else if v has more than 2 neighbours then
7:         for all Vertices u_i in N_{G'}(v) do
8:             ALG(F ∪ u_i, U ∪ (N_{G'}(v) − u_i), R)
9:         end for
10:        ALG(F, U ∪ N_{G'}(v), R)
11:    end if
12: end procedure
```

If $d(v) > 2$ the algorithm branches on all the different ways we can add at most one neighbour of v to F, and adding the remaining neighbourhood

to U. Since v is already in F, and it is in S, at most one neighbour of v can be added to F. If more neighbours of v were added to F, there would be an S-cycle in F. Therefore the remaining neighbourhood must be added to U.

Case 2.2 If v is in F but not in S we proceed with Case 2.2. This procedure is described in detail in Algorithm 4.8. For this case we branch in two ways. We let u_1 be an undecided neighbour of v that is in S. We branch on either adding u_1 to U or adding u_1 to F and $N_G(v) - u_1$ to U. The first branch is safe as we simply remove u_1 from G', as such u_1 can never be part of an S-cycle in F. In the second branch we add u_1 to F and since v is already in S we cannot add any more neighbours of v to F without creating an S-cycle in F so we have to add the rest of the neighbourhood to U. So this rule is safe.

Algorithm 4.8 Case 2.2 in Algorithm for enumerating all minimal SFVSs

```
1: procedure CASE 2.2
2:     Let u_1 be undecided vertex, neighbour of v
3:     ALG(F, U ∪ u_1, R)
4:     ALG(F ∪ u_1, U ∪ (N_{G'}(v) − u_1), R)
5: end procedure
```

The algorithm branches until it has reached all possible solutions. If these solutions are minimal subset feedback vertex sets we count and store them. The worst case branching vector for this algorithm is $(4, 4, 4, 4, 2, 4)$. This branching vector gives the branching number 1.6708 which in turn implies a running time of $O(1.6708^n)$. For more in-depth analysis of the running time we refer to [12].

4.2 Implementation Details

In this section we will describe the different choices that were made when implementing Algorithm 4.1. We will first describe choices that were made specific to the implementation of the algorithm, then the choices that were made concerning the implementation when run on chordal graphs. And last we will describe choices that were made specifically concerning the implementation when run on split graphs. We will not describe how we generated graphs as this was done the same way as described in the previous chapter, but we will mention what graphs that were generated.

Unlike the implementation of Algorithm 3.1 we chose to implement the PEO as a field variable instead of giving it as input to the implementation. This way we find the PEO by removing all vertices contained in $U \cup R$ from the ordering. Note that we do not change the order of the vertices in the

PEO. This was done to avoid manipulating the PEO for all the different branches that are generated at each level. The graph G and the set S, were also kept as field variables. This was because these sets of vertices do not change throughout the algorithm. The method header is given below to give an insight into how a call to Algorithm 4.1 would work in practise. Here `f` is an S-forest of G, `sfvs` is the set of deleted vertices in G and `r` is the set of hidden vertices.

```
public void alg(HashMap<Integer, Integer> f,
                HashMap<Integer,Integer> sfvs,
                HashMap<Integer, Integer> r)
```

For each solution generated by the implementation of Algorithm 4.1, we test whether this is in fact a minimal subset feedback vertex set. This is done by verifying that it is not possible to remove any vertices from U, without making an S-cycle in F. That is; we test that $U - v$ is not a subset feedback vertex set of G for every v in U. To check whether a set U is a subset feedback vertex set of G, we implemented a depth-first-search, that specifically searched for any cycles in $G - U$ that contained at least one vertex of S.

We checked for chordality in graphs the same way as described in the previous section, by generating an ordering α with the MCS-algorithm given in Algorithm 1.1, and then verifying that α is in fact a PEO with the CHECK-PEO algorithm given in Algorithm 1.2.

We wanted to test the implementation of Algorithm 4.1 for all different choices of S. As a consequence we could not, for chordal graphs, test the implementation on as many graphs as we did in the previous section. The algorithm was run up to 2^n times for each graph of size n that we tested it on. This was because all unique sets of S were generated. Note that we did not include the empty set for S as then the solution is trivial. As the implementation is tested for all choices of S, we limited the graphs that were generated to very dense graphs. We chose very dense graphs as the most interesting results found in already tested graphs were found in very dense graphs. As an example the currently best known lower bound example, for chordal graphs, consists of $n/5$ disjoint copies of the complete graph on 5 vertices.

We implemented Equation 1.1 that was given in Theorem 4 in Chapter 1.2.2. This equation was implemented to check whether a graph is a split graph or not. It works by identifying the last vertex that could be part of the clique in the graph using the index of this vertex in a sorted degree sequence to solve the equation.

We mentioned in Chapter 2.2 that the set S can not contain any vertices from the clique of the split graph. Therefore we know that there are many sets of S that there is no point on testing our implementation for. We sorted out irrelevant sets of S with the same concepts used when recognizing

whether a graph is a split graph. Since we knew that a graph was a split graph, we used the same index w, that is found in Equation 1.1 to identify which vertices were part of the clique and then removing any sets that contained these vertices. As a consequence we were able to test the algorithm on larger data sets than we did for chordal graphs when $s \geq 11$. Note that since split graphs are a sub class of chordal graphs there are naturally more chordal graphs than there are split graphs for a given graph size.

4.3 Test results of our implementation

In this section we will present the results we have achieved when running our implementation of Algorithm 4.1 on chordal and split graphs. In particular the test results for split graphs are very interesting as we managed to find a graph that achieves a better lower bound than the currently best known lower bound.

4.3.1 Test results for chordal graphs

The bounds on the number of minimal subset feedback vertex sets in chordal graphs are very close, though not equal. In Table 4.1 we give the output of our implementation when run on graphs of upto 15 vertices. In this table, s is the size of the graphs. We give the number of chordal graphs we have tested our implementation on, and the number of minimal subset feedback vertex sets that are needed to achieve at least the currently best known lower bound. Further we give the number of leaves that the implementation has reached, and the number of minimal subset feedback vertex sets that were found for each graph size. In addition we show the lower bounds we were able to achieve for each graph size. This bound is given by $SFVS^{n/s}$ where $SFVS$ is the maximum number of minimal subset feedback vertex sets we have found in graphs of size s. In Figure 4.2 we compare the upper bound given by Golovach et al. in [12], the number of leaves found by our implementation and the lower bound given by the number of minimal subset feedback vertex sets we have found for each graph size. Notice that we also manage to achieve the currently best known lower bound in a graph with 10 vertices. This graph is shown in Figure 4.1.

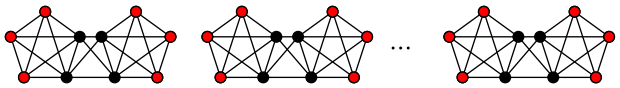


Figure 4.1: A graph with 1.5848^n minimal subset feedback vertex sets

The red vertices are vertices in S, and each connected component of the graph has 100 minimal subset feedback vertex sets. Assuming there are $n/10$

connected components in the graph, it achieves $100^{n/10} \approx 1.5848^n$ subset feedback vertex sets.

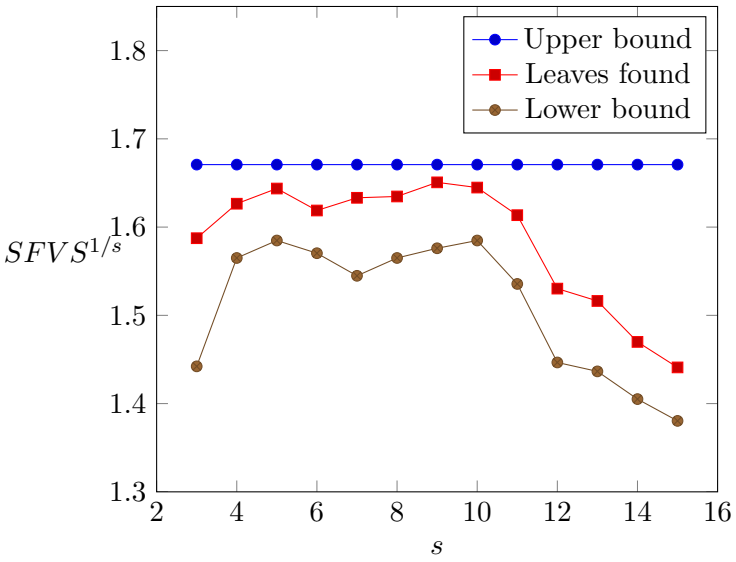


Figure 4.2: Chart showing the result given by Algorithm 4.1 as given by 4.3.

s	# graphs	# sfvs needed	# leaves	# minimal SFVS	$SFVS^{n/s}$
3	2	4	4	3	1.4422^n
4	5	7	7	6	1.5650^n
5	15	10	12	10	1.5848^n
6	58	16	18	15	1.5704^n
7	272	26	31	21	1.5448^n
8	1614	40	51	36	1.5650^n
9	11911	64	91	60	1.5760^n
10	109539	100	145	100	1.5848^n
11	34736	159	193	112	1.5356^n
12	457	252	165	84	1.4466^n
13	1384	398	224	111	1.4365^n
14	457	631	220	117	1.4051^n
15	265	1000	240	126	1.3804^n

Table 4.1: Table showing the test result for Algorithm 4.1 on split graphs

When we analyse the results depicted in Figure 4.2 we can see that the number of leaves found is close to the upper bound. But what is more interesting is the gap between the number of leaves found and the lower bound for each graph size.

The results we have achieved on graphs of size ≥ 11 show a drastic decline in both the number of leaves found and the number of minimal subset feedback vertex sets. We believe this was due to the small selection of graphs that was used when testing our implementation, when the number of vertices in the graph was ≥ 11. This indicates that the most interesting graphs for graphs of size ≥ 11 are not as dense as we believed when generating graphs.

The observations we have made concerning the upper and lower bounds on the number of minimal subset feedback vertex sets in chordal graphs, indicate that the upper bound could be too high. This is mainly based on the observation that the algorithm produces more leaves than it produces minimal subset feedback vertex sets.

4.3.2 Test results for split graphs

As was mentioned in the introduction of this chapter we have managed to find a better lower bound on the number of minimal subset feedback vertex sets in split graphs. In fact we were able to find three examples that achieve a better lower bound.

Before we present these examples we repeat how to construct a lower bound example for split graphs. For chordal graphs it is possible to construct a lower bound example by making n/s disjoint copies of a graph on s vertices. This is not possible for split graphs as a split graph has to be connected. But we construct a lower bound example in a similar way, only instead of the n/s copies being disjoint we connect all the vertices of the cliques in each copy into an even larger clique. As mentioned in Chapter 2.2 none of the vertices in the set S can be in the clique. They have to be in the independent set. Due to this any cycles we break in one copy will not affect any cycles in a different copy. Even though they are connected through the clique. Next we will give the graphs that achieve better lower bounds than the currently best known lower bound.

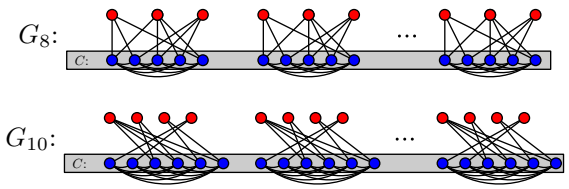


Figure 4.3: Figure showing examples of a graphs achieving better lower bounds than 1.4422^n

We found one graph G_8 with 8 vertices and 20 minimal subset feedback vertex sets that could be used to construct a lower bound example that gives the bound 1.4542^n. We also found one graph G_{10} with 10 vertices and 39 minimal subset feedback vertex sets that can be used to construct a lower

bound example with 1.4424^n minimal subset feedback vertex sets. These graphs are both depicted in Figure 4.3. We note that all the vertices of the grey area in these figures are connected as a clique. In addition all the red vertices are vertices contained in S and in the independent set of the graph.

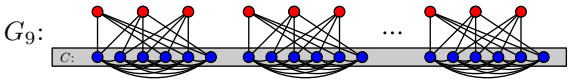


Figure 4.4: Figure showing an example of a graph achieving the currently best known lower bound: 1.4645^n

The graph giving the best lower bound that we were able to find is constructed from a graph with 9 vertices that has 31 minimal subset feedback vertex sets, giving the new best known lower bound 1.4645^n. This lower bound example is given as G_9 in Figure 4.4. Note that to achieve this bound the entire independent set is contained in S. In Figure 4.5 we show a detailed view of the graph we use to achieve the new currently best known lower bound. The red vertices are the vertices in the independent set and they are all contained in S. The actual minimal subset feedback vertex sets for this figure are given in Table 4.2.

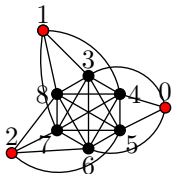


Figure 4.5: Figure showing the graph that is used to construct the new lower bound example in Figure 4.4

$[0,1,2]$	$[1,2,3,4,5]$	$[1,2,4,5,6]$	$[1,2,3,5,6]$	$[1,2,3,4,6]$
$[0,2,4,7,8]$	$[0,2,3,7,8]$	$[2,3,4,6,8]$	$[0,2,3,4,8]$	$[2,3,4,5,8]$
$[2,3,4,6,7]$	$[0,2,3,4,7]$	$[2,3,4,5,7]$	$[3,5,6,7,8]$	$[4,5,6,7,8]$
$[0,1,6,7,8]$	$[0,3,6,7,8]$	$[0,4,6,7,8]$	$[3,4,6,7,8]$	$[0,1,5,7,8]$
$[0,3,5,7,8]$	$[0,4,5,7,8]$	$[3,4,5,7,8]$	$[1,4,5,6,8]$	$[0,1,5,6,8]$
$[1,3,5,6,8]$	$[3,4,5,6,8]$	$[1,4,5,6,7]$	$[0,1,5,6,7]$	$[1,3,5,6,7]$
$[3,4,5,6,7]$				

Table 4.2: All minimal subset feedback vertex sets for the graph depicted in Figure 4.5.

With this new lower bound example we have managed to answer one of the open questions given by Golovach et al. in [12]. The question on

whether there exist a split graph with a better lower bound than 1.4422^n, a bound achieved by the graph in Figure 4.6.

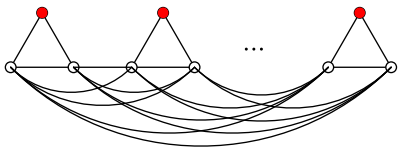


Figure 4.6: The graph giving the previously best known lower bound for the number of minimal subset feedback vertex sets in split graphs

s	# graphs	# sfvs needed	# leaves	# minimal SFVS	$SFVS^{n/s}$
3	2	3	4	3	1.4422^n
4	5	5	6	4	1.4142^n
5	12	7	6	5	1.3797^n
6	35	9	16	9	1.4422^n
7	168	13	24	12	1.4261^n
8	393	19	36	20	1.4542^n
9	1666	27	64	31	1.4645^n
10	8543	39	96	39	1.4424^n
11	5473	57	101	46	1.4163^n
12	6038	81	256	81	1.4422^n
13	1009	117	122	64	1.3770^n
14	1412	169	89	58	1.3364^n
15	159	243	1024	243	1.4422^n

Table 4.3: Table showing the test result for Algorithm 4.1 on split graphs

In Table 4.3 we give an overview of what we have found when running our implementation on split graphs. We let s be the size of the graphs. We give the number of graphs that we have tested the implementation on for every size s. We give the number of leaves found, and we give the maximum number of minimal subset feedback vertex sets we have found. We also give the bound $SFVS^{n/s}$. Where $SFVS$ is the maximum number of minimal subset feedback vertex sets, we have found in graphs of size s. These results are depicted in Figure 4.7.

We believe that the drastic decline in the maximum number of minimal subset feedback vertex sets for graphs of size ≥ 11 was because we were not able to test the implementation on all graphs for these sizes. We observed for chordal graphs that the graph examples that achieve the best bounds are from graphs that are not necessarily very sparse or very dense. This seems to be true for split graphs as well. As such it could be that the

results output by the implementation of Algorithm 4.1, for these graphs sizes, are not representative of how many leaves and how many minimal subset feedback vertex sets exist in these graph sizes. In fact the positive results we achieved for graphs with size 8, 9 and 10 support the notion that there could exist more graphs achieving a better lower bound than 1.4422^n when $s \geq 11$.

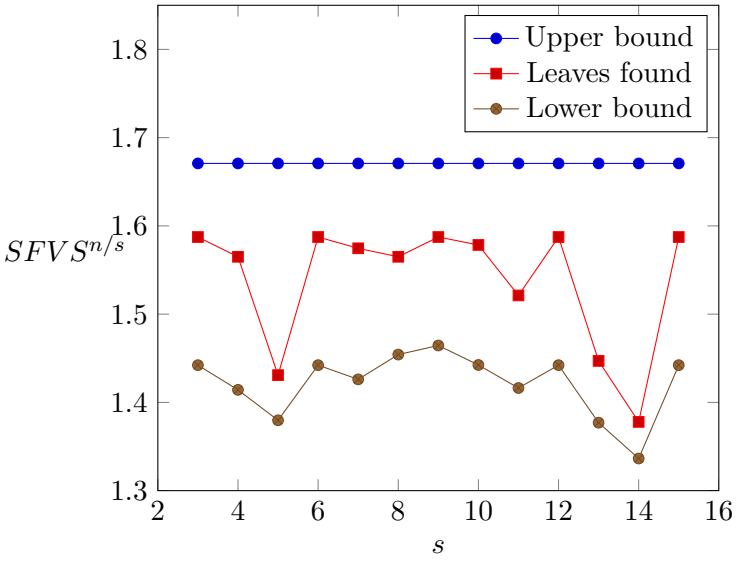


Figure 4.7: Chart showing the result given by Algorithm 4.1 as given by Table 4.3.

In Figure 4.8 we compare the number of leaves found for split graphs with the number of leaves found for chordal graphs. From Chapter 1.2.2 we know that split graphs are chordal, and in this figure we can see that split graphs are not the sub-class of chordal graphs that generates the most leaves. We chose to only compare the result of the graphs with at most 10 vertices as we ran our implementation on the same data sets for both split and chordal graphs for these sizes. In addition to these differences we see that there is a substantial gap between the number of leaves found and the number of minimal subset feedback vertex sets found in graphs of a given size. This is shown in Figure 4.7.

In this section we were able to give a better lower bound on the number of minimal subset feedback vertex sets in split graphs. The observations we were able to make when comparing the number of leaves found to the number of minimal subset feedback vertex sets found, and the comparison between the number of leaves found in chordal graphs as opposed to split graphs strongly indicate that the upper bound given by Golovach et al. in [12] for chordal graphs is too high for split graphs.

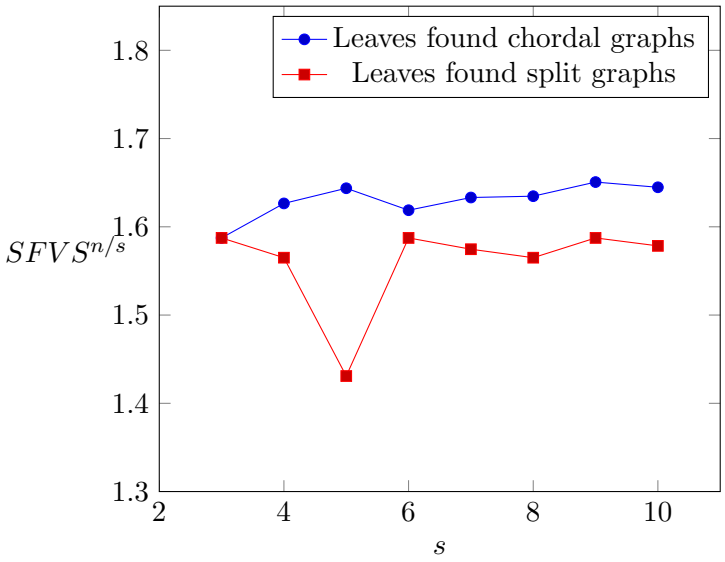


Figure 4.8: Chart comparing the number of leaves found in split graphs and chordal graphs when we test Algorithm 4.1.

Chapter 5

Conclusion

In this chapter we will first give a summary of this thesis. Then we will discuss possible open questions.

5.1 Summary

In the first chapters we gave background information on minimal dominating sets, minimal subset feedback vertex sets, chordal and split graphs. We gave algorithms for testing whether a graph is chordal and for testing whether a graph is a split graph.

The goal of the thesis was to see whether it was possible to achieve better upper or lower bounds for the number of minimal dominating sets in chordal graphs and for the number of minimal subset feedback vertex sets in chordal and split graphs. To achieve this goal we implemented two algorithms. One algorithm for enumerating all minimal dominating sets in chordal graph [2], and one algorithm for enumerating all minimal subset feedback vertex sets in chordal graphs [12]. We tested the latter algorithm on both chordal and split graphs.

In terms of upper bounds we managed to find strong indications that the upper bound is too high for both minimal dominating sets in chordal graphs and for minimal subset feedback vertex sets in split graphs. There were some indications that that the upper bound on the number of minimal subset feedback vertex sets in chordal graphs is also too high and could be equal to the lower bound. For minimal dominating sets in chordal graphs we managed to improve the practical running time of the implementation of Algorithm 3.1 given by [2]. In spite of these results we were not able to prove any better upper upper bounds than the ones given in [2] and [12].

When we studied the lower bounds on the number of minimal subset feedback vertex sets in split graphs we were able to find several graphs that achieved a better lower bound than the previously best known lower bound example. The new lower bound on the number of minimal subset feedback

vertex sets in split graphs is 1.4645^n.

For the lower bound on the number of minimal dominating sets in chordal graphs we found that the currently best known lower bound is achieved in several non-isomorphic graphs where the number of vertices in each component is divisible by 3. This supports the hypothesis that $3^{n/3}$ is in fact the upper bound for minimal dominating sets in chordal graphs.

We were not able to achieve any better results for the lower bound on the number of minimal subset feedback vertex sets in chordal graphs.

5.2 Further work

Our results concerning the number of minimal dominating sets in chordal graphs could indicate that the upper bound given by Couturier et al. in [2] is too high, though we have not been able to prove this. Therefore we pose the question:

1. Could there exist an algorithm that gives a better upper bound than 1.6181^n, on the number of minimal dominating sets in chordal graphs?

The results we were able to achieve for the number of minimal subset feedback vertex sets in split graphs indicate that there exist graphs that give a better lower bound than 1.4422^n, but the upper and lower bounds are still not equal. As such we ask:

1. Could there exist a better lower bound on the number of minimal subset feedback vertex sets in split graphs than 1.4645^n?

2. Could there exist a better upper bound on the number of minimal subset feedback vertex sets in split graphs?

Appendix A

Recognizing chordal and split graphs

A.1 MCS Algorithm

```
public static LinkedList<Integer> mcsHashSet
                (ArrayList<ArrayList<Integer>> edge,
                 int numNodes){
        LinkedList<Integer> nP = new LinkedList<Integer>();
        ArrayList<HashSet<Integer>> v=
            new ArrayList<HashSet<Integer>>();
        int[] nV = new int[numNodes];

        for(int i =0; i<numNodes; i++){
            v.add(i, new HashSet<Integer>());
        }
        for(int i =0; i<numNodes; i++){
            v.get(0).add(i);
            nV[i] =0;
        }
        int i =numNodes-1;
        int j=0;

        while(i>=0){
            int node = v.get(j).iterator().next();
            v.get(j).remove(node);
            nV[node]=-1;
            nP.addFirst(node);

            for(int a =0; a< edge.get(node).size(); a++){
                if(nV[edge.get(node).get(a)]>=0){
```

```
                    int temp = edge.get(node).get(a);
                    v.get(nV[temp]).remove(temp);
                    nV[temp]+=1;
                    v.get(nV[temp]).add(temp);
                }
            }
            i--;
            j++;
            while(j>=0 && j<numNodes && v.get(j).isEmpty()){
                j--;
            }
        }
        return nP;
    }
```

A.2 checkPEO

```
public static boolean isChordal(LinkedList<Integer> peo,
                    ArrayList<ArrayList<Integer>> edge){
        int w;
        int v;
        ArrayList<Integer> p = new ArrayList<Integer>();
        p.addAll(peo);

        int [] follower = new int[peo.size()];
        int [] index = new int[peo.size()];

        for(int i=0; i<peo.size(); i++){
            w=p.get(i);
            follower[w]=w;
            index[w] = i;
            for(int j=0; j<edge.get(w).size(); j++){
                if(p.indexOf(edge.get(w).get(j))<i){
                    v= edge.get(w).get(j);
                    index[v] = i;
                    if(follower[v] ==v){
                        follower[v]=w;
                    }
                }
            }
            for(int j=0; j<edge.get(w).size(); j++){
                if(p.indexOf(edge.get(w).get(j))<i){
                    v= edge.get(w).get(j);
```

```
                    if(index[follower[v]]<i)
                        return false;
                }
            }
        }
        return true;
    }
```

A.3 Split graph recognition.

```
public static boolean isSplit
    (ArrayList<ArrayList<Integer>> edge){
        ArrayList<Pair> unSortedDeg =
            generateDegreeSequence(edge);
        ArrayList<Pair> deg =
            sortAccordingToDegree(unSortedDeg, edge.size());

        int w = findW(deg);
        int left =0;
        int right;

        for (int i = 1; i <= w; i++) {
            left += deg.get(i-1).grad;
        }
        right = w *(w-1);
        for (int i = w+1; i <= deg.size(); i++) {
            right+=deg.get(i-1).grad;
        }

        return left==right;
    }
```

Appendix B

Implementation of MDS algorithm

```
public void finn_mds(HashMap<Integer,Integer> slettaNoder,
                     ArrayList<Integer> dSet,
                     LinkedHashSet<Integer> p){
     if(slettaNoder.size() == graf.size() ||
        isDominatingSet(dSet) ){
           numLeaves++;
           if(isMinimalDominatingSet(dSet)){
               mdsListe.add(dSet);
           }
        }
     else{
        int x = p.iterator().next();
        if(checkIfHasNoNeighbour(x, slettaNoder)){
            if(isDominated(x, dSet)){
                slettaNoder.put(x, x);
                p.remove(x);
                finn_mds(slettaNoder, dSet, p);
            }else{
                slettaNoder.put(x, x);
                p.remove(x);
                dSet.add(x);
                finn_mds(slettaNoder, dSet, p);
            }
        }
        else if(isDominated(x,dSet)){
            if(neighbourhoodIsDominated(x, dSet)){
                finn_mds(removeNode(x, slettaNoder),
                         dSet,
```

```
                removeFromPEO(x, p));
        }else {
            HashMap<Integer, Integer> slettaNoderCopy =
                new HashMap<Integer, Integer>();
            for (Map.Entry e : slettaNoder.entrySet()) {
                slettaNoderCopy.put(
                   (Integer) e.getKey(),
                   (Integer) e.getValue());
            }
            ArrayList<Integer> dSetCopy =
                new ArrayList<Integer>();
            dSetCopy.addAll(dSet);
            LinkedHashSet<Integer> pCopy =
                new LinkedHashSet<Integer>();
            Iterator e = p.iterator();
            while (e.hasNext()) {
                pCopy.add((Integer) e.next());
            }
            finn_mds(removeNeighbourhood(x, slettaNoder),
                     addToDomSet(x, dSet),
                     removeNeighbourhoodFromPEO(x, p));
            finn_mds(removeNode(x, slettaNoderCopy),
                     dSetCopy,
                     removeFromPEO(x, pCopy));
        }
    }
    else{
        HashMap<Integer,Integer> slettaNoderCopy =
            new HashMap<Integer, Integer>();
        for(Map.Entry e : slettaNoder.entrySet()){
            slettaNoderCopy.put(
                (Integer) e.getKey(),
                (Integer) e.getValue());
        }
        ArrayList<Integer> dSetCopy =
            new ArrayList<Integer>();
        dSetCopy.addAll(dSet);
        LinkedHashSet<Integer> pCopy =
            new LinkedHashSet<Integer>();
        Iterator e = p.iterator();
        while(e.hasNext()){
            pCopy.add((Integer)e.next());
        }
        int y1 = hentNabo(x,slettaNoder);
```

```
            int y2 = hentNabo(x,slettaNoderCopy);

            finn_mds(removeTwoNodes(x,y1,slettaNoder),
                     addToDomSet(y1, dSet),
                     removeTwoFromPEO(x, y1, p));
            finn_mds(removeNode(y2, slettaNoderCopy),
                     dSetCopy,
                     removeFromPEO(y2, pCopy));
        }
    }
}
```

Appendix C

Implementation of MSFVS algorithm

```
public void alg(HashMap<Integer, Integer> f,
                HashMap<Integer, Integer> sfvs,
                HashMap<Integer, Integer> r){
   LinkedList<Integer> peoCopy =
      new LinkedList<Integer>();
   HashMap<Integer, Integer> fCopy =
      new HashMap<Integer, Integer>();
   HashMap<Integer, Integer> sfvsCopy =
      new HashMap<Integer, Integer>();
   HashMap<Integer, Integer> rCopy =
      new HashMap<Integer, Integer>();
   copyMap(f, fCopy);
   copyMap(sfvs, sfvsCopy);
   copyMap(r, rCopy);
   reduce(fCopy, sfvsCopy, rCopy);
   if(fCopy.size() + sfvsCopy.size() == graf.size()){
       numberOfLeaves++;
       if(checkMaximalSForest(sfvsCopy)){
           fSets.add(fCopy);
           uSets.add(sfvsCopy);
       }
   }else {
       if(!fCopy.isEmpty()) {
           if (!isSforest(fCopy)) {
               return;
           }
       }
       peoCopy.addAll(peo);
```

```
cleanPEO(rCopy, sfvsCopy, peoCopy);
int v = peoCopy.remove();
int neigh = findNumNeighboursInG(v,sfvsCopy, rCopy);
//case 1
if (!fCopy.containsKey(v)) {
    if(!sfvsCopy.containsKey(v)) {
        //1.1
        if (S.contains(v)) {
            if (noNeighboursInF(v, fCopy,rCopy )) {
                if (neigh == 2) {
                    case11numNeigh2(fCopy, sfvsCopy,
                                    rCopy, v);
                } else if (neigh == 3) {
                    case11numNeigh3(fCopy, sfvsCopy,
                                    rCopy, v);
                } else if (neigh >= 4) {
                    case11numNeigh4(fCopy, sfvsCopy,
                                    rCopy, v);
                }
            }//1.2
            else if(!noNeighboursInF(v, fCopy,rCopy )){
                case12(fCopy, sfvsCopy, rCopy, v);
            }
        }//1.3
        else if (!S.contains(v)) {
            if (noNeighboursInF(v, fCopy,rCopy )){
                if (neigh == 2) {
                    case13numNeigh2(fCopy, sfvsCopy,
                                    rCopy, v);
                } else if (neigh == 3) {
                    case13numNeigh3(fCopy, sfvsCopy,
                                    rCopy, v);
                } else if (neigh >= 4) {
                    case13numNeigh4(fCopy, sfvsCopy,
                                    rCopy, v);
                }
            }//1.4
            else if(!noNeighboursInF(v, fCopy,rCopy )){
                int u = -1;
                for (int i = 0;
                  i < graf.get(v).size();
                  i++){
                    int t = graf.get(v).get(i);
                    if (fCopy.containsKey(t) &&
```

```
                                !rCopy.containsKey(t){
                                    u = t;
                                    break;
                                }
                            }
                            if (S.contains(u)) {
                                case14uInS(fCopy, sfvsCopy,
                                           rCopy, v, u);
                            } else {
                                case14uNotInS(fCopy, sfvsCopy,
                                              rCopy, v, u);
                            }
                        }
                    }
                } else { //case 2
                    if (S.contains(v)) {
                        //2.1
                        if (neigh == 2) {
                            case21NumNeigh2(fCopy, sfvsCopy,
                                            rCopy, v);
                        } else if (neigh >= 3) {
                            case21NumNeigh3(fCopy, sfvsCopy,
                                            rCopy, v);
                        }
                    }//2.2
                    else {
                        int numNeigbours = graf.get(v).size();
                        if (neigh > 0 &&
                            !hasNeighbourInF(v,fCopy,rCopy)){
                                case22(fCopy, sfvsCopy,
                                       rCopy, v, numNeigbours);
                        }
                    }
                }
            }
        }
```

C.1 Reduce

```
private void reduce(HashMap<Integer, Integer> f,
                    HashMap<Integer, Integer> sfvs,
                    HashMap<Integer, Integer>r){
```

```
boolean changed = true;
while(changed) {
    changed = false;
    //Rule A
    for (int i = 0; i < graf.size(); i++) {
        if(f.containsKey(i)|| sfvs.containsKey(i)){
            continue;
        }
        ArrayList<Integer> cand =
           new ArrayList<Integer>();
        for (int j = 0; j < graf.get(i).size(); j++) {
            int u = graf.get(i).get(j);
            if (f.containsKey(u) && !r.containsKey(u)){
                cand.add(u);
            }
        }
        if(cand.size()<2){
            continue;
        }
        boolean update = false;
        for (int j = 0; j < cand.size()-1; j++) {
            int u = cand.get(j);
            for (int k = j+1; k < cand.size(); k++) {
                int w = cand.get(k);
                if(f.containsKey(u) &&
                   f.containsKey(w)){
                    if(graf.get(u).contains(w)){
                        if(S.contains(u) ||
                           S.contains(w) ||
                           S.contains(i)){
                               sfvs.put(i,i);
                               update = true;
                               changed = true;
                               break;
                        }
                    }
                }
            }
            if(update){
                break;
            }
        }
    }
    if(changed){
```

```
        continue
    }

    //Rule B
    for (int i = 0; i < graf.size(); i++) {
        if(sfvs.containsKey(i) || r.containsKey(i)){
            continue;
        }
        int numN = findNumNeighboursInG(i, sfvs, r);
        //legger v til i f og r
        if(numN<2){
            if (!f.containsKey(i)){
               f.put(i,i);
            }
            r.put(i,i);
            changed = true;
            break;
        }
    }
    if(changed){
        continue;
    }
    //Rule C
    for (int i = 0; i < graf.size(); i++) {
        boolean inS = false;
        if (r.containsKey(i) || sfvs.containsKey(i))
            continue;
        if(S.contains(i)){
            continue;
        }
        for (int j = 0; j < graf.get(i).size(); j++) {
            int u = graf.get(i).get(j);
            if(S.contains(u) &&
               !sfvs.containsKey(u) &&
               !r.containsKey(u)){
                    inS = true;
                    break;
            }
        }
        if(inS){
            continue;
        }
        boolean isSimp = isSimplicial(i,r,sfvs);
        if(isSimp){
```

```
                if(!f.containsKey(i) && !sfvs.containsKey(i)){
                    f.put(i,i);
                }
                r.put(i,i);
                changed = true;
                break;
            }
        }
    }
}
```

C.2 Case 1

C.2.1 Case 1.1

```
private void case11numNeigh2(
     HashMap<Integer, Integer> fCopy,
     HashMap<Integer, Integer> sfvsCopy,
     HashMap<Integer, Integer> rCopy, int v) {
        int u1 = -1, u2 = -1;
        boolean foundFirst = false;
        for (int i = 0; i < graf.get(v).size(); i++) {
            int a = graf.get(v).get(i);
            if (foundFirst){
                if (sfvsCopy.containsKey(a) ||
                    rCopy.containsKey(a)){
                        continue;
                }
                u2 = a;
                break;
            }
            if (sfvsCopy.containsKey(a) ||
                rCopy.containsKey(a)) {
                    continue;
            }
            u1 = a;
            foundFirst = true;

        }
        assert u1!=-1:"ugyldig verdig på u1";
        assert u2!=-1:"ugyldig verdig på u2";
        HashMap<Integer, Integer> fCopy1 =
            new HashMap<Integer, Integer>();
        HashMap<Integer, Integer> sfvsCopy1 =
```

```
            new HashMap<Integer, Integer>();
        HashMap<Integer, Integer> rCopy1 =
            new HashMap<Integer, Integer>();
        copyMap(fCopy, fCopy1);
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);

        fCopy1.put(u1, u1);
        fCopy1.put(u2, u2);
        sfvsCopy1.put(v, v);
        alg(fCopy1, sfvsCopy1, rCopy1);

        copyMap(fCopy, fCopy1);
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);
        fCopy1.put(v, v);
        fCopy1.put(u2, u2);
        sfvsCopy1.put(u1, u1);
        alg(fCopy1, sfvsCopy1, rCopy1);

        copyMap(fCopy, fCopy1);
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);
        fCopy1.put(v, v);
        fCopy1.put(u1, u1);
        sfvsCopy1.put(u2, u2);
        alg(fCopy1, sfvsCopy1, rCopy1);

        copyMap(fCopy, fCopy1);
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);
        fCopy1.put(v, v);
        sfvsCopy1.put(u1, u1);
        sfvsCopy1.put(u2, u2);
        alg(fCopy1, sfvsCopy1, rCopy1);
    }

private void case11numNeigh3(
    HashMap<Integer, Integer> fCopy,
    HashMap<Integer, Integer> sfvsCopy,
    HashMap<Integer, Integer> rCopy, int v){
        int u1 = -1, u2 = -1, u3 = -1;
        boolean foundFirst = false;
        boolean foundSec = false;
```

```
for (int i = 0; i < graf.get(v).size(); i++) {
    int a = graf.get(v).get(i);
    if (foundSec) {
        if (sfvsCopy.containsKey(a) ||
            rCopy.containsKey(a)) {
                continue;
        }
        u3 = a;
        break;
    } else if (foundFirst) {
        if (sfvsCopy.containsKey(a) ||
            rCopy.containsKey(a)) {
                continue;
        }
        u2 = a;
        foundSec = true;

    } else {

        if (sfvsCopy.containsKey(a) ||
            rCopy.containsKey(a)) {
                continue;
        }
        u1 = a;
        foundFirst = true;
    }
}
HashMap<Integer, Integer> fCopy1 =
    new HashMap<Integer, Integer>();
HashMap<Integer, Integer> sfvsCopy1 =
    new HashMap<Integer, Integer>();
HashMap<Integer, Integer> rCopy1 =
    new HashMap<Integer, Integer>();
copyMap(fCopy, fCopy1);
copyMap(sfvsCopy, sfvsCopy1);
copyMap(rCopy, rCopy1);

fCopy1.put(v, v);
fCopy1.put(u1, u1);
sfvsCopy1.put(u2, u2);
sfvsCopy1.put(u3, u3);
alg(fCopy1, sfvsCopy1, rCopy1);

copyMap(fCopy, fCopy1);
```

```
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);
        fCopy1.put(v, v);
        fCopy1.put(u2, u2);
        sfvsCopy1.put(u1, u1);
        sfvsCopy1.put(u3, u3);
        alg(fCopy1, sfvsCopy1, rCopy1);

        copyMap(fCopy, fCopy1);
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);
        fCopy1.put(v, v);
        fCopy1.put(u3, u3);
        sfvsCopy1.put(u1, u1);
        sfvsCopy1.put(u2, u2);
        alg(fCopy1, sfvsCopy1, rCopy1);

        copyMap(fCopy, fCopy1);
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);
        fCopy1.put(v, v);
        sfvsCopy1.put(u1, u1);
        sfvsCopy1.put(u2, u2);
        sfvsCopy1.put(u3, u3);
        alg(fCopy1, sfvsCopy1, rCopy1);

        copyMap(fCopy, fCopy1);
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);
        fCopy1.put(u1, u1);
        sfvsCopy1.put(v, v);
        alg(fCopy1, sfvsCopy1, rCopy1);

        copyMap(fCopy, fCopy1);
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);
        fCopy1.put(u2, u2);
        fCopy1.put(u3, u3);
        sfvsCopy1.put(v, v);
        sfvsCopy1.put(u1, u1);
        alg(fCopy1, sfvsCopy1, rCopy1);
    }
private void case11numNeigh4(
    HashMap<Integer, Integer> fCopy,
```

```
HashMap<Integer, Integer> sfvsCopy,
HashMap<Integer, Integer> rCopy, int v) {
    HashMap<Integer, Integer> fCopy1 =
        new HashMap<Integer, Integer>();
    HashMap<Integer, Integer> sfvsCopy1 =
        new HashMap<Integer, Integer>();
    HashMap<Integer, Integer> rCopy1 =
        new HashMap<Integer, Integer>();
    copyMap(fCopy, fCopy1);
    copyMap(sfvsCopy, sfvsCopy1);
    copyMap(rCopy, rCopy1);

    sfvsCopy1.put(v, v);
    alg(fCopy1, sfvsCopy1, rCopy1);

    copyMap(fCopy, fCopy1);
    copyMap(sfvsCopy, sfvsCopy1);
    copyMap(rCopy, rCopy1);
    fCopy1.put(v, v);
    for (int i = 0; i < graf.get(v).size(); i++) {
        int a = graf.get(v).get(i);
        if(rCopy1.containsKey(a) ||
           sfvsCopy1.containsKey(a))
               continue;
        sfvsCopy1.put(a, a);
    }
    alg(fCopy1, sfvsCopy1, rCopy1);

    for (int i = 0; i < graf.get(v).size(); i++) {
        copyMap(fCopy, fCopy1);
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);
        int ui = graf.get(v).get(i);
        if(rCopy1.containsKey(ui) ||
           sfvsCopy1.containsKey(ui))
               continue;
        fCopy1.put(v, v);
        fCopy1.put(ui, ui);
        for (int j = 0; j < graf.get(v).size(); j++) {
            int a = graf.get(v).get(j);
            if (a == ui) {
                continue;
            }
            if(rCopy1.containsKey(a) ||
```

```
                    sfvsCopy1.containsKey(a))
                        continue;
                sfvsCopy1.put(a, a);
            }
            alg(fCopy1, sfvsCopy1, rCopy1);
        }
    }
```

C.2.2 Case 1.2

```
private void case12(
    HashMap<Integer, Integer> fCopy,
    HashMap<Integer, Integer> sfvsCopy,
    HashMap<Integer, Integer> rCopy, int v) {
        int u1 = -1;
        //finner den ene naboen som må være i F
        for (int i = 0; i < graf.get(v).size(); i++) {
            if (fCopy.containsKey(graf.get(v).get(i)) &&
                !rCopy.containsKey(graf.get(v).get(i))) {
                    u1 = graf.get(v).get(i);
                    break;
            }
        }
        HashMap<Integer, Integer> fCopy1 =
            new HashMap<Integer, Integer>();
        HashMap<Integer, Integer> sfvsCopy1 =
            new HashMap<Integer, Integer>();
        HashMap<Integer, Integer> rCopy1 =
            new HashMap<Integer, Integer>();
        ArrayList<Integer> naboer = new ArrayList<Integer>();
        finnNaboer(sfvsCopy, rCopy, v, naboer); //O(n)
        copyMap(fCopy, fCopy1);
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);

        sfvsCopy1.put(v, v);
        alg(fCopy1, sfvsCopy1, rCopy1);

        copyMap(fCopy, fCopy1);
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);
        fCopy1.put(v, v);
        for (Integer aNaboer : naboer) {
            if (aNaboer == u1 || aNaboer == v) {
```

```
                continue;
            }
            sfvsCopy1.put(aNaboer, aNaboer);
        }
        alg(fCopy1, sfvsCopy1, rCopy1);
    }
```

C.2.3 Case 1.3

```
private void case13numNeigh2(
    HashMap<Integer, Integer> fCopy,
    HashMap<Integer, Integer> sfvsCopy,
    HashMap<Integer, Integer> rCopy, int v) {
        int u1 = -1,u2 = -1;
        boolean foundFirst = false;
        for (int i = 0; i < graf.get(v).size(); i++) {
            int a = graf.get(v).get(i);
            if (foundFirst){
                if (sfvsCopy.containsKey(a) ||
                    rCopy.containsKey(a)){
                       continue;
                }
                u2 = a;
                break;
            }
            if (sfvsCopy.containsKey(a) ||
                rCopy.containsKey(a)) {
                   continue;
            }
            u1 = a;
            foundFirst = true;

        }
        assert u1!=-1: "u1 har ugyldig verdi";
        assert u2 != -1: "u2 har ugyldig verdi";
        HashMap<Integer, Integer> fCopy1 =
            new HashMap<Integer, Integer>();
        HashMap<Integer, Integer> sfvsCopy1 =
            new HashMap<Integer, Integer>();
        HashMap<Integer, Integer> rCopy1 =
            new HashMap<Integer, Integer>();
        copyMap(fCopy, fCopy1);
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);
```

```
        fCopy1.put(u1, u1);
        fCopy1.put(u2, u2);
        sfvsCopy1.put(v, v);
        alg(fCopy1, sfvsCopy1, rCopy1);

        copyMap(fCopy, fCopy1);
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);
        fCopy1.put(v, v);
        fCopy1.put(u2, u2);
        sfvsCopy1.put(u1, u1);
        alg(fCopy1, sfvsCopy1, rCopy1);

        copyMap(fCopy, fCopy1);
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);
        fCopy1.put(v, v);
        fCopy1.put(u1, u1);
        sfvsCopy1.put(u2, u2);
        alg(fCopy1, sfvsCopy1, rCopy1);

        copyMap(fCopy, fCopy1);
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);
        fCopy1.put(v, v);
        sfvsCopy1.put(u1, u1);
        sfvsCopy1.put(u2, u2);
        alg(fCopy1, sfvsCopy1, rCopy1);
    }
private void case13numNeigh3(
    HashMap<Integer, Integer> fCopy,
    HashMap<Integer, Integer> sfvsCopy,
    HashMap<Integer, Integer> rCopy, int v) {
        int u1 = -1, u2 = -1, u3 = -1;
        boolean foundFirst = false;
        boolean foundSec = false;
        for (int i = 0; i < graf.get(v).size(); i++) {
            int a = graf.get(v).get(i);
            if (foundSec) {
                if (sfvsCopy.containsKey(a) ||
                    rCopy.containsKey(a)) {
                       continue;
                }
```

```
            u3 = a;
            break;
        } else if (foundFirst) {
            if (sfvsCopy.containsKey(a) ||
                rCopy.containsKey(a)) {
                   continue;
            }
            u2 = a;
            foundSec = true;

        } else {

            if (sfvsCopy.containsKey(a) ||
                rCopy.containsKey(a)) {
                   continue;
            }
            u1 = a;
            foundFirst = true;
        }
    }
    if(S.contains(u1)){
        assert S.contains(u1) : "Does not contain u1";
    }
    else if (S.contains(u2)) {
        int temp = u1;
        u1 = u2;
        u2 = temp;
    } else if (S.contains(u3)) {
        int temp = u1;
        u1 = u3;
        u3 = temp;
    }


    HashMap<Integer, Integer> fCopy1 =
        new HashMap<Integer, Integer>();
    HashMap<Integer, Integer> sfvsCopy1 =
        new HashMap<Integer, Integer>();
    HashMap<Integer, Integer> rCopy1 =
        new HashMap<Integer, Integer>();
    copyMap(fCopy, fCopy1);
    copyMap(sfvsCopy, sfvsCopy1);
    copyMap(rCopy, rCopy1);
```

```
fCopy1.put(u1, u1);
fCopy1.put(v, v);
sfvsCopy1.put(u2, u2);
sfvsCopy1.put(u3, u3);
alg(fCopy1, sfvsCopy1, rCopy1);

copyMap(fCopy, fCopy1);
copyMap(sfvsCopy, sfvsCopy1);
copyMap(rCopy, rCopy1);
fCopy1.put(u1, u1);
fCopy1.put(u2, u2);
sfvsCopy1.put(v, v);
sfvsCopy1.put(u3, u3);
alg(fCopy1, sfvsCopy1, rCopy1);

copyMap(fCopy, fCopy1);
copyMap(sfvsCopy, sfvsCopy1);
copyMap(rCopy, rCopy1);
fCopy1.put(u1, u1);
fCopy1.put(u3, u3);
sfvsCopy1.put(v, v);
sfvsCopy1.put(u2, u2);
alg(fCopy1, sfvsCopy1, rCopy1);

copyMap(fCopy, fCopy1);
copyMap(sfvsCopy, sfvsCopy1);
copyMap(rCopy, rCopy1);
fCopy1.put(u1, u1);
sfvsCopy1.put(v, v);
sfvsCopy1.put(u2, u2);
sfvsCopy1.put(u3, u3);
alg(fCopy1, sfvsCopy1, rCopy1);

copyMap(fCopy, fCopy1);
copyMap(sfvsCopy, sfvsCopy1);
copyMap(rCopy, rCopy1);
fCopy1.put(v, v);
sfvsCopy1.put(u1, u1);
alg(fCopy1, sfvsCopy1, rCopy1);

copyMap(fCopy, fCopy1);
copyMap(sfvsCopy, sfvsCopy1);
copyMap(rCopy, rCopy1);
```

```
        fCopy1.put(u2, u2);
        fCopy1.put(u3, u3);
        sfvsCopy1.put(u1, u1);
        sfvsCopy1.put(v, v);
        alg(fCopy1, sfvsCopy1, rCopy1);
    }
private void case13numNeigh4(
    HashMap<Integer, Integer> fCopy,
    HashMap<Integer, Integer> sfvsCopy,
    HashMap<Integer, Integer> rCopy, int v) {
        int u1 = -1;
        for (int i = 0; i < graf.get(v).size(); i++) {
            int temp = graf.get(v).get(i);
            if (S.contains(temp) &&
                !rCopy.containsKey(temp) &&
                !sfvsCopy.containsKey(temp)) {
                    u1 = temp;
                    break;
            }
        }
        ArrayList<Integer> naboer = new ArrayList<Integer>();
        finnNaboer(sfvsCopy, rCopy, v, naboer);
        HashMap<Integer, Integer> fCopy1 =
           new HashMap<Integer, Integer>();
        HashMap<Integer, Integer> sfvsCopy1 =
           new HashMap<Integer, Integer>();
        HashMap<Integer, Integer> rCopy1 =
           new HashMap<Integer, Integer>();
        copyMap(fCopy, fCopy1);
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);

        sfvsCopy1.put(u1, u1);
        alg(fCopy1, sfvsCopy1, rCopy1);

        copyMap(fCopy, fCopy1);
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);
        fCopy1.put(u1, u1);
        sfvsCopy1.put(v, v);
        for (Integer aNaboer : naboer) {
            int temp = aNaboer;
            if (temp == u1 || temp == v) {
                continue;
```

```
            }

            sfvsCopy1.put(temp, temp);
        }
        alg(fCopy1, sfvsCopy1, rCopy1);

        copyMap(fCopy, fCopy1);
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);
        fCopy1.put(v, v);
        fCopy1.put(u1, u1);
        for (Integer aNaboer : naboer) {
            int temp = aNaboer;
            if (temp == u1 || temp == v) {
                continue;
            }
            sfvsCopy1.put(temp, temp);
        }
        alg(fCopy1, sfvsCopy1, rCopy1);


        for (int i = 0; i < naboer.size(); i++) {
            copyMap(fCopy, fCopy1);
            copyMap(sfvsCopy, sfvsCopy1);
            copyMap(rCopy, rCopy1);
            int temp = naboer.get(i);
            if (temp == u1 || temp == v) {
                continue;
            }
            fCopy1.put(u1, u1);
            fCopy1.put(temp, temp);
            sfvsCopy1.put(v, v);
            for (Integer aNaboer : naboer) {
                int a = aNaboer;
                if (a == u1 || a == temp || a == v) {
                    continue;
                }
                sfvsCopy1.put(a, a);
            }
            alg(fCopy1, sfvsCopy1, rCopy1);

        }
    }
```

C.2.4 Case 1.4

```
private void case14uInS(
    HashMap<Integer, Integer> fCopy,
    HashMap<Integer, Integer> sfvsCopy,
    HashMap<Integer, Integer> rCopy,
    int v,
    int u){
        HashMap<Integer, Integer> fCopy1 =
            new HashMap<Integer, Integer>();
        HashMap<Integer, Integer> sfvsCopy1 =
            new HashMap<Integer, Integer>();
        HashMap<Integer, Integer> rCopy1 =
            new HashMap<Integer, Integer>();
        copyMap(fCopy, fCopy1);
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);
        ArrayList<Integer> naboer = new ArrayList<Integer>();
        finnNaboer(sfvsCopy, rCopy, v, naboer);

        for (int i = 0; i < naboer.size(); i++) {
            copyMap(fCopy, fCopy1);
            copyMap(sfvsCopy, sfvsCopy1);
            copyMap(rCopy, rCopy1);
            int a = naboer.get(i);

            if (a==u){
                continue;
            }

            fCopy1.put(a, a);

            for (Integer aNaboer : naboer) {
                int b = aNaboer;
                if (b == u || b == a) {
                    continue;
                }

                sfvsCopy1.put(b, b);
            }
            alg(fCopy1, sfvsCopy1, rCopy1);
        }
    }
private void case14uNotInS(
```

```
    HashMap<Integer, Integer> fCopy,
    HashMap<Integer, Integer> sfvsCopy,
    HashMap<Integer, Integer> rCopy,
    int v,
    int u) {
        int w = -1;
        for (int i = 0; i < graf.get(v).size(); i++) {
            int temp = graf.get(v).get(i);
            if (S.contains(temp) &&
                !sfvsCopy.containsKey(temp) &&
                !rCopy.containsKey(temp)) {
                    w = temp;
                    break;
            }
        }
        HashMap<Integer, Integer> fCopy1 =
            new HashMap<Integer, Integer>();
        HashMap<Integer, Integer> sfvsCopy1 =
            new HashMap<Integer, Integer>();
        HashMap<Integer, Integer> rCopy1 =
            new HashMap<Integer, Integer>();
        copyMap(fCopy, fCopy1);
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);
        ArrayList<Integer> naboer = new ArrayList<Integer>();
        finnNaboer(sfvsCopy, rCopy, v, naboer);

        fCopy1.put(w, w);
        sfvsCopy1.put(v, v);
        for (Integer aNaboer : naboer) {
            int a = aNaboer;
            if (a == u || a == w || a == v) {
                continue;
            }
            sfvsCopy1.put(a, a);
        }
        alg(fCopy1, sfvsCopy1, rCopy1);

        copyMap(fCopy, fCopy1);
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);
        sfvsCopy1.put(w, w);
        alg(fCopy1, sfvsCopy1, rCopy1);
    }
```

C.3 Case 2

C.3.1 Case 2.1

```
private void case21NumNeigh2(
    HashMap<Integer, Integer> fCopy,
    HashMap<Integer, Integer> sfvsCopy,
    HashMap<Integer, Integer> rCopy,
    int v) {
        int u = -1, w =-1;

        boolean foundFirst = false;
        for (int i = 0; i < graf.get(v).size(); i++) {
            int a = graf.get(v).get(i);
            if (foundFirst){
                if (sfvsCopy.containsKey(a) ||
                    rCopy.containsKey(a)){
                       continue;
                }
                w = a;
                break;
            }
            if (sfvsCopy.containsKey(a) ||
                rCopy.containsKey(a)) {
                   continue;
            }
            u = a;
            foundFirst = true;

        }
        HashMap<Integer, Integer> fCopy1 =
            new HashMap<Integer, Integer>();
        HashMap<Integer, Integer> sfvsCopy1 =
            new HashMap<Integer, Integer>();
        HashMap<Integer, Integer> rCopy1 =
            new HashMap<Integer, Integer>();
        copyMap(fCopy, fCopy1);
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);

        fCopy1.put(u, u);
        sfvsCopy1.put(w, w);
        alg(fCopy1, sfvsCopy1, rCopy1);
```

```
        copyMap(fCopy, fCopy1);
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);
        sfvsCopy1.put(u, u);
        alg(fCopy1, sfvsCopy1, rCopy1);
    }
private void case21NumNeigh3(
    HashMap<Integer, Integer> fCopy,
    HashMap<Integer, Integer> sfvsCopy,
    HashMap<Integer, Integer> rCopy,
    int v) {
        HashMap<Integer, Integer> fCopy1 =
            new HashMap<Integer, Integer>();
        HashMap<Integer, Integer> sfvsCopy1 =
            new HashMap<Integer, Integer>();
        HashMap<Integer, Integer> rCopy1 =
            new HashMap<Integer, Integer>();
        copyMap(fCopy, fCopy1);
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);
        ArrayList<Integer> naboer = new ArrayList<Integer>();
        finnNaboer(sfvsCopy, rCopy, v, naboer);

        for (int i = 0; i < naboer.size(); i++) {
            int x = naboer.get(i);
            if(x==v)
                continue;
            fCopy1.put(x, x);
            for (Integer aNaboer : naboer) {
                int a = aNaboer;
                if (a == x || a== v) {
                    continue;
                }
                sfvsCopy1.put(a, a);
            }
            alg(fCopy1, sfvsCopy1, rCopy1);
            copyMap(fCopy, fCopy1);
            copyMap(sfvsCopy, sfvsCopy1);
            copyMap(rCopy, rCopy1);
        }

        for (Integer aNaboer : naboer) {
            int a = aNaboer;
            if (a== v){
```

```
                continue;
            }
            sfvsCopy1.put(a, a);
        }
        alg(fCopy1, sfvsCopy1, rCopy1);
    }
```

C.3.2 Case 2.2

```
private void case22(
    HashMap<Integer, Integer> fCopy,
    HashMap<Integer, Integer> sfvsCopy,
    HashMap<Integer, Integer> rCopy,
    int v,
    int numNeigbours) {
        int u = -1;
        for (int i = 0; i < numNeigbours; i++) {
            int temp = graf.get(v).get(i);
            if (S.contains(temp) &&
                !fCopy.containsKey(temp) &&
                !sfvsCopy.containsKey(temp)) {
                    u = temp;
                    break;
            }

        }
        ArrayList<Integer> naboer = new ArrayList<Integer>();
        finnNaboer(sfvsCopy, rCopy, v, naboer);
        HashMap<Integer, Integer> fCopy1 =
            new HashMap<Integer, Integer>();
        HashMap<Integer, Integer> sfvsCopy1 =
            new HashMap<Integer, Integer>();
        HashMap<Integer, Integer> rCopy1 =
            new HashMap<Integer, Integer>();
        copyMap(fCopy, fCopy1);
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);

        sfvsCopy1.put(u, u);
        alg(fCopy1, sfvsCopy1, rCopy1);

        copyMap(fCopy, fCopy1);
        copyMap(sfvsCopy, sfvsCopy1);
        copyMap(rCopy, rCopy1);
```

```
        fCopy1.put(u, u);
        for (Integer aNaboer : naboer) {
            int temp = aNaboer;
            if (temp == u || temp == v) {
                continue;
            }
            sfvsCopy1.put(temp, temp);
        }
        alg(fCopy1, sfvsCopy1, rCopy1);
    }
```

Bibliography

[1] Andreas Brandstädt, Jeremy P Spinrad, et al. *Graph classes: a survey*, volume 3. Siam, 1999.

[2] 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(0):82 – 94, 2013.

[3] Jean-François Couturier, Pinar Heggernes, Pim van 't Hof, and Yngve Villanger. Maximum number of minimal feedback vertex sets in chordal graphs and cographs. In Joachim Gudmundsson, Julián Mestre, and Taso Viglas, editors, *Computing and Combinatorics*, volume 7434 of *Lecture Notes in Computer Science*, pages 133–144. Springer Berlin Heidelberg, 2012.

[4] Gabriel Andrew Dirac. On rigid circuit graphs. In *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, volume 25, pages 71–76. Springer, 1961.

[5] Guy Even, Joseph (. Naor, Baruch Schieber, and Leonid Zosin. Approximating minimum subset feedback sets in undirected graphs with applications. *SIAM Journal on Discrete Mathematics*, 13(2):255–13, 2000. Copyright - Copyright] © 2000 Society for Industrial and Applied Mathematics; Last updated - 2012-03-02.

[6] Guy Even, Joseph (. Naor, and Leonid Zosin. An 8-approximation algorithm for the subset feedback vertex set problem. *SIAM Journal on Computing*, 30(4):1231–22, 2000. Copyright - Copyright] © 2000 Society for Industrial and Applied Mathematics; Last updated - 2012-06-30.

[7] Fedor V. Fomin, Fabrizio Grandoni, Artem V. Pyatkin, and Alexey A. Stepanov. Combinatorial bounds via measure and conquer: Bounding minimal dominating sets and applications. *ACM Trans. Algorithms*, 5(1):9:1–9:17, December 2008.

[8] Fedor V. Fomin and Dieter Kratsch. *Exact Exponential Algorithms*. Springer, 2010.

[9] FedorV. Fomin, Pinar Heggernes, Dieter Kratsch, Charis Papadopoulos, and Yngve Villanger. Enumerating minimal subset feedback vertex sets. In Frank Dehne, John Iacono, and Jörg-Rüdiger Sack, editors, *Algorithms and Data Structures*, volume 6844 of *Lecture Notes in Computer Science*, pages 399–410. Springer Berlin Heidelberg, 2011.

[10] Delbert Fulkerson and Oliver Gross. Incidence matrices and interval graphs. *Pacific journal of mathematics*, 15(3):835–855, 1965.

[11] F. Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM Journal on Computing*, 1(2):180–187, 1972.

[12] Petr A. Golovach, Pinar Heggernes, Dieter Kratsch, and Reza Saei. Subset feedback vertex sets in chordal graphs. *Journal of Discrete Algorithms*, 26(0):7 – 15, 2014.

[13] Martin Charles Golumbic. *Algorithmic graph theory and perfect graphs*, volume 57. Elsevier, 2004.

[14] Teresa W Haynes, Stephen Hedetniemi, and Peter Slater. *Fundamentals of domination in graphs*. CRC Press, 1998.

[15] Pinar Heggernes. Treewidth, partial k-trees, and chordal graphs. Delpensum INF334, 2006.

[16] Yoichi Iwata. A faster algorithm for dominating set analyzed by the potential method. In Dániel Marx and Peter Rossmanith, editors, *Parameterized and Exact Computation*, volume 7112 of *Lecture Notes in Computer Science*, pages 41–54. Springer Berlin Heidelberg, 2012.

[17] E.L. Lawler. A note on the complexity of the chromatic number problem. *Information Processing Letters*, 5(3):66 – 67, 1976.

[18] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, {II}. *Journal of Symbolic Computation*, 60(0):94 – 112, 2014.

[19] R Garey Michael and S Johnson David. Computers and intractability: a guide to the theory of np-completeness. *WH Freeman & Co., San Francisco*, 1979.

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

[21] Donald J Rose. Triangulated graphs and the elimination process. *Journal of Mathematical Analysis and Applications*, 32(3):597 – 609, 1970.

[22] Charles Semple and Mike Steel. Phylogenetics. 2003.

[23] R. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13(3):566–579, 1984.