Inclusion/Exclusion Meets Measure and Conquer: Exact algorithms for counting dominating sets

Johan M. M. van Rooij

Jesper Nederlof

Thomas C. van Dijk

Technical Report UU-CS-2008-043
November 2008

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Inclusion/Exclusion Meets Measure and Conquer: Exact algorithms for counting dominating sets

Johan M. M. van Rooij Jesper Nederlof Thomas C. van Dijk

Abstract

We look at the principle of inclusion/exclusion from a branching perspective. More specifically, we combine traditional branching with inclusion/exclusion based branching and analyse such algorithms by means of measure and conquer. This branching is combined with path decomposition techniques on sparse instances, and some reduction rules.

We consider the standard set cover formulation of dominating set and present an algorithm that counts the number of dominating sets of each cardinality in $\mathcal{O}(1.5048^n)$ time. This algorithm computes much more information than the previous fastest decision algorithm for minimum dominating set in slightly less time. For counting the number of minimum dominating sets, our algorithm significantly improves previous results.

When restricted to c-dense graphs, circle graphs, 4-chordal graphs or weakly chordal graphs, our combination of branching with inclusion/exclusion leads to significantly faster counting and decision algorithms than the previously fastest algorithms for dominating set.

All results can be extended to counting (minimum) weighted dominating sets when the size of the set of possible weight sums is polynomially bounded.

1 Introduction

The field of exact exponential time algorithms has been an area of growing interest over the last few years. Many techniques have been developed or rediscovered, and various surveys on the field have been written [12, 18, 24, 26].

Most notable among these new or rediscovered techniques are *measure and conquer* [11, 13] and *inclusion/exclusion* [2, 6, 19]. Both techniques have been demonstrated on the Set Cover problem in early stages: measure and conquer was introduced on a set cover formulation of Minimum Dominating Set, and in [6] inclusion/exclusion was used for counting set coverings and set partitionings.

The best known shape of inclusion/exclusion is a sum over some powerset (for examples, see [6, 4, 19]). However, the fundamental branching perspective from [2] is more direct and powerful. In this paper, we will apply this branching perspective to set cover instances obtained from the set cover formulation of dominating set that has been used to introduce measure and conquer [11].

In this setting, we use a traditional branching rule to branch on a set, or an application of inclusion/exclusion to branch on an element. The sole application of either one of these strategies gives a typical exhaustive search or the aforementioned shape of inclusion/exclusion sum, respectively. We use both branching strategies in unity obtaining a *mixed branch and reduce* algorithm that can be analysed using measure and conquer.

Until 2004, no exact algorithm for Minimum Dominating Set beating the trivial $\mathcal{O}(2^n n^{\mathcal{O}(1)})$ was known. In that year, three algorithms were published: Fomin et al. obtained an $\mathcal{O}(1.9379^n)$ time algorithm [15], Randerath and Schiermeyer an $\mathcal{O}(1.8999^n)$ time algorithm [22], and Grandoni an $\mathcal{O}(1.8019^n)$ time algorithm [17]. One year later, the algorithm of Grandoni was analysed using measure and conquer giving a bound of $\mathcal{O}(1.5137^n)$ on the running time [11]. This was later improved by Van Rooij and Bodlaender [25] to $\mathcal{O}(1.5063^n)$.

When generalised to *counting minimum dominating sets*, there is an algorithm by Fomin et al. running in time $\mathcal{O}(1.5535^n)$ [10]. This algorithm combines traditional branching with dynamic

programming over path decompositions: an approach we will follow for our own algorithm as well. Related to this is a result by Björklund and Husfeldt showing that the number of minimum dominating sets in a cubic graph can be counted in $\mathcal{O}(1.3161^n)$ [4] using inclusion/exclusion in combination with dynamic programming over path decompositions. Although these combinations are known, there are, to our knowledge, no existing algorithms combining measure and conquer with inclusion/exclusion.

Our algorithm is even more general. It counts the number of dominating sets in an n-vertex graph of each size $0 \leq \kappa \leq n$, with an upper bound on the running time of $\mathcal{O}(1.5048^n)$. This is slightly faster than even the current fastest algorithm that computes a minimum dominating set. Thus, we also obtain the currently fastest algorithm for computing a minimum dominating set.

Gaspers et al. [16] show that algorithms for the set cover formulation of dominating set can be combined with dynamic programming over tree decompositions to obtain faster running times for the dominating set problem restricted to some graph classes. These classes are c-dense graphs, chordal graphs, circle graphs, 4-chordal graphs and weakly chordal graphs. We show that our mixed branching approach with inclusion/exclusion branches works even better on four of these graph classes; we not only improve these results because we have a faster algorithm for the underlying set cover problem, but more significantly improve these results by exploiting vertices of high degree twice by using both techniques. Moreover, we also count the number of dominating sets of each size, in contrast to the previous results that compute a single minimum dominating set.

Our paper is organised in the following way. We begin by introducing the problems and terminology in Section 2. Thereafter, we will discuss how to use inclusion/exclusion to branch in Section 3. Then a description of the algorithm and its reduction rules is presented in Section 4, followed by a measure and conquer analysis in Section 5, and an analysis of the dynamic programming over path decompositions in Section 6. We conclude with the consideration of the special graph classes in Section 7.

2 Preliminaries

We consider the counting variant of the Minimum Dominating Set problem on an n-vertex graph $G = (V, E)$.

#Minimum Dominating Set
Instance: A graph $G = (V, E)$.
Question: How many minimum dominating sets exist for G, i.e., how many subsets $V' \subseteq V$ with $|V'|$ minimal such that for all $u \in V \backslash V'$ there is a $v \in V'$ for which $(u, v) \in E$?

We solve the #Minimum Dominating Set problem by considering dominating sets of all sizes and solve #κ-Dominating Set for all $0 \leq \kappa \leq n$.

#κ-Dominating Set
Instance: A graph $G = (V, E)$ and a positive integer κ.
Question: How many dominating sets of size κ exist for G, i.e., how many subsets $V' \subseteq V$ with $|V'| = \kappa$ such that for all $u \in V \backslash V'$ there is a $v \in V'$ for which $(u, v) \in E$?

We use different perspectives on this problem. These will give us additional insight into the structure of the problem. We will often switch between these different perspectives throughout the presentation of our algorithm.

As is common in contemporary work on dominating set algorithms, we formulate the problem as a Set Cover problem [17]. In our case, this means formulating #κ-Dominating Set as #κ-Set Cover.

#κ-Set Cover

Instance: A collection $\mathcal{S}$ of subsets of a finite universe $\mathcal{U}$ and a positive integer κ.

Question: How many set covers for $\mathcal{U}$ of size κ does $\mathcal{S}$ contain, i.e., how many subsets $\mathcal{S}' \subseteq \mathcal{S}$ with $|\mathcal{S}'| = \kappa$ such that every element of $\mathcal{U}$ belongs to at least one member of $\mathcal{S}'$.

Transforming #κ-Dominating Set to #κ-Set Cover is straightforward: for every vertex in the dominating set instance introduce both an element in $\mathcal{U}$ ('every vertex has to be dominated') and a set in $\mathcal{S}$ containing the elements corresponding to the vertices in its closed neighbourhood ('a vertex dominates itself and all its neighbours'). We often speak of a set cover instance $\mathcal{S}$ over the universe $\mathcal{U}$ without specifying $\mathcal{U}$, in that case, it is defined implicitly by $\mathcal{S}$ through $\mathcal{U} = \cup\mathcal{S}$. Using this perspective, we do not speak of the degree of a vertex but of the *cardinality* $|S|$ of a set S and the *frequency* of an element e.

In this paper, however, we deviate from this standard formulation by considering $\mathcal{S}$ to be a multiset of sets. Our multiset notation is derived directly from standard set notation. We use $S^m \in \mathcal{S}$ if we want to stress that the multiplicity of S in $\mathcal{S}$ is m, and use $\text{set}(\mathcal{S})$ for the underlying set of $\mathcal{S}$. When quantifying over $\mathcal{S}$, we will always consider a set $S \in \mathcal{S}$ its multiplicity number of times, thus $|\{S \in \mathcal{S}\}| = |\mathcal{S}|$. Mostly, we want reason about the underlying set of $\mathcal{S}$, but we will also need the multiplicities. Not to confuse both, we define the frequency of an element $\text{freq}(e)$ to be the number of distinct sets in which an element e occurs. In addition, we use $\#(e)$ if we want to include set multiplicities; this represents the total number of sets in which an element e occurs. Similarly, we let $|\mathcal{S}|$ be the number of distinct sets in $\mathcal{S}$, ignoring set multiplicities, while we let $\#(\mathcal{S})$ be the total number of sets in $\mathcal{S}$, respecting multiplicities.

Furthermore, let $\mathcal{S}[e]$ be the collection of sets in $\mathcal{S}$ containing the element e, and let $\mathcal{U}[\mathcal{S}']$ be the subset of the universe $\mathcal{U}$ with elements from $\mathcal{S}' \subseteq \mathcal{S}$ ($\mathcal{U}[\mathcal{S}'] = \cup\mathcal{S}'$). Finally, we use $\mathcal{S}' \subset \mathcal{S}$ if $\mathcal{S}' \subseteq \mathcal{S}$ and $\mathcal{S}' \neq \mathcal{S}$, and we use $\{\emptyset^0\} = \emptyset$ while $\{\emptyset^1\} = \{\emptyset\}$.

In order to express the size of a set cover instance, the *dimension* of a set cover instance is defined as $\dim(\mathcal{S},\mathcal{U}) = |\mathcal{S}| + |\mathcal{U}|$. Hence a dominating set instance on an n-vertex graph is transformed to a set cover instance of dimension $d \leq 2n$.

Following [10], the third (and final) way we look at the problem allows us to use strong techniques from graph theory on set cover instances.

Definition (Incidence graph) Given a set cover instance $\mathcal{S}$ over the universe $\mathcal{U}$, the *incidence graph* $G_{\mathcal{S}}$ of $\mathcal{S}$ is the bipartite graph with *red vertices* $V_{Red} = \mathcal{S}$ and *blue vertices* $V_{Blue} = \mathcal{U}$. Vertices $S \in V_{Red}$ and $u \in V_{Blue}$ are adjacent if and only if $u \in S$.

Now consider a solution to a Set Cover instance. This corresponds to a subset V' of the red vertices V_{Red} of the incidence graph such that all blue vertices are dominated (adjacent to a red vertex in V'). We call such a set a red/blue dominating set.

#κ-Red/Blue Dominating Set

Instance: A graph $G = (V_{Red} \cup V_{Blue}, E)$ and a positive integer κ

Question: How many red/blue dominating sets of size κ exist for G, i.e., how many subsets $V' \subseteq V_{Red}$ with $|V'| \leq \kappa$ such that for all $u \in V_{Blue}$ there is a $v \in V'$ for which $(u, v) \in E$.

Observe that #κ-Set Cover is equal to #κ-Red/Blue Dominating Set on the corresponding incidence graph.

Having introduced the problem, we need some additional notation. Let $V' \subseteq V$ be a subset of the vertices of G; we denote the subgraph induced by V' by $G[V']$. Furthermore, we denote the maximum degree of a graph G by $\Delta(G)$.

Let n_1 and n_2 be lists of numbers of equal length l. We define $n_1 + n_2$ and $n_1 - n_2$ by piecewise addition and subtraction. We denote the numbers in the list n_1 by $[n_1]_0$ up to $[n_1]_{l-1}$. Furthermore, we add an element e to the front or back of a list by using the notations $(e;\ n_1)$ and $(n_1;\ e)$, respectively. When using this notation, we write $(n_1;\ e^m)$ when adding the element e to the back of the list m times.

3 Inclusion/Exclusion Based Branching

We will begin by showing that one can look at Inclusion/Exclusion from a branching perspective, see also [3]. In this way, we can Inclusion/Exclusion branch on an element in a Set Cover instance in the same way as one would normally branch on a set.

The canonical branching rule for Set Cover is branching on a set. Sets are optional in a solution: either a set is in the solution or it not. In both branches, the problem is simplified. If we *discard* the set, we decrease the number of sets. If we *take* the set, we decrease the number of sets and in addition, this set covers all its elements and those elements can therefore be removed from the instance, decreasing the number of elements as well. The minimum set cover for the instance is either the one returned by the discard branch or the one returned by the take branch with the branch set added to it.

The counting problem can also be handled by branching steps of this type because the total number of solutions is the sum of both branches. We can do this because sets are *optional* in a solution. The branch on a set can be denoted as adding the number of solutions where it is *required* to take the set to the number of solution where it is *forbidden* to take the set:

$$\text{OPTIONAL} = \text{REQUIRED} + \text{FORBIDDEN}$$

If we are counting κ-set covers and we branch to take a set (that is, in the 'required' branch), then we should count $(\kappa - 1)$-set covers in that branch. In the 'forbidden' branch, we do not decrease κ.

We now consider branching on an element [3]. Such a branching step is unusual, and may appear strange at first sight, as elements are not optional. Inspired by Inclusion/Exclusion techniques and because we count the number of solutions, we can, however, rearrange the above formula to give:

$$\text{REQUIRED} = \text{OPTIONAL} - \text{FORBIDDEN}$$

That is, the number of ways to cover a certain element is equal to the number of ways to optionally cover it, minus the number of ways to not cover it. This is interesting because this branching rule also simplifies the instance in both branches. If we choose to make it *optional* to cover a certain element, we can remove that element from every set it occurs in, reducing the size of sets. If we choose the element *forbidden*, then we have to remove every set in which the element occurs, which is an even greater reduction in size. We have not selected a set to be in the cover in both branches, so in both branches we are looking for κ-set covers.

Consider a branching algorithm without reduction rules and without employing branch-and-bound. If the branching rule is based on an optional property of the problem, as is typically the case, the algorithm is an *exhaustive search*. A similar concept exists for an algorithm in which branching is based on a required property, which we call *inclusion/exclusion based branching* or simply *IE-branching*: without reduction rules, this is an inclusion/exclusion algorithm.

To see this, let c'_κ be the number of set covers of cardinality κ, and let $a(X)$ be the number of sets in $\mathcal{S}$ that do not include any element of X. Consider the branching tree after exhaustively applying IE-branchings. In each subproblem in this tree, each element is either optional, or forbidden. We look at the contribution of a leaf to the total number computed when X is the set of forbidden elements in this leaf. Notice that the $2^{|\mathcal{U}|}$ leaves represent the subsets $X \subseteq \mathcal{U}$. A minus sign is added for each time we have entered a forbidden branch, so the contribution of this branch will be $(-1)^{|X|}$ times $\binom{a(X)}{\kappa}$. This last number equals the number of set covers of cardinality κ where it is optional to cover each element not in X and forbidden to cover an element in X. All together, this gives us the following expression for c'_κ:

$$c'_\kappa = \sum_{X \subseteq \mathcal{U}} (-1)^{|X|} \binom{a(X)}{\kappa}$$

Björklund et al. [6] give the following expression for c_κ:

$$c_\kappa = \sum_{X \subseteq \mathcal{U}} (-1)^{|X|} a(X)^\kappa$$

These expressions are identical except for the fact that the formula of Björklund et al. counts the number of set covers c_κ where they allow a single set to be picked multiple times.

Consider the effect of the branching rules on the incidence graph. A branch where we take a set does exactly the same operation on the graph as a branch where we forbid an element, only the former is on a red vertex while the latter is on a blue vertex. The same relation holds between a branch where we discard a set and a branch where we make an element optional: both branching types are symmetric to each other. This symmetry is not complete, however, because for other purposes the red and blue vertices are not equivalent. That is, blue vertices must be dominated by red vertices, which leads to different reduction rules depending on the colour of a vertex.

4 An Algorithm for Counting Dominating Sets

We now give an algorithm (Algorithm 1) for the #κ-Dominating Set problem. Our algorithm works on the #κ-Set Cover transformation of the problem and returns a list containing the number of set covers of size κ for each $0 \leq \kappa \leq n$. It is a branch and reduce algorithm, branching both on sets and on elements, following the methodology discussed in Section 3. This section will be devoted to the description of this algorithm except for a subroutine that employs pathwidth techniques. This subroutine is discussion in Section 6. We will start by describing two simple subroutines often used by Algorithm 1, after which we will describe Algorithm 1 from top to bottom.

The first subroutine `eliminate-set(`S`,`$\mathcal{S}$`)` removes the set S *and* all its elements from $\mathcal{S}$ in such a way that while S is removed any set that possibly turns into an empty set due to the removal of the elements of S remains as an empty sets in $\mathcal{S}$. If S has multiplicity greater than one, then all copies of S are removed as well. Secondly, `eliminate-element(`e`,`$\mathcal{S}$`)` removes the element e *and* all sets in $\mathcal{S}$ containing e.

$$\texttt{eliminate-set(}S\texttt{,}\mathcal{S}\texttt{)} = \{S' \backslash S \mid S' \in (\mathcal{S} \setminus \{S\})\}$$

$$\texttt{eliminate-elem(}e\texttt{,}\mathcal{S}\texttt{)} = \mathcal{S} \setminus \{S \in \mathcal{S} \mid e \in S\}$$

We want to emphasise here that when any of these two subroutines are called, then not only the the set S or the element e is removed, but also the elements or sets directly involved with it.

We are now ready for a complete, top to bottom, description of Algorithm 1. The algorithm takes as input a multiset of sets $\mathcal{S}$ forming a set cover instance $(\mathcal{S},\mathcal{U})$ over the universe $\mathcal{U} = \cup\mathcal{S}$, and it returns a list of length $\#(\mathcal{S}) + 1$ containing for each κ, $0 \leq \kappa \leq \#(\mathcal{S})$, the number of set covers of size exactly κ. Before branching or applying pathwidth techniques, the algorithm tries to reduce the instance to a simpler instance in polynomial time. To this end, it employs a series of reduction rules that form the first part of the algorithm. These will be described now.

Base Case
On some inputs, the set cover instance is completely reduced to a multiset of empty sets by the reduction rules below. This is handled by our *base case*. In this case, there are no elements left to cover and we have m (empty) sets left to choose from. Thus, the number of set covers of size κ equals $\binom{m}{\kappa}$ which is returned for all $0 \leq \kappa \leq m$.

Unique Elements
Whenever there exists an element e of frequency one in $\mathcal{U}$, the set S containing e must belong to every set cover because otherwise e will not be covered. Therefore, the algorithm takes this set and goes in recursion on the simplified instance returned by `eliminate-set(`S`,`$\mathcal{S}$`)`.

When there exists an element e which only occurs in a single set S from which there exist m copies, the algorithm does something similar. At least one of these sets must belong to the set cover, but regardless of the number of copies chosen in the cover, the same simplified subproblem is generated by the call to `eliminate-set(`S`,`$\mathcal{S}$`)`. Therefore, we can use the result of this one recursive call to compute the number of set covers of size κ as if we considered all possible number of copies of S the algorithm could have taken. The algorithm does so by summing over all possible number of sets i it could have taken, and for each such i, it computes the number of choices $\binom{m}{i}$

Algorithm 1 `Count-SC`$(\mathcal{S},d)$

Input: A multiset of sets $\mathcal{S}$ over the universe $\mathcal{U} = \cup\mathcal{S}$.

Output: A list of length $\#(\mathcal{S}) + 1$ containing the number of set covers of $(\mathcal{S},\mathcal{U})$ of each size $0 \leq \kappa \leq \#(\mathcal{S})$.

1: //reduction rules
2: **if** $\mathcal{S} = \{\emptyset^m\}, m \geq 0$ **then** //base case
3: **return** $(\binom{m}{0}, \binom{m}{1}, \ldots, \binom{m}{m})$
4: **else if** $\exists e \in S^m \in \mathcal{S} : \text{freq}(e) = 1$ **then** //unique elements
5: $n_{take} =$ `Count-SC(eliminate-set(`S`,`$\mathcal{S}$`))`
6: **return** $(n_0, n_1, \ldots, n_{\#(\mathcal{S})})$, where:
$n_\kappa = \sum_{i=\max(1,\kappa-\#(\mathcal{S})+m)}^{\min(\kappa,m)} \binom{m}{i} [n_{take}]_{\kappa-i}$
7: **else if** $\exists e, e' \in \mathcal{U} : \mathcal{S}[e] \subseteq \mathcal{S}[e']$ **then** //subsumption
8: **return** `Count-SC(`$\{S\backslash\{e'\} \mid S \in \mathcal{S}\}$`)`
9: **else if** $\exists \emptyset \subset \mathcal{C} \subset \mathcal{S} : \{\mathcal{S}[e] | e \in \mathcal{U}[\mathcal{C}]\} = \mathcal{C}$ **then** //connected components
10: Let $\bar{\mathcal{C}} = \mathcal{S}\backslash\mathcal{C}$, $n_\mathcal{C} =$ `Count-SC(`$\mathcal{C}$`)`, $n_{\bar{\mathcal{C}}} =$ `Count-SC(`$\bar{\mathcal{C}}$`)`
11: **return** `Merge-Components(`$n_\mathcal{C}$`,`$n_{\bar{\mathcal{C}}}$`)`
12: **end if**
13: //implicit reduction rule[1]: identical sets
14:
15: //branching or path decomposition
16: Let $S^m \in \mathcal{S}$ be of maximum cardinality and not an exceptional case[2]
17: Let $e \in \mathcal{U}$ be of maximum frequency, also not an exceptional case[2]
18: Preference order P: $\mathsf{S}_4 < \mathsf{S}_5 < \mathsf{S}_6 < \mathsf{E}_5 < \mathsf{E}_6 < \mathsf{S}_7 < \mathsf{E}_7 < \mathsf{E}_{\geq 8} = S_{\geq 8}$
19: **if** $\mathsf{S}_{|S|}$ and $\mathsf{E}_{\text{freq}(e)}$ are too small to be in P **then** //path decomposition
20: **return** `Count-SC-PW(`$\mathcal{S}$`)`
21: **else if** $\mathsf{E}_{\text{freq}(e)}$ is in the order P and $\mathsf{E}_{\text{freq}(e)} \not< \mathsf{S}_{|S|}$ **then** //element branch
22: $n_{optional} =$ `Count-SC(`$\{S'\backslash\{e\} \mid S' \in \mathcal{S}\}$`)`
23: $n_{forbidden} =$ (`Count-SC(eliminate-elem(`e`,`$\mathcal{S}$`))`; $0^{\#(e)}$)
24: **return** $n_{optional} - n_{forbidden}$
25: **else** //$\mathsf{S}_{|S|}$ is in the order P and $\mathsf{S}_{|S|} \not< \mathsf{E}_{\text{freq}(e)}$ //set branch
26: $n_{take} =$ `Count-SC(eliminate-set(`S`,`$\mathcal{S}$`))`
27: $n_{discard} =$ (`Count-SC(`$\mathcal{S}\backslash\{S^m\}$`)`; 0^m)
28: **return** $(n_0, n_1, \ldots, n_{|\mathcal{S}|})$, where:
$n_\kappa = \left(\sum_{i=\max(1,\kappa-|S|+m)}^{\min(\kappa,m)} \binom{m}{i} [n_{take}]_{\kappa-i}\right) + [n_{discard}]_\kappa$
29: **end if**

[1] The multiset representation makes the identical set rule implicit. We emphasise that identical sets created by branching are handled by multiplicity counters.

[2] There are some exceptional combinations of cardinalities of sets and frequencies of elements on which the algorithm will not branch. These will be handled by the path decomposition phase. For a complete list of these cases see Overview 1.

times the number of set covers of size $\kappa - i$ from the recursive call. See also, lines 5 and 6 of the pseudo code.

Subsumption
If there exists an element $e \in \mathcal{U}$ which occurs in every set (and possibly in more sets) in which another element $e' \in \mathcal{U}$ occurs, then every set cover that covers e also covers e'. In this case, we can remove e' from the current instance and obtain a simpler instance to which we recursively apply our algorithm.

Connected Components
If the incidence graph contains multiple connected components, then we can solve the problem on each component separately and merge the results. The subroutines `Merge-Components`$(n_{\mathcal{C}}, n_{\bar{\mathcal{C}}})$ performs this merging. Let $\mathcal{C}$, $\bar{\mathcal{C}}$ be two disjoint sets of connected component of $\mathcal{S}$ and let $n_{\mathcal{C}}$, $n_{\bar{\mathcal{C}}}$ be the solutions to these two subproblems. In order to compute the number of set covers of size κ for $\mathcal{C} \cup \bar{\mathcal{C}}$, this subroutine sums over all possible sizes i of set covers for $\mathcal{C}$ and multiplies this number by the number of set covers for $\bar{\mathcal{C}}$ of size $\kappa - i$.

$$\texttt{Merge-Components}(n_{\mathcal{C}}, n_{\bar{\mathcal{C}}}) = (n_0, n_1, \ldots, n_{\#(\mathcal{C} \cup \bar{\mathcal{C}})})$$

$$\text{where: } n_\kappa = \sum_{i=\max(0,\kappa-\#(\bar{\mathcal{C}}))}^{\min(\kappa,\#(\mathcal{C}))} [n_{\mathcal{C}}]_i \times [n_{\bar{\mathcal{C}}}]_{\kappa-i}$$

Identical Sets
Remind that the unique elements rule also handles elements that occur only in multiple copies of the same set. The idea behind this is that the same subproblem will be generated independent of the number of these identical set we choose, and this will be used throughout the algorithm. Therefore, we could say that our algorithm considers identical sets to be 'removed' and uses multiplicity counters it stead. By the notation and definition from Section 2, we will also not count identical sets twice in the dimension of the problem. Hence, we have an implicit reduction rule removing identical sets. This we use to our advantage in the analysis of the running time in Section 5.

Having treated the reduction rules we now continue with the branching steps of the algorithm.

When no reduction rules are applicable, the algorithm chooses a set of maximum cardinality from the sets in the instance that are not exceptional cases, and it chooses an element of maximum frequency from the instance that is also not an exceptional case. We postpone the discussion of these exceptional cases for a moment, and remark that this choice for maximum cardinality and frequency resembles choosing more efficient branchings. This is so, since if an elements frequency is larger, then more sets are excluded in the forbidden branch and more sets are reduced in cardinality in the optional branch, and similar considerations exist for set branches.

The algorithm needs to choose whether it is going to branch on a set or on an elements. For this it uses the following preference order P.

$$\mathsf{P}: \quad \mathsf{S}_4 < \mathsf{S}_5 < \mathsf{S}_6 < \mathsf{E}_5 < \mathsf{E}_6 < \mathsf{S}_7 < \mathsf{E}_7 < \mathsf{E}_{\geq 8} = S_{\geq 8}$$

In this ordering, $\mathsf{S}_i < \mathsf{E}_j$ means that the algorithm prefers to branch on an element of frequency j over branching on a set of cardinality i.

Notice that sets of cardinality at most three and elements of frequency at most four do not occur in the preference order P. These cardinalities are considered to be too small and these frequencies too low for efficient branching. Instances on which no efficient branching is possible are handled by path decomposition techniques by calling `Count-SC-PW`$(\mathcal{S})$.

The exceptional cases are described in Overview 1. These exceptional cases exist because in the analysis in Sections 5 and 6 we often know the neighbourhood of a vertex representing a set or an element in the incidence graph. Such neighbourhoods are important for the worst case behaviour of the algorithm. And, for some neighbourhoods, despite the general rule imposed by the preference order, it is more efficient to handle them by the path decomposition part of our algorithm than

There are exceptional cases of elements on which, despite the preference order, Algorithm 1 does not branch. These cases represent local neighbourhoods of sets or elements which would increase the running time of the algorithm when branched on, but can be handled by dynamic programming on a path decomposition quite effectively. The exceptional cases are:

1. Elements of frequency five that occur in many sets of small cardinality. More specifically, if we let a 5-tupple $(s_1, s_2, s_3, s_4, s_5, s_6)$ represent a frequency five element occurring s_i times in a cardinality i set, then our special cases can be denoted as:

   (1, 4, 0, 0, 0, 0) - (0, 5, 0, 0, 0, 0) - (1, 3, 1, 0, 0, 0) - (0, 4, 1, 0, 0, 0) - (1, 2, 2, 0, 0, 0)
   (0, 3, 2, 0, 0, 0) - (1, 1, 3, 0, 0, 0) - (0, 2, 3, 0, 0, 0) - (0, 1, 4, 0, 0, 0) - (1, 0, 4, 0, 0, 0)
   (1, 3, 0, 1, 0, 0) - (0, 4, 0, 1, 0, 0) - (1, 2, 1, 1, 0, 0) - (0, 3, 1, 1, 0, 0) - (1, 1, 2, 1, 0, 0)
   (1, 0, 3, 1, 0, 0) - (1, 2, 0, 2, 0, 0) - (1, 3, 0, 0, 1, 0) - (1, 2, 1, 0, 1, 0) - (1, 3, 0, 0, 0, 1)

2. Sets of cardinality four, five or six, that have one of the above elements contained in them.

Overview 1: Exceptional Cases for Algorithm 1

by branching. These neighbourhoods are our exceptional cases. How this influences the running time of our algorithms will become more clear from the analyses in Sections 5 and 6.

We conclude the description of Algorithm 1 by some remarks on the pseudo code of the branching steps. In the branch where an element e is forbidden, a number of zeros is added to the list containing the number of set covers of each size (line 23). This is done because the number of set covers of size κ for $n - \#(e) \leq \kappa \leq n$ equals zero since no sets containing e may be chosen. Also, we remark that when the algorithm branches on a set of multiplicity m, it sums over all possible number of identical copies it can take in the cover (line 28). This works in the same way as explained with the unique elements rule.

5 Measure and Conquer Analysis

We analyse Algorithm 1 using the measure and conquer methodology [11, 13]. To this end, we introduce a non standard complexity measure $k(\mathcal{S},\mathcal{U})$ on problem instances; we introduce weight functions $v, w : \mathbb{N} \to [0, 1]$ giving weight $v(i)$ to an element of frequency i and weight $w(i)$ to a set of cardinality i, respectively. This gives us the following complexity measure:

$$k(\mathcal{S},\mathcal{U}) = \sum_{S \in \text{set}(\mathcal{S})} w(|S|) + \sum_{e \in \mathcal{U}} v(\text{freq}(e))$$

This measure is identical to the one used in [11, 25]. Notice that k is at most the dimension d of the set cover instance, and hence if we prove the running time of the algorithm to be $\mathcal{O}(\alpha^k)$, we have also proved a running time of $\mathcal{O}(\alpha^d)$. For convenience, we let $\Delta v(i) = v(i) - v(i-1)$ and $\Delta w(i) = w(i) - w(i-1)$ be the complexity reductions gained by reducing the frequency of an element or the cardinality of a set by one.

We start the analysis of the running time of Algorithm 1 by bounding the number of subproblems generated by branching.

Lemma 1 *Let $N_h(k)$ be number of subproblems of measured complexity h generated by Algorithm 1 on an input of measured complexity k. Then:*

$$N_h(k) < 1.22670^{k-h}$$

Proof. To correctly analyse the branching, we will use the following constraints on the weights:

1. $v(0) = v(1) = w(0) = 0$
2. $\Delta v(i) \geq 0$ for all $i \leq 1$
3. $\Delta w(i) \geq 0$ for all $i \leq 1$
4. $\Delta v(i) \geq \Delta v(i+1)$ for all $i \geq 1$
5. $\Delta w(i) \geq \Delta w(i+1)$ for all $i \geq 1$
6. $2\Delta v(5) \leq v(2)$

First, we observe that we can set the weight of elements of frequency one and sets of cardinality zero to zero because they are removed by the reduction rules or ignored in the branching phase, respectively. Second, we do not want the complexity of our instance to increase when decreasing the frequency of an element or the cardinality of a set, therefore weights must be increasing. This covers restrictions 1-3. Restrictions 4-6 will be explained during the analysis below.

Consider branching on an element e contained in s_i sets of cardinality i. In the branch where e is optional, the element e is removed and all sets containing e are reduced in cardinality by one. And, in the branch where e is forbidden, e is removed together with all sets containing e. The removal of these sets also results in a reduction of the frequencies of all other elements in these sets. This leads to two subproblems which are reduced in complexity by $\Delta k_{optional}$ and $\Delta k_{forbidden}$, respectively.

$$\begin{aligned} \Delta k_{optional} &= v(\text{freq}(e)) + \sum_{i=1}^{\infty} s_i \Delta w(i) \\ \Delta k_{forbidden} &= v(\text{freq}(e)) + \sum_{i=1}^{\infty} s_i w(i) + \Delta v(\text{freq}(e)) \sum_{i=1}^{\infty} (i-1) s_i \end{aligned}$$

Here we bound each extra reduction of the frequency of an element because of the removal of a set by $\Delta v(\text{freq}(e))$. This is correct because when we branch on an element, it is of highest frequency and we have the constraints $\Delta v(i) \geq \Delta v(i+1)$ (Constraint 4), and this frequency is at least five and $2\Delta v(5) \leq v(2)$ (Constraint 6). Constraint 4 assures that each time we reduce the cardinality of a set more than once, the values $\Delta v(i)$ for $i < \text{freq}(e)$ are large enough, and Constraint 6 handles the fact that an element can be completely removed if it occurs only in removed sets. Without this constraint, we could have taken all the weight from this element already since $v(1) = 0$, leaving none for its final occurrence.

Now consider branching on a set S containing e_i elements of frequency i. As discussed in Section 3, the behaviour is similar to an element branch, but with the roles of elements and sets interchanged. In the branch where S is discarded, the set S is removed and all its elements have their frequency reduced by one. And, in the branch where we take S, the set S is removed together with all its elements which leads to the reduction of the cardinalities of other sets. However, the similarity ends when elements of frequency two are involved. After discarding S, these elements will occur uniquely in the instance, and hence some set is included in the solution by the unique elements rule. These elements cannot have their second occurrence in the same sets since that would have triggered the subsumption rule before branching; we remove an additional set of cardinality at least one for each such element.

This leads to the following values for $\Delta k_{discard}$, Δk_{take}:

$$\begin{aligned} \Delta k_{discard} &= w(|S|) + \sum_{i=2}^{\infty} e_i \Delta v(i) + e_2 w(1) \\ \Delta k_{take} &= w(|S|) + \sum_{i=2}^{\infty} e_i v(i) + \Delta w(|S|) \sum_{i=2}^{\infty} (i-1) e_i \end{aligned}$$

Here we use Constraint 5 ($\Delta w(i) \geq \Delta w(i+1)$) to bound the additional reduction in complexity due to the reduction in cardinalities of sets after covering and removing elements. This is almost similar to the analysis of branching on elements.

Recall from the statement of Lemma 1 that $N_h(k)$ is the number of subproblems of measured complexity h generated to solve a problem instance of measured complexity k. By the above

considerations we have:

$$N_h(k) \leq N_h(k - \Delta k_{optional}) + N_h(k - \Delta k_{forbidden})$$

$$N_h(k) \leq N_h(k - \Delta k_{discard}) + N_h(k - \Delta k_{take})$$

with the appropriate values of $\Delta k_{optional}$ and $\Delta k_{forbidden}$ for every possible element branch, and the appropriate values $\Delta k_{discard}$ and Δk_{take} for every possible set branch. The solution of this set of recurrence relations is a function of the form α^{k-h} where α is the largest positive real root of the corresponding set of equations:

$$1 = \alpha^{-\Delta k_{optional}} + \alpha^{-\Delta k_{forbidden}} \qquad 1 = \alpha^{-\Delta k_{discard}} + \alpha^{-\Delta k_{take}}$$

for all $|S| = \sum_{i=2}^{\infty} e_i$ and all $\text{freq}(e) = \sum_{i=1}^{\infty} s_i$ agreeing with the preference order P except for the exceptional cases in Overview 1 (see Section 4). Also, for each element branch $s_1 \leq 1$ since an element cannot occur twice in a cardinality one set by the identical sets rule.

Notice that because we do not branch on the first kind of exceptional cases from Overview 1, we must keep in mind that there is a second kind of exceptional cases created by the fact that we only branch on local neighbourhoods respecting the preference order P. These are the second kind of exceptional cases in Overview 1.

What remains is to choose weight functions that respect the constraints and minimise the solution to the set of recurrence relations. We simplify this optimisation problem by noting that the weight functions will converge to 1 at some point p resulting in:

$$w(p') = v(p') = 1 \quad \text{and} \quad \Delta w(p'+1) = \Delta v(p'+1) = 0 \quad \text{for all } p' \geq p$$

Therefore, the recurrence relations corresponding to $|S| > p+1$ and $\text{freq}(e) > p+1$ are dominated by those corresponding to $|S| > p$ and $\text{freq}(e) = p$, respectively. This leads to a large, but finite, numerical optimisation problem (quasiconvex program [9]). We solve this by computer obtaining an upper bound on the number of subproblems of measured complexity h generated. This upper bound is 1.22670^{k-h} using the following set of weights:

i	1	2	3	4	5	6	≥ 7
$v(i)$		0.409958	0.776286	0.982138	0.995507	1	1
$w(i)$	0.367311	0.614495	0.767367	0.870084	0.972800	0.996358	1

This proves the lemma. □

Notice that the above Measure and Conquer analysis is more difficult to perform compared to a standard analysis. In particular, there are different preference orders, and for each preference order, we obtain a new quasiconvex program. Moreover, for each such order, all possible combinations of s_i or e_i need to be put to a test for whether they can be included in this analysis (representing cases to branch on), or whether they can be excluded from the analysis and handled more efficiently by dynamic programming on the path decomposition. The preference order and exceptional cases used in Algorithm 1 appear to give the best bound on the running time. We found these by both exhaustive search and trial and error by hand.

6 Path Decomposition Dynamic Programming

In this section we will discuss the subroutine `Count-SC-PW`($\mathcal{S}$). Algorithm 1 calls this subroutine if an efficient branching is not possible, that is, there do not exist any large sets or high frequency elements that are not in any of the exceptional cases in Overview 1. Note that it solves the same problem as Algorithm 1, with extra information on the possible cardinalities of sets and frequencies of elements. This subroutine uses dynamic programming on path decompositions. The combination of pathwidth techniques combined with a measure and conquer analysis on branching was introduced by Fomin et al. [10].

We will start by giving some terminology on path decompositions.

Definition (Path decomposition) A *path decomposition* of a graph $G = (V, E)$ is a sequence of bags (sets of vertices) $X = \langle X_1, \ldots, X_r \rangle$ such that:

- $\bigcup_{i=1}^{r} X_i = V$.
- for each $(u, v) \in E$, there exists a X_i such that $\{u, v\} \subseteq X_i$.
- if $v \in X_i$ and $v \in X_k$ then $v \in X_j$, for all $i \leq j \leq k$.

The bag X_i is said to *introduce* $v \notin X_{i-1}$ if $X_i = X_{i-1} \cup \{v\}$ and it is said to *forget* $v \in X_{i-1}$ if $X_i = X_{i-1} \setminus \{v\}$. If for all $2 \leq i \leq r$, X_i either introduces or forgets a vertex, then X is called a *nice path decomposition*. The *width* of X is $\max_{1 \leq i \leq r} |X_i| - 1$ and the pathwidth $\mathrm{pw}(G)$ of G is the minimum over the widths of all its path decompositions. It is easy to transform any path decomposition into a nice path decomposition of equal width in polynomial time.

The subroutine `Count-SC-PW`($\mathcal{S}$) uses dynamic programming on a path decomposition on the incidence graph $G_{\mathcal{S}}$ of our set cover instance $\mathcal{S}$.

Proposition 2 *Given $G_{\mathcal{S}} = (V_{Red} \cup V_{Blue}, E)$ and a nice path decomposition X of $G_{\mathcal{S}}$ of width at most p, the number of red/blue dominating sets for $G_{\mathcal{S}}$ of each size $0 \leq \kappa \leq |X_{Red}|$ can be counted in $\mathcal{O}(2^p n^{\mathcal{O}(1)})$ time.*

Proof. Consider the standard dynamic programming algorithm computing the minimum red/blue dominating set in $G_{\mathcal{S}}$ using the path decomposition X. A vertex in X_i can have two states: it is in the dominating set or not; and a blue vertex in X_i can also have two states: it is dominated or not. This algorithm runs in $\mathcal{O}(2^p n^{\mathcal{O}(1)})$.

Using the same structure, we construct a new dynamic programming algorithm computing for each state of the vertices in X_i and for each $0 \leq \kappa \leq |V_{Red}|$ the number of red/blue dominating sets on $G[(\bigcup_{1 \leq j \leq i} X_j)]$ of size κ that agree with these states on X_i. This increases the (exponential) number of states by only a linear factor. Since each state can still be computed in polynomial time using the dynamic program, the new algorithm also runs in $\mathcal{O}(2^p n^{\mathcal{O}(1)})$. □

What remains is to prove that we can compute a path decomposition of suitably bounded pathwidth on the input graphs of `Count-SC-PW`($\mathcal{S}$) in polynomial time. To this end, we use the following result by Fomin et al. [10].

Proposition 3 ([10]) *For any $\varepsilon > 0$, there exists and integer n_ε such that for every graph G with $n > n_\varepsilon$ vertices,*

$$\mathrm{pw}(G) \leq \frac{1}{6} n_3 + \frac{1}{3} n_4 + \frac{13}{30} n_5 + \frac{23}{45} n_6 + n_{\geq 7} + \epsilon n$$

where n_i is the number of vertices of degree i in G for any $i \in \{3, \ldots, 6\}$ and $n_{\geq 7}$ is the number of vertices of degree at least 7. Moreover, a path decomposition of the corresponding width can be constructed in polynomial time.

Remark 4. If our graph $G_{\mathcal{S}}$ has a vertex of degree d with i degree one neighbours, then this vertex can be considered to be a degree $d - i$ vertex in Proposition 3. Namely, if we remove all degree one vertices from $G_{\mathcal{S}}$ and then compute its path decomposition, then we can reintroduce these vertices by inserting consecutive introduce and forget bags in the decomposition, increasing its pathwidth by at most one.

Now we are ready to prove a bound on the running time of `Count-SC-PW`($\mathcal{S}$).

Lemma 5 *`Count-SC-PW`($\mathcal{S}$) runs in time $\mathcal{O}(1.2226^k)$ when applied to a set cover instance of measured complexity k.*

Proof. We will now prove an upper bound on the pathwidth of graphs input to `Count-SC-PW`($\mathcal{S}$) of measured complexity k. To this end we formulate a linear program in which all variables have

the domain $[0, \infty)$. In a simpler form, this was also done by Fomin et al. [10]. From this paper, we take the first part of the linear program:

$$\begin{aligned} \max \quad z &= \frac{1}{6}n_3 + \frac{1}{3}n_4 + \frac{13}{30}n_5 + \frac{23}{45}n_6 \qquad \text{such that:} \\ 1 &= \sum_{i=1}^{6} w(i)x_i + \sum_{i=2}^{5} v(i)y_i \qquad (1) \\ \sum_{i=1}^{6} ix_i &= \sum_{i=2}^{5} iy_i \qquad (2) \end{aligned}$$

In this linear program, x_i and y_i represent the number of sets of cardinality i and elements of frequency i per unit of measured complexity in a worst case instance, respectively. Notice that a subproblem on which `Count-SC-PW`($\mathcal{S}$) is called can have sets of cardinality at most six, from which the cardinalities four, five and six only occur as exceptional cases (Overview 1), and elements of frequency at most five. Constraint 1 guarantees that these x_k and y_k use exactly one unit of measured complexity. And, Constraint 2 guarantees that both partitions of the bipartite incidence graph have an equal number of edges.

The objective function, however, is formulated in terms of the variables n_i. It represents the maximum pathwidth z of a graph per unit of measured complexity using Proposition 3. The variables n_i represent the number of vertices of degree i in the input graph per unit of measured complexity. Following Remark 6, n_i is the number of sets of cardinality i plus the number of elements of frequency i that have no cardinality one set occurrence and the number of elements of frequency $i+1$ that do occur in a cardinality one set. These size one sets will use measure and not increase the pathwidth, so we can assume that they will only exist if involved in an exceptional case. More precisely, if we let C be the set of exceptional frequency five element cases defined in Overview 1, and let c_i be the number of occurrences in a cardinality i set for exceptional case $c \in C$, then we can introduce the variables p_c for the number of occurrences of each exceptional case $c \in C$ per unit of measured complexity. This gives us to following additional constraints to the linear program:

$$n_3 = x_3 + y_3 \qquad (3)$$

$$n_4 = x_4 + y_4 + \sum_{c \in C, c_1 > 0} p_c \qquad (4)$$

$$n_5 = x_5 + \sum_{c \in C, c_1 = 0} p_c \qquad (5)$$

$$n_6 = x_6 \qquad (6)$$

$$y_5 = \sum_{c \in C} p_c \qquad (7)$$

The next thing to do is to add additional constraints justified by the exceptional cases. Observe that whenever $p_c > 0$ for some $c \in C$, then there exist further restrictions on the instance because we know the cardinalities of the sets in which these exceptional element occur. We lower bound the number of sets of cardinalities one, two and three in the instance using Constraint 8. And, we upper bound the number of sets of cardinality four, five and six by using that there can be at most one such set per exceptional frequency five element contained in it. This is done in Constraint 9.

$$x_i \geq \sum_{c \in \mathcal{C}} \frac{c_i}{i} p_c \qquad \text{for } i \in \{1, 2, 3\} \qquad (8)$$

$$x_i \leq \sum_{c \in \mathcal{C}} c_i p_c \qquad \text{for } i \in \{4, 5, 6\} \qquad (9)$$

The solution to this linear program is $z = 0.28991$ with all variables equal to zero, except:

$$\begin{array}{lll} x_1 = 0.267603 & x_3 = 0.267603 & x_4 = 0.267603 \\ y_4 = 0.200703 & y_5 = 0.267603 & n_3 = 0.267603 \\ n_4 = 0.735910 & p_{(1,0,3,1,0,0)} = 0.267603 & \end{array}$$

As a result the dynamic program on this path decomposition runs in time $\mathcal{O}(2^{zk}n^{\mathcal{O}(1)})$ which equals $\mathcal{O}(1.2226^k)$. We complete the proof by noting that although Proposition 3 only applies to graphs of size at least n_ϵ, the result holds because we can fix ϵ to be small enough to disapear in the rounding of the running time and consider all smaller graphs than n_ϵ to be handled in constant time. □

Combining Lemma 1 and Lemma 5 gives our main result.

Theorem 6 *Algorithm 1 computes the number of dominating sets of all sizes $0 \leq \kappa \leq n$ in an n vertex graph G in $\mathcal{O}(1.5048^n)$ when it is applied to a set cover formulation of this problem.*

Proof. Let $T(k)$ be the time used on a problem of measured complexity k, and let H_k be the set of all possible complexities of subproblems of a problem of complexity k. Then by Lemma 1 and Lemma 5:

$$T(k) \leq \sum_{h \in H_k} N_h(k) \cdot 1.2226^h \leq \sum_{h \in H_k} 1.22670^{k-h} \cdot 1.2226^h \leq \sum_{h \in H_k} 1.22670^k$$

Since $|H_k|$ is polynomially bounded because we only use a finite number of weights, Algorithm 1 runs in time $\mathcal{O}(1.22670^d)$, where d is the dimension of the set cover problem instance. Using the standard set cover formulation of dominating set, this gives a running time of $\mathcal{O}(1.22670^{2n}) < \mathcal{O}(1.5048^n)$. □

7 Algorithms for Dominating Set Restricted to Some Graph Classes

The algorithm, given and analysed in Sections 4-6, not only gives the currently fastest algorithm to compute the number of dominating sets of given sizes, but also is the currently fastest algorithm for the minimum dominating set problem. However, the improvement over the previous fastest minimum dominating set algorithm [25] is only small. When we consider the dominating set problem on specific graph classes, we get a much larger improvement with our approach.

Gaspers et al. consider exact algorithms for the dominating set problem on special graph classes on which this problem is still NP-complete [16]. They consider c-dense graphs, circle graphs, chordal graphs, 4-chordal graphs, and weakly chordal graphs. They show that if we restrict ourselves to such a graph class, then there are either many vertices of high degree allowing more efficient branching, or the graph has low treewidth allowing us to efficiently solve the problem by dynamic programming over a tree decomposition. In this section, we will show that by using our two branching rules we need less vertices of high degree to obtain the same effect with more effective branching, further improving the results on four of these graph classes.

We begin by showing that having many vertices of high degree is beneficial to the running time of an algorithm in our approach. Combining the results of Section 4-6 with a result from Gaspers et al. [16], we directly obtain:

Proposition 7 ([16], Theorem 6) *Let $t > 0$ be a fixed integer, and let $\mathcal{G}_t$ be a class of graphs with for all $G = (V, E) \in \mathcal{G}_t$: $|\{v \in V \ : \ d(v) \geq t - 2\}| \geq t$. Then there is a $\mathcal{O}(1.22670^{2n-t})$ time algorithm to solve the minimum dominating set problem on graphs in $\mathcal{G}_t$.*

We will now give and prove a stronger variant of this proposition.

Lemma 8 *Let $t > 0$ be a fixed integer, and let $\mathcal{G}_t$ be a class of graphs with for all $G = (V, E) \in \mathcal{G}_t$: $|\{v \in V \ : \ d(v) \geq t - 2\}| \geq \frac{1}{2}t$. Then there is a $\mathcal{O}(1.22670^{2n-t})$ time algorithm to solve the minimum dominating set problem on graphs in $\mathcal{G}_t$.*

Proof. Let H be the set of vertices of degree at least $t - 2$ from the statement of the lemma, and consider the set cover formulation of the dominating set problem.

Let S be a set corresponding to a vertex in H. We branch on this set and consider the branch in which we take this set in the set cover: the set is removed and all its elements are covered and hence removed also. These are at least $t-1$ elements, and therefore this branch results in a problem of dimension at most $2n-t$. Only a single set is removed in the other branch, in which case we repeat this process and branch on the next set represented by another vertex in H. This gives us $\frac{1}{2}t$ problem instances of dimension at most $2n-t$ and one problem instance of dimension $2n-\frac{1}{2}t$ because here $\frac{1}{2}t$ sets are removed.

In this latter instance, we use our new inclusion/exclusion based branching rule on the elements corresponding to the vertices in H. These elements still have frequency at least $\frac{1}{2}t-1$, since only $\frac{1}{2}t$ sets have been discarded until now. When branching on an element and forbidding it, a subproblem of dimension at most $2n-t$ is created because at least an additional element and $\frac{1}{2}t-1$ sets are removed in this branch. What remains is one subproblem generated in the branch after discarding $\frac{1}{2}t$ sets and making $\frac{1}{2}t$ elements optional. Since all these sets and elements are removed in these branches, this also gives us a problem of dimension $2n-t$.

The above procedure generates $t+1$ problems of dimension $2n-t$, which can all be solved by Algorithm 1 in $\mathcal{O}(1.22670^{2n-t})$ time. These are only a linear number of instances giving us a total running time of $\mathcal{O}(1.22670^{2n-t})$. □

Definition (c-dense graph) A graph $G=(V,E)$ is said to be c-dense, if $|E| \geq c$ where c a constant with $0 < c < \frac{1}{2}$.

If we combine our dominating set algorithm with the approach from Gaspers et al. [16], we obtain an $\mathcal{O}\left(1.22670^{(1+\sqrt{1-2c})n}\right)$ algorithm that counts the number of dominating sets of each size on a c-dense graph. Using Lemma 8 we improve upon this, as shown below. A graphical comparison of both results can be found in Figure 1.

Corollary 9 *The number of dominating sets of all sizes $0 \leq \kappa \leq n$ in a c-dense graph can counted in $\mathcal{O}\left(1.22670^{(\frac{1}{2}+\frac{1}{2}\sqrt{9-16c})n}\right)$.*

Proof. By a counting argument in [16], any graph has a set of high degree vertices that allow application of Lemma 8 with parameter t if it has enough edges. For c-dense graphs this is when:

$$|E| \geq cn^2 \geq \frac{1}{2}\left(\frac{1}{2}t-1\right)(n-1) + \frac{1}{2}\left(n-\frac{1}{2}t+1\right)(t-3)$$

If $t \leq \frac{1}{2}(4+3n) - \frac{1}{2}\sqrt{-8n+9n^2-16cn^2}$ then this is the case. By taking t maximal in this inequality and removing all factors that disappear in the big-$\mathcal{O}$, we obtain a running time of $\mathcal{O}\left(1.22670^{(\frac{1}{2}+\frac{1}{2}\sqrt{9-16c})n}\right)$. □

Note that this results also gives the current fastest algorithm for minimum dominating set on c-dense graphs.

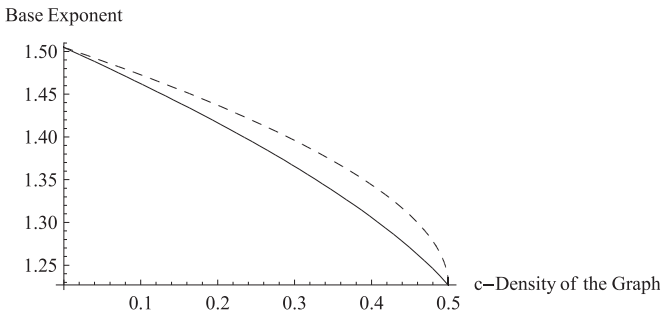
We now consider circle graphs, chordal graphs, 4-chordal graphs, and weakly chordal graphs. Let us first introduce these graph classes.

Definition (Circle graph) A *circle graph* is an intersection graph of chords in a circle: every vertex represents a chord, and vertices are adjacent if the cords intersect.

Definition (Chordal graph) A graph is *chordal* if it has no chordless cycle.

Definition (4-Chordal graph) A graph is *4-chordal* if it has no chordless cycle of length more than four.

Definition (Weakly chordal graph) A graph G is weakly chordal if both G and its complement are 4-chordal.



The solid line represents the upper bound on the running time of our algorithm, and the dashed line represents the upper bound obtained from [16] after plugging in our faster algorithm for dominating set.

Figure 1: Comparison of bounds on the running time on c-dense graphs.

On these graph classes Gaspers et al. [16] balance dynamic programming over tree decompositions to the many vertices of high degree approach.

Definition (Tree decomposition) A *tree decomposition* of a graph $G = (V, E)$ is a tree T with in each node a bag (set of vertices) X_t such that:

- $\bigcup_{X_t \in T} X_t = V$.
- for each $(u, v) \in E$, there exists a $X_t \in T$ such that $\{u, v\} \subseteq X_t$.
- for all $X_1, X_2, X_3 \in T$, if X_2 is on the path from X_1 to X_3 in T, then $X_1 \cap X_3 \subseteq X_2$.

The *width* of a tree decomposition T is $\max_{X_i \in T} |X_i| - 1$ and the treewidth $\text{tw}(G)$ of G is the minimum over the widths of all its tree decompositions.

Similar to path decompositions, a tree decomposition is a *nice tree decomposition* if it is a binary tree and every non-leaf bag is either an introduce, forget or join bag. In this terminology, the bag X_t is said to *introduce* v if it equals its child plus the vertex v, and it is said to *forget* v if it equals its child minus the vertex v. Additionally, the bag X_t joins its right child X_r and its left child X_l if $X_r = X_l = X_t$. Kloks [20] has shown that any tree decomposition can be transformed into a nice tree decomposition of equal width in polynomial time.

A nice tree decomposition can, just as a nice path decomposion, be used for dynamic programming. For more information on tree decompositions and dynamic programming over tree decompositions see [7, 8]. A straightforward dynamic programming algorithm for dominating set running in $\mathcal{O}(4^t n^{\mathcal{O}(1)})$ can be found in [1].

We use a faster algorithm for dominating set on graphs of bounded treewidth [23] using fast subset convolutions [5].

Proposition 10 ([23]) *Given an n-vertex graph $G = (V, E)$ and nice tree decomposition T of G of width t, the number of dominating sets in G of each size $0 \leq \kappa \leq n$ can be counted in $\mathcal{O}(3^t n^{\mathcal{O}(1)})$ time.*

The following lemma is based on [16] and gives our running times on the other graph classes. Note that a graph class $\mathcal{G}$ is a hereditary graph class if all induced subgraphs of any graph $G \in \mathcal{G}$ are in $\mathcal{G}$ too.

Lemma 11 *Let $\mathcal{G}$ be a hereditary class of graphs such that $tw(G) \leq c\Delta(G)$ for all $G \in \mathcal{G}$ and for which such tree decompositions can be computed in polynomial time. Then there is a $\mathcal{O}\left(\max\left(1.22670^{2n-t'n}, 3^{(c+\frac{1}{2})t'n}\right)\right)$ time algorithm that counts the number of dominating sets of each size in a graph $G = (V, E)$. Both terms are balanced if: $t' = \frac{4}{2+d+2cd}$ where $d = 1/\log_3(1.22670)$.*

Proof. Let X be the set of vertices of degree at least $t'n$. If $|X| \geq \frac{1}{2}t'n$ then we can apply Lemma 8 with $t = t'n$ giving a running time of $\mathcal{O}(1.22670^{2n-t'n})$.

Otherwise $|X| \leq \frac{1}{2}t'n$ and $\Delta(G[V \setminus X]) < t'n$. Since $G[V \setminus X]$ belongs to $\mathcal{G}$ we know that:

$$\mathrm{tw}(G) \leq \mathrm{tw}(G[V \setminus X]) + |X| \leq c\Delta(G[V \setminus X]) + |X| < ct'n + \frac{1}{2}t'n = \left(c + \frac{1}{2}\right)t'n$$

Now we can apply Proposition 10 giving us a running time of $\mathcal{O}(3^{(c+\frac{1}{2})t'n})$.

The balancing follows from basic calculations. □

Proposition 12 ([16]) *The following graph classes are hereditary graph classes with $tw(G) \leq c\Delta(G)$ for all $G \in \mathcal{G}$ using the following values of c.*

- *Circle graphs ($c = 4$).*
- *4-Chordal graphs ($c = 3$).*
- *Weakly Chordal graphs ($c = 2$).*

The corresponding tree decomposition can be computed in polynomial time.

Corollary 13 *There exist algorithms that count the number of dominating sets of each size in a circle graph in time $\mathcal{O}(1.4806^n)$, in a 4-chordal graph in $\mathcal{O}(1.4741^n)$, and in a weakly chordal graph in $\mathcal{O}(1.4629^n)$.*

Proof. Combine Lemma 11 and Proposition 12. □

As a final remark, we state that we could not use our two branching rules to improve the result on chordal graphs beyond plugging in our faster algorithm for general dominating set. We combine the approach of Gaspers et al. [16] with Theorem 6 and obtain the following proposition.

Proposition 14 *There exist algorithms that count the number of dominating sets of each size in a chordal graph in time $\mathcal{O}(1.3712^n)$.*

Proof. Following [16], combine Algorithm 1 with a $\mathcal{O}(2^t n^{\mathcal{O}(1)})$ algorithm on chordal graphs of treewidth t. □

8 Conclusion

In this paper we have show that we can use inclusion/exclusion based branching in combination with traditional branching and analyse such an algorithm by means of measure and conquer. Since the use of inclusion/exclusion restricts you to counting problems, which allow less reduction rules than their decision counterparts, we shifted to pathwidth based techniques on sparse instances. This combination resulted in an algorithm that computes the number of dominating sets of each cardinality slightly faster than any previous algorithm that computes a single minimum dominating set. Furthermore, we have shown that this leads to a significant speed up when restricted to some graph classes, while we still compute much more information.

While the improvement of the running time for the studied problems are interesting, we believe that the most important contribution of our paper is the novel combination of inclusion/exclusion and branching with a measure and conquer analysis. This gives a nice way to get rid of the usual $\mathcal{O}(2^n n^{\mathcal{O}(1)})$ running times obtained by inclusion/exclusion algorithms when applied to general graphs.

We note that we can use the same algorithm to the weighted versions of the problems as long as the size of the set of possible weight sums Σ is polynomially bounded. The only modification necessary is to make the algorithms compute the number of set covers of each possible weight $w \in \Sigma$ at each step.

Our algorithms are highly dependent on the current best known upper bounds on the pathwidth of bounded degree graphs [10, 14]. Any result that would improve these bounds would also improve our algorithm.

We remark that we use dynamic programming over path decompositions, while tree decompositions are more general and would allow the same running times by using fast subset convolutions to perform join operations [5, 23]. Kneis et al. [21] have some results on treewidth bounds for bounded degree graphs, but weaker than the pathwidth results we use. We consider it to be an important open problem to give stronger (or even tight) bounds on the treewidth or pathwidth of bounded degree graphs for which decompositions can be computed efficiently.

For further research, we are looking at more problems to which our inclusion/exclusion meets measure and conquer approach can be applied.

Acknowledgements

We would like to thank our advisor Hans L. Bodlaender for his enthusiasm for this research and for useful comments on an earlier draft of this paper. We would also like to thank Alexey A. Stepanov for useful discussions on [10].

References

[1] J. Alber, H. L. Bodlaender, H. Fernau, T. Kloks, and R. Niedermeier. Fixed parameter algorithms for dominating set and related problems on planar graphs. *Algorithmica*, 33:461–493, 2002.

[2] E. T. Bax. Inclusion and exclusion algorithm for the hamiltonian path problem. *Information Processing Letters*, 47(4):203–207, 1993.

[3] E. T. Bax. Recurrence-based reductions for inclusion and exclusion algorithms applied to #P problems, 1996.

[4] A. Björklund and T. Husfeldt. Exact algorithms for exact satisfiability and number of perfect matchings. *Algorithmica*, 52(2):226Ű249, 2008.

[5] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. Fourier meets Möbius: fast subset convolution. In *STOC '07: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 67–74, New York, NY, USA, 2007. ACM.

[6] A. Björklund, T. Husfeldt, and M. Koivisto. Set partitioning via inclusion-exclusion. *SIAM Journal of Computing, Special Issue for FOCS 2006, to appear.*

[7] H. L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–23, 1993.

[8] H. L. Bodlaender and A. M. C. A. Koster. Combinatorial optimization on graphs of bounded treewidth. *The Computer Journal*, 51(3):255–269, 2008.

[9] D. Eppstein. Quasiconvex analysis of backtracking algorithms. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004*, pages 781–790, 2004.

[10] F. V. Fomin, S. Gaspers, S. Saurabh, and A. A. Stepanov. On two techniques of combining branching and treewidth. *Algorithmica.* Special issue of ISAAC 2006, to appear.

[11] F. V. Fomin, F. Grandoni, and D. Kratsch. Measure and conquer: Domination — a case study. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming, ICALP 2005*, pages 191–203. Springer Verlag, Lecture Notes in Computer Science, vol. 3580, 2005.

[12] F. V. Fomin, F. Grandoni, and D. Kratsch. Some new techniques in design and analysis of exact (exponential) algorithms. *Bulletin of the EATCS*, 87:47–77, 2005.

[13] F. V. Fomin, F. Grandoni, and D. Kratsch. Measure and conquer: a simple $O(2^{0.288n})$ independent set algorithm. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006*, pages 18–25, 2006.

[14] F. V. Fomin and K. Høie. Pathwidth of cubic graphs and exact algorithms. *Information Processing Letters*, 97:191–196, 2006.

[15] F. V. Fomin, D. Kratsch, and G. J. Woeginger. Exact (exponential) algorithms for the dominating set problem. In J. Hromkovič, M. Nagl, and B. Westfechtel, editors, *Proceedings 30th International Workshop on Graph-Theoretic Concepts in Computer Science WG'04*, pages 245–256. Springer Verlag, Lecture Notes in Computer Science, vol. 3353, 2004.

[16] S. Gaspers, D. Kratsch, and M. Liedloff. Exponential time algorithms for the minimum dominating set problem on some graph classes. In L. Arge and R. Freivalds, editors, *Proceedings of the 10th Scandinavian Workshop on Algorithm Theory, SWAT 2006*, pages 148–159. Springer Verlag, Lecture Notes in Computer Science, vol. 4059, 2006.

[17] F. Grandoni. A note on the complexity of minimum dominating set. *J. Disc. Alg.*, 4:209–214, 2006.

[18] K. Iwama. Worst-case upper bounds for ksat. *Bulletin of the EATCS*, 82:61–71, 2004.

[19] R. M. Karp. Dynamic programming meets the principle of inclusion-exclusion. *Operations Research Letters*, 1:49–51, 1982.

[20] T. Kloks. *Treewidth. Computations and Approximations.* Lecture Notes in Computer Science, Vol. 842. Springer-Verlag, Berlin, 1994.

[21] J. Kneis, D. Mölle, S. Richter, and P. Rossmanith. Algorithms based on the treewidth of sparse graphs. In *Proceedings of the 31st International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2005)*, volume 3787 of *LNCS*, pages 385–396. Springer, 2005.

[22] B. Randerath and I. Schiermeyer. Exact algorithms for minimum dominating set. Technical Report zaik2005-501, Universität zu Köln, Cologne, Germany, 2005.

[23] P. Rossmanith. Using fast set convolutions to compute minimal dominating sets. Talk at Dagstuhl Seminar 07281, Structure Theory and FPT Algorithmics for Graphs, Digraphs and Hypergraphs, Juli 2007.

[24] U. Schöning. Algorithmics in exponential time. In *Proceedings of the 22nd International Symposium on Theoretical Aspects of Computer Science (STACS 2005)*, pages 36–43. Springer Verlag, Lecture Notes in Computer Science, vol. 3404, 2005.

[25] J. M. M. van Rooij and H. L. Bodlaender. Design by measure and conquer – a faster exact algorithm for dominating set. In S. Albers and P. Weil, editors, *Proceedings of the 25th Annual Symposium on Theoretical Aspects of Computer Science, STACS 2008*, pages 657–668. IBFI Schloss Dagstuhl, 2008.

[26] G. J. Woeginger. Exact algorithms for NP-hard problems: A survey. In *Combinatorial Optimization: "Eureka, you shrink"*, pages 185–207, Berlin, 2003. Springer Lecture Notes in Computer Science, vol. 2570.