Algorithmica (2014) 69:685–740
DOI 10.1007/s00453-013-9759-2

Inclusion/Exclusion Meets Measure and Conquer

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

Received: 20 July 2012 / Accepted: 13 February 2013 / Published online: 27 February 2013

Abstract Inclusion/exclusion and measure and conquer are two central techniques from the field of exact exponential-time algorithms that recently received a lot of attention. In this paper, we show that both techniques can be used in a single algorithm. This is done by looking at the principle of inclusion/exclusion as a branching rule. This inclusion/exclusion-based branching rule can be combined in a branch-and-reduce algorithm with traditional branching rules and reduction rules. The resulting algorithms can be analysed using measure and conquer allowing us to obtain good upper bounds on their running times.

In this way, we obtain the currently fastest exact exponential-time algorithms for a number of domination problems in graphs. Among these are faster polynomial-space

Preliminary parts of this paper have appeared in the conference proceedings of three different conferences. Preliminary parts appeared under the title 'Inclusion/Exclusion Meets Measure and Conquer: Exact algorithms for counting dominating sets' at the 17th Annual European Symposium on Algorithms (ESA 2009), Lecture Notes in Computer Science 5757, pp. 554–565, under the title 'Polynomial Space Algorithms for Counting Dominating Sets and the Domatic Number' at the 7th International Conference on Algorithms and Complexity (CIAC 2010), Lecture Notes in Computer Science 6078, pp. 73–84, and under the title 'Inclusion/Exclusion Branching for Partial Dominating Set and Set Splitting' at the 5th International Symposium on Parameterized and Exact Computation (IPEC 2011), Lecture Notes in Computer Science 6478, pp. 204–215. The last paper received the best student paper award of IPEC 2011.

J. Nederlof (✉)
Department of Information and Computing Sciences, Utrecht University, Princetonplein 5, 3584 CC Utrecht, The Netherlands
e-mail: j.nederlof@uu.nl

J.M.M. van Rooij
Consultants in Quantitative Methods, Vonderweg 16, 5616 RM Eindhoven, The Netherlands
e-mail: jmmrooij@cs.uu.nl

T.C. van Dijk
Lehrstuhl für Informatik I, Universität Würzburg, Am Hubland, 97074 Würzburg, Germany
e-mail: thomas.dijk@uni-wuerzburg.de

and exponential-space algorithms for #Dominating Set and Minimum Weight Dominating Set (for the case where the set of possible weight sums is polynomially bounded), and a faster polynomial-space algorithm for Domatic Number.

This approach is also extended in this paper to the setting where not all requirements in a problem need to be satisfied. This results in faster polynomial-space and exponential-space algorithms for Partial Dominating Set, and faster polynomial-space and exponential-space algorithms for the well-studied parameterised problem k-Set Splitting and its generalisation k-Not-All-Equal Satisfiability.

Keywords Exact exponential algorithm · Dominating set · NP-hard · Branching · Inclusion/Exclusion

1 Introduction

One could argue that the two most important recent new developments in the field of exact exponential-time algorithms are the use of *inclusion/exclusion* by Björklund et al. [8, 13] and *measure and conquer* by Fomin et al. [33]. Although the use of inclusion/exclusion for exact exponential-time algorithms was introduced for the Travelling Salesman Problem by Kohn et al. [51], Karp [48], and Bax [4] much earlier, Björklund et al. popularised it only recently by using it to solve many covering and partitioning problems [8, 13]. The approach has seen many recent new applications, e.g., see [2, 6–13, 58]. Measure and conquer is a new general approach introduced by Fomin et al. to obtain good upper bounds on the running time of branch-and-reduce algorithms that builds on earlier work by Eppstein [26]. Since its introduction it has become the standard approach to analyse branch-and-reduce algorithms, which can be seen from many recent publications [5, 16, 17, 19, 29–34, 40, 42, 43, 66, 67].

In this paper, we will show that both techniques can be used in one single algorithm. Similar to Bax [4], we observe that the principle of inclusion/exclusion can be interpreted as branching. In this way, we can combine traditional branching rules with inclusion/exclusion-based branching rules, and we can analyse such an algorithm using measure and conquer. We use this approach to obtain the currently fastest exact exponential-time algorithms for a number of domination problems in graph. Among these results are an $\mathcal{O}(1.5673^n)$-time polynomial-space algorithm and an $\mathcal{O}(1.5002^n)$-time-and-space algorithm for #Dominating Set and for Minimum Weight Dominating Set for the case where the set of possible weight sums is polynomially bounded (both with further improvements when restricted to some graph classes), and an $\mathcal{O}(2.7139^n)$-time polynomial-space algorithm for Domatic Number.

We also extend our approach to problems where not all the requirements to which we would apply the inclusion/exclusion-based branching rule must be satisfied. This results in an extended inclusion/exclusion-based branching rule that we use to obtain an $\mathcal{O}(1.5673^n)$-time polynomial-space algorithm and an $\mathcal{O}(1.5014^n)$-time-and-space algorithm for Partial Dominating Set. Our perhaps most impressive results are obtained when we apply extended inclusion/exclusion-based branching after the recent kernel for k-SET SPLITTING and obtain an $\mathcal{O}^*(1.8213^k)$-time polynomial-space algorithm and an $\mathcal{O}^*(1.7171^k)$-time-and-space algorithm for this well-studied parameterised problem, where k denotes the number of sets to be split. These two results can be extended to algorithms that solve the more general problem k-NOT-ALL-EQUAL SATISFIABILITY within the same running times.

1.1 Overview of Results and Comparison to Previous Results

DOMINATING SET is a problem that has received a lot of recent attention in the study of exact exponential-time algorithms. While the first algorithm improving the trivial $\mathcal{O}^*(2^n)$-time bound appeared not before 2004 [37, 45, 63], the introduction of measure and conquer allowed a big improvement to take place resulting in an $\mathcal{O}(1.5260^n)$-time polynomial-space algorithm and an $\mathcal{O}(1.5137^n)$-time-and-space algorithm [33]. In a number of steps, this was improved to $\mathcal{O}(1.4969^n)$ time and polynomial space by van Rooij and Bodlaender [66]. The currently fastest algorithms for DOMINATING SET are given by very recent results by Iwata and run in $\mathcal{O}(1.4864^n)$ time and polynomial space or $\mathcal{O}(1.4689^n)$ time and space [47].

For #DOMINATING SET, no faster polynomial-space algorithms are known to us than the algorithm that enumerates all minimal dominating sets, which currently requires $\mathcal{O}(1.7159^n)$ time [34]. Using exponential space, Fomin et al. have shown that this problem can be solved in $\mathcal{O}(1.5535^n)$ time and space [31]. In this paper, we improve these results to $\mathcal{O}(1.5673^n)$ time and polynomial space or to $\mathcal{O}(1.5002^n)$ time and space. Furthermore, we extend our results to some graph classes as studied by Gaspers et al. [41]: they consider DOMINATING SET restricted to c-dense graphs, circle graphs, chordal graphs, 4-chordal graphs, and weakly chordal graphs. We improve their results and extend them to the more general problem #DOMINATING SET; see Table 1. Note that some of our algorithms are slower than the recent algorithm for DOMINATING SET by Iwata [47].

For MINIMUM WEIGHT DOMINATING SET, the combinatorial bound of enumerating all minimal dominating sets was improved to an $\mathcal{O}(1.5780^n)$-time algorithm by Fomin et al. [35]. When restricted to using weights with the property that the set of possible weight sums is polynomially bounded, Fomin et al. have also given an

Table 1 Results for DOMINATING SET by Gaspers et al. [41] on five graph classes compared to our results for #DOMINATING SET

Graph class	Results of Gaspers et al. [41]	Our results
c-dense graphs	$\mathcal{O}(1.5063^{(\frac{1}{2}+\frac{1}{2}\sqrt{1-2c})n})$	$\mathcal{O}(1.5002^{(\frac{1}{4}+\frac{1}{4}\sqrt{9-16c})n})$
chordal graphs	$\mathcal{O}(1.4124^n)$	$\mathcal{O}(1.3687^n)$
weakly chordal graphs	$\mathcal{O}(1.4776^n)$	$\mathcal{O}(1.4590^n)$
4-chordal graphs	$\mathcal{O}(1.4845^n)$	$\mathcal{O}(1.4700^n)$
circle graphs	$\mathcal{O}(1.4887^n)$	$\mathcal{O}(1.4764^n)$

$\mathcal{O}(1.5535^n)$-time-and-space algorithm [31]. Similar to #DOMINATING SET, we give algorithms that solve MINIMUM WEIGHT DOMINATING SET for the case where the set of possible weight sums is polynomially bounded in $\mathcal{O}(1.5673^n)$ time and polynomial space or to $\mathcal{O}(1.5002^n)$ time and space.

The first exponential-time algorithm related to DOMATIC NUMBER is due to Reige and Rothe: they gave an $\mathcal{O}(2.9416^n)$-time and polynomial-space algorithm to decide whether the domatic number of a graph is at least three [61] (this problem is also known as 3-DOMATIC NUMBER). Later, Fomin et al. gave an $\mathcal{O}(2.8718^n)$-time and exponential-space algorithm for DOMATIC NUMBER in [34]. Using the set partitioning via inclusion/exclusion framework, Björklund et al. have improved this to $\mathcal{O}^*(2^n)$ time and space [13], and later to $\mathcal{O}^*(2^n)$ time and $\mathcal{O}(1.7159^n)$ space [12]. Using only polynomial space, they also gave an $\mathcal{O}(2.8718^n)$-time algorithm that uses the minimal dominating set enumeration procedure of Fomin et al. [34]. Finally, Reige et al. show that the 3-DOMATIC NUMBER can be computed in $\mathcal{O}(2.695^n)$ time and polynomial space [62]. In this paper, we improve previous results using polynomial space by giving an $\mathcal{O}(2.7139^n)$-time polynomial-space algorithm. Note that this running time nears that of the best known polynomial space result on the special case of 3-DOMATIC NUMBER.

For PARTIAL DOMINATING SET, the only previous algorithm beating the trivial $\mathcal{O}^*(2^n)$-time bound was given by Liedloff in his PhD thesis and runs in $\mathcal{O}(1.6183^n)$ time and polynomial space [53]. We improve this result to $\mathcal{O}(1.5673^n)$ time and polynomial space or to $\mathcal{O}(1.5014^n)$ time and space. We note that while the parameterised version of DOMINATING SET is $\mathcal{W}[2]$-complete [25], the parameterised version of PARTIAL DOMINATING SET parameterised by the number of vertices that need to be dominated t is Fixed-Parameter Tractable as shown by Kneis et al. [50]. The currently fastest parameterised algorithm is due to Koutis and Williams and runs in $\mathcal{O}^*(2^t)$ time [52]. When parameterised by the size of the partial dominating set and restricted to planar graphs, Amini et al. have shown that the problem is Fixed-Parameter Tractable [1], and Fomin et al. have given a subexponential-time algorithm [38].

A well-studied parameterised problem for which we give faster parameterised algorithms is k-SET SPLITTING. Both the parameterised and the unparameterised versions of this problem have been studied extensively, both combinatorially and algorithmically; see [3, 20, 23, 24, 27, 28, 54–56, 60, 70, 71]. There exists a sequence of algorithms for k-SET SPLITTING that each improve upon previous publications, see Table 2. Recently, an efficient kernel for this problem has been found by Lokshtanov and Saurabh [54]. This kernel has been used to obtain the previously fastest algorithms for this problem, namely an $\mathcal{O}^*(2^k)$-time polynomial-space algorithm and an $\mathcal{O}^*(1.9630^k)$-time-and-space algorithm. We will use the same kernel and apply an algorithm based on our new extended inclusion/exclusion-based branching rule to obtain an $\mathcal{O}^*(1.8213^k)$-time polynomial-space algorithm and an $\mathcal{O}^*(1.7171^k)$-time-and-space algorithm. We note that our results for k-SET SPLITTING can be extended to solve the more general problem k-NOT-ALL-EQUAL SATISFIABILITY within the same running time.

Table 2 List of known results for k-SET SPLITTING

Authors		Time	Space usage
Dehne, Fellows, Rosamond	[23]	$\mathcal{O}^*(72^k)$	polynomial
Dehne, Fellows, Rosamond, Shaw	[24]	$\mathcal{O}^*(8^k)$	polynomial
Lokshtanov and Sloper	[55]	$\mathcal{O}^*(2.6499^k)$	polynomial
Chen and Lu (randomized algorithm)	[20]	$\mathcal{O}^*(2^k)$	polynomial
Lokshtanov and Saurabh	[54]	$\mathcal{O}^*(2^k)$	polynomial
Lokshtanov and Saurabh	[54]	$\mathcal{O}^*(1.9630^k)$	exponential
This paper		$\mathcal{O}^*(1.8213^k)$	polynomial
This paper		$\mathcal{O}^*(1.7171^k)$	exponential

1.2 Technical Novelties

This paper contains a number of technical contributions. Most importantly, we revisit the inclusion/exclusion-based branching as first used by Bax [4] and observe that this way of looking at the principle of inclusion/exclusion can be used in a standard branch-and-reduce algorithm together with traditional branching and reduction rules. As a result, such an algorithm can be analysed by techniques such as measure and conquer [33]. This is the basic tool used to obtain most of our results.

Secondly, we show that the inclusion/exclusion-based branching rule can be extended to the more general setting where not all requirements of a problem need to be satisfied, but only a given number of them. This resulted in our extended inclusion/exclusion-based branching rule. We use this branching rule to obtain our results for PARTIAL DOMINATING SET, k-SET SPLITTING, and k-NOT-ALL-EQUAL SATISFIABILITY.

Finally, we consider combining branch-and-reduce algorithms with treewidth-based dynamic procedures to solve sparse instance. This combination has been used before. Early examples include results by Kneis et al. for MAXIMUM CUT, MAXIMUM 2-SATISFIABILITY, MAXIMUM EXACT 2-SATISFIABILITY and DOMINATING SET on cubic graphs [49], and by Fomin et al. for k-EDGE DOMINATING SET and #DOMINATING SET [31]. The results by Gaspers et al. [41] for DOMINATING SET on some graph classes can also be considered as such a result.

We introduce a combination of branching with a novel treewidth-based approach that, in contrast to the work by Fomin et al. [31], requires only polynomial space while being much simpler than the approach of Kneis et al. [49]. Our approach is based on an annotation procedure that allows us to deal with sparse instances in polynomial space with reasonable efficiency. This procedure acts as a set of reduction rules while it does not remove anything from the instance: it just annotates parts of it. When the unannotated part of the instance is simple enough, we remove the annotations. Because of the specific way used to annotate vertices, we can prove that the resulting instance has treewidth at most two, i.e., it is a generalised series-parallel graph. On such graphs, the problem can be solved in polynomial time. A nice property of our annotation procedure is that it can straightforwardly combined with the exponential-space approach by Fomin et al. [31]. Many of our exponential-space results rely on this combination.

1.3 Paper Organisation

This paper is organised as follows. Section 2 gives some preliminaries and problem definitions, including a discussion on measure and conquer and an overview of known results related to treewidth that we need in this paper. This is followed by the introduction of the inclusion/exclusion-based branching rule and the extended inclusion/exclusion-based branching rule in Sect. 3. In Sect. 4, we give our polynomial-space results for domination problems in graphs. Hereafter, we give our polynomial-space results for the considered parameterised problems in Sect. 5. Then, we improve most of our results at the cost of using exponential-space in Sect. 6. Finally, we give some additional results including a slightly faster algorithm for #DOMINATING SET on general graphs and the results for #DOMINATING SET on some graph classes in Sect. 7. We conclude by some concluding remarks in Sect. 8.

2 Preliminaries

Let $G = (V, E)$ be an n-vertex undirected, simple graph. The open neighbourhood of a vertex $v \in V$ is denoted by $N(v) = \{u \in V \mid \{u, v\} \in E\}$, and its closed neighbourhood by $N[v] = N(v) \cup \{v\}$. The degree of a vertex $v \in V$ is denoted by $d(v) = |N(v)|$. The set of vertices at distance two from a vertex $v \in V$ is denoted by $N^2(v)$, and the set of vertices at distance at most two from a vertex $v \in V$ is denoted by $N^2[v]$.

For a vertex set $X \subseteq V$, we denote by $G[X]$ the subgraph induced by X, that is, $G[X] = (X, (X \times X) \cap E)$. Also, we denote by $d_X(v)$ the degree of the vertex v in the subgraph induced by X, and by $N_X(v)$, $N_X[v]$ the intersection of the open, respectively closed, neighbourhood of v with X, i.e., $N_X(v) = N(v) \cap X$, $N_X[v] = N[v] \cap X$, $d_X(v) = |N_X(v)|$. The notations $N^2_X[v]$ and $N^2_X(v)$ are defined analogously.

Let $\mathcal{S}$ be a collection of sets over a universe $\mathcal{U}$ (the pair $(\mathcal{S}, \mathcal{U})$ is often called a set system or a hypergraph (denoted as $(\mathcal{U}, \mathcal{S})$)). For a set $S \in \mathcal{S}$, the size, or cardinality, of S is the number of elements in S, which is denoted by $|S|$. The frequency of an element $e \in \mathcal{U}$ is the number of sets in which e occurs. The *incidence graph* of a set system $(\mathcal{S}, \mathcal{U})$ is the bipartite graph $G = (\mathcal{S} \cup \mathcal{U}, E)$ with a vertex for each $S \in \mathcal{S}$ and each $e \in \mathcal{U}$ and an edge between a vertex representing a set S and a vertex representing an element e if and only if $e \in S$, i.e., $\{e, S\} \in E$ if and only if $e \in S$. A red-blue graph $G = (\mathcal{R} \cup \mathcal{B}, E)$ is a bipartite graph with red vertex set $\mathcal{R}$ and blue vertex set $\mathcal{B}$.

We use the $\mathcal{O}^*$-notation, which is similar to the usual $\mathcal{O}$-notation but suppresses all factors polynomially bounded in the input size. For example, if n and k are at most the input size then $\mathcal{O}(n^2k2^k)$ is $\mathcal{O}^*(2^k)$.

In this paper we will often refer to the Inclusion/Exclusion technique that allows to express sizes of intersections of sets in sizes of unions of sets. Since knowledge of the principle itself is not required in this paper, we will only briefly summarise it here by the following theorem that can be found in many textbooks on discrete mathematics.

Theorem 1 (Folklore) *Let U be a set and $A_1, \dots, A_n \subseteq U$. With the convention $\bigcap_{i \in \emptyset} \overline{A_i} = U$, the following holds*:

$$\left| \bigcap_{i \in \{1,\dots,n\}} A_i \right| = \sum_{X \subseteq \{1,\dots,n\}} (-1)^{|X|} \left| \bigcap_{i \in X} \overline{A_i} \right| \tag{2.1}$$

2.1 Problem Definitions

Given a collection of sets $\mathcal{S}$ over a universe $\mathcal{U}$, a *set cover* of $(\mathcal{S}, \mathcal{U})$ is a collection of sets $\mathcal{C} \subseteq \mathcal{S}$ such that $\bigcup_{S \in \mathcal{C}} S = \mathcal{U}$. In the SET COVER Problem, we are given $(\mathcal{S}, \mathcal{U})$ and are asked to compute (the size of) a minimum cardinality set cover.

A *dominating set* is a set of vertices $D \subseteq V$ such that every vertex $v \in V$ is either in D or adjacent to some vertex in D, i.e., D is a dominating set if and only if $V = \{v \in V \mid \exists u \in D : v \in N[u]\}$. A dominating set D is called a minimum dominating set if it is of minimum cardinality among all dominating sets in G. The DOMINATING SET problem is the computational problem of computing (the size of) a minimum dominating set:

DOMINATING SET
Input: A graph $G = (V, E)$, and an integer $k \in \mathbb{N}$.
Question: Does there exist a dominating set $D \subseteq V$ in G of size at most k?

We often denote by #PROBLEM the counting problem that asks to compute the number of solutions to the problem PROBLEM. E.g., the #DOMINATING SET problem is the problem of computing the number of dominating sets, and the #SET COVERproblem is the problem of computing the number of set covers. In this paper, we often solve more general problems and count the number of dominating sets (or set covers) of size κ, for each value $0 \le \kappa \le n$, to solve #DOMINATING SET (or #SET COVER).

Any vertex set $D \subseteq V$ is a *partial dominating set*, and such a partial dominating set D dominates all vertices that have a neighbour in D.

PARTIAL DOMINATING SET
Input: A graph $G = (V, E)$, an integer $t \in \mathbb{N}$, and an integer $k \in \mathbb{N}$.
Question: Does there exist a partial dominating set $D \subseteq V$ in G of size at most k that dominates at least t vertices?

Given a red-blue graph $G = (\mathcal{R} \cup \mathcal{B}, E)$, a red-blue dominating set is a subset of the red vertices $D \subseteq \mathcal{R}$ that dominates all blue vertices, i.e., $\mathcal{B} = \{b \in \mathcal{B} \mid \exists_{r \in \mathcal{R}}\, b \in N(r)\}$. The RED-BLUE DOMINATING SET problem asks to compute (the size of) a minimum cardinality red-blue dominating set. Partial red-blue dominating sets are defined analogously. Notice that the requirement that a red-blue graph is bipartite plays no role in this problem, as the only edges of interest for the domination requirement are the edges between a red vertex and a blue vertex.

We also consider weighted versions of domination problems. Given a graph $G = (V, E)$ and a weight function $\omega : V \to \mathbb{R}_+$, a *minimum weight dominating set* is a dominating set D in G with minimal total vertex weight $\sum_{v\in D}\omega(v)$. The MINIMUM WEIGHT DOMINATING SET problem asks to compute (the total weight of) a minimum weight dominanting set. When considering weighted problems, we often require that the set of possible weight sums is polynomially bounded, that is, we require that $|\{\sigma \in \mathbb{R}_+ \mid \exists_{X\subseteq V}\ \sigma = \sum_{v\in X}\omega(v)\}| \le p(n)$ for some polynomial $p(n)$.

When solving the above domination problem, the following transformations are often used, e.g., see [31, 33, 35, 46, 47, 66]. From these transformation, we will often use the one from (partial) dominating sets to (partial) red-blue dominating sets.

Proposition 2 *Let $H = (V, F)$ be a graph. Consider the set system $(\mathcal{S}, \mathcal{U})$ defined through $\mathcal{U} = V$ and $\mathcal{S} = \{N[v] \mid v \in V\}$, and consider the red-blue graph $G = (\mathcal{R} \cup \mathcal{B}, E)$ defined as the incidence graph of $(\mathcal{S}, \mathcal{U})$ with red vertex set $\mathcal{R} = \mathcal{S}$ and blue vertex set $\mathcal{B} = \mathcal{U}$. There exists size-preserving one-to-one correspondences between the following associated sets of items*:

1. *The (partial) dominating sets in a graph H*;
2. *The (partial) set covers of the set system $(\mathcal{S}, \mathcal{U})$*;
3. *The (partial) red-blue dominating sets in G.*

Proof (1 and 2.) It is easy to see that, for every vertex $v \in V$, the set $S = N[v]$ in $\mathcal{S}$ corresponds to the ability of v to dominate other vertices, and that the element v in $\mathcal{U}$ corresponds to the need of v to be dominated. The mapping $v \leftrightarrow N[v]$ gives a bijection between the vertices in V and sets in $\mathcal{S}$. This mapping can straightforwardly be extended to a mapping between the $2^{|V|}$ vertex subsets and the $2^{|\mathcal{S}|}$ collections of sets. The required size-preserving one-to-one correspondence is the restriction of this last mapping to the dominating sets in G and set covers of $(\mathcal{S}, \mathcal{U})$.

(2 and 3.) The vertices of G corresponding to the sets in a set cover of $(\mathcal{S}, \mathcal{U})$ form a red-blue dominating set in G. Similary, the sets corresponding to the vertices of a red blue dominating set in G from a set cover of $(\mathcal{S}, \mathcal{U})$. This defines the one-to-one correspondence. □

A *domatic k-partition* of a graph G is a partition $V_1, V_2, \dots, V_k$ of the vertices V such that each V_i is a dominating set in G. The *domatic number* of G is the largest $k \in \mathbb{N}$ for which there exists a domatic k-partition, that is, it is the maximum number of disjoint dominating sets in G.

DOMATIC NUMBER
Input: A graph $G = (V, E)$ and an integer k.
Question: Can G be partitioned into at least k dominating sets?

Every graph has domatic number at least one, since the set of all vertices V is a dominating set in G. Every graph that has no isolated vertices has domatic number of at least two. This follows from the following simple observation: take any maximal independent set I in G (such a set must clearly exist); now, I is a dominating set because I is maximal, and $V \setminus I$ is a dominating set because I is an independent set and there are no isolated vertices. Deciding whether the domatic number of G is at least k, for $k \ge 3$, is $\mathcal{NP}$-hard [39].

A parameterised problem is a computational problem in which the input has two parts: the first part is the instance x, and the second part is the parameter k. In the study of parameterised algorithms, one analyses algorithms using two complexity parameters: the problem parameter k and a parameter proportional to the size of the instance x. In this paper, we propose algorithms for two parameterised problems: k-SET SPLITTING and k-NOT-ALL-EQUAL SATISFIABILITY.

Consider a collection of sets $\mathcal{S}$ over a universe $\mathcal{U}$. In the k-SET SPLITTING problem, we are asked to divide the elements of $\mathcal{U}$ into two colour classes: red and green. In this setting, a set $S \in \mathcal{S}$ is said to be *split* if S contains at least one element of each of the two colour classes, i.e., at least one red element and at least one green element.

***k*-SET SPLITTING**

Input:	A collection of sets $\mathcal{S}$ over a universe $\mathcal{U}$.
Parameter:	An integer $k \in \mathbb{N}$.
Question:	Can $\mathcal{U}$ be partitioned into two colour classes such that at least k sets from $\mathcal{S}$ are split?

This problem is also known as k-MAXIMUM HYPERGRAPH 2-COLOURING.

We also consider the generalisation of this problem that is known as k-NOT-ALL-EQUAL SATISFIABILITY. In this problem, we are given a set of clauses C containing literals of variables from a set X and are asked whether there exists a truth assignment to the variables in X such that at least k clauses in C contain a literal set to true and a literal set to false? Notice that this problem extends k-SET SPLITTING if one identifies clauses with sets and variables with elements. The difference is that variables can occur both as positive literals and as negative literals in the clauses of a k-NOT-ALL-EQUAL SATISFIABILITY instance while elements do not have a sign in a k-SET SPLITTING instance.

2.2 Measure and Conquer

A *branch-and-reduce* algorithm (or branching algorithm) is a recursive divide-and-conquer algorithm that consists of a series of reduction rules and branching rules. Reduction rules are rules that can be applied in polynomial time to simplify, or even solve, instances with specific properties. Reduction rules that immediately solve the instance without further recursion are also called halting rules. The branching rules are rules that solve an instance by recursively generating a series of smaller instances of the same problem. This is done such that a solution to the original problem instance can be constructed from the solutions to the generated instances. The tree generated by the structure of recursive calls is called the *branching tree* of the algorithm, and individual recursive calls made by the algorithm are called branches of the algorithm.

To analyse the running time of a branch-and-reduce algorithm, one tries to find a bound on the size of the branching tree, i.e., a bound on the number of subproblems that are generated. This is because the branching tree generally has exponential size, which dominates the running time because all rules are executed in polynomial time. In general, this is done using a set of recurrence relations of the following form:

$$N(n) \leq N(n - \Delta n_1) + N(n - \Delta n_2) + \cdots + N(n - \Delta n_r)$$

Here, $N(n)$ is the number of subproblems generated for an input of size n (e.g., consisting of n vertices), and the Δn_i represent the reduction in problem size for the i-th subproblem generated by a branching rule. In the recurrence relation, we omit a '+1' to count the subproblem that is branched on because this would only effect the constant factor in the running time. Using $N(n) = 1$ for every $n \leq 0$, we can use standard techniques to find an upper bound on $N(n)$ of the form $N(n) \leq c^n$, where c is the largest positive real root of the set of equations $1 = \sum_{i=1}^{r} c^{-n_i}$. As a result, we know that our algorithm runs in $\mathcal{O}^*(c^n)$ time. For a general treatment of branch-and-reduce algorithms and their analysis, see [36] or [65].

Often, better bounds on the running time of a branch-and-reduce algorithm can be obtained by using *Measure and Conquer* [33]. In a measure-and-conquer analysis, the branching tree is analysed using a non-standard measure of instance size. For example, one can use the following measure k on the size of a red-blue graph $G = (\mathcal{R} \cup \mathcal{B}, E)$:

$$k := \sum_{b \in \mathcal{B}} v(d(b)) + \sum_{r \in \mathcal{R}} w(d(r))$$

Here, $v, w : \mathbb{N} \to \mathbb{R}_+$ are weight functions assigning a measure of $v(d(b))$ to a blue vertex b of degree $d(b)$ and a measure of $w(d(r))$ to a red vertex r of degree $d(r)$.

Such a measure can be used to gain upper bounds on the size of a branching tree in exactly the same way as described above, that is, by formulating and solving a set of recurrence relations. This set of recurrence relations is, in general, much larger than for a standard analysis because different local configurations (in the example: configurations involving different vertex degrees) lead to different recurrences. For an analysis to be correct, the measure must have the property that no subproblem that is generated by either a reduction rule or a branching rule has a higher measure than the original problem, i.e., the measure may never increase. Often this is guaranteed by enforcing constraints on the weight functions being used.

What we have obtained in this way, is an analysis that gives us an upper bound on the running time of the algorithm that depends on the chosen weight functions. One wants to choose these weight functions in such a way that the resulting running time is minimised. This results in a mathematical optimisation problem. Such a problem can be made finite by using a finite number of different weights. In general this is done by only making the weights corresponding to low vertex degrees variable; in the example this can be done by setting $v(i) = v(p)$ and $w(i) = w(p)$ for all $i \geq p$ for some $p \in \mathbb{N}$. The resulting finite problem can be solved numerically by a computer.

More on measure and conquer can be found in [33, 36]. For information on how to solve the numerical optimisation problems resulting from measure-and-conquer analyses, see [26, 43, 65]. For the technical details of the solver that we used, see [65].

2.3 Treewidth

The branch-and-reduce algorithms in the paper often use tree-decomposition-based dynamic programming methods to solve instances that have been simplified significantly by branching.

Definition 3 (Tree Decomposition) A *tree decomposition* of a graph $G = (V, E)$ consists of a tree T in which each node $x \in T$ has an associated set of vertices $X_x \subseteq V$ (called a *bag*) such that $\bigcup_{x \in T} X_x = V$ and the following properties hold:

1. For each $\{u, v\} \in E$, there exists an X_x such that $\{u, v\} \subseteq X_x$.
2. If $v \in X_x$ and $v \in X_y$, then $v \in X_z$ for all nodes z on the path from node x to node y in T.

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

We sometimes also use *path decompositions*. A path decomposition is a tree decomposition in which the tree T is a path, i.e., a tree in which all nodes have degree at most two. The pathwidth $\mathrm{pw}(G)$ of G is the minimum width over all possible path decompositions of G.

Given a tree or path decomposition of a graph G, we can effectively solve the problems we study using dynamic programming on tree decompositions. This is formalised in the following results by van Rooij et al. [68]. For a general treatment of tree decompositions and dynamic programming on tree decompositions, see for example [15].

Proposition 4 [68] *Given a tree decomposition of a red-blue graph $G = (\mathcal{R} \cup \mathcal{B}, E)$ of width tw, the number of partial red-blue dominating sets of size κ dominating exactly t vertices, for each combination of values $0 \leq \kappa \leq |\mathcal{R}|$, $0 \leq t \leq |\mathcal{B}|$, can be computed in $\mathcal{O}^*(2^{tw})$ time.*

Proposition 5 [68] *Given a tree decomposition a graph $G = (V, E)$ of width tw, the number of dominating sets of size κ, for each value $0 \leq \kappa \leq n$, can be computed in $\mathcal{O}^*(3^{tw})$ time.*

We can apply the above propositions if we have a tree decomposition of a graph of small width. To obtain these, we will use two different results.

For our polynomial-space algorithms, we use the following well-known fact on the treewidth of a graph, e.g., see [14], combined with the easy to check fact that a collection of disjoint cycles has treewidth at most two.

Proposition 6 *Let G be a graph of treewidth at least two. The following operations do not increase the treewidth of G:*

- *duplicating an edge,*
- *adding a vertex of degree one,*
- *subdividing an edge,*
- *contracting an edge incident to a vertex of degree two.*

For our exponential-space algorithms, we use the following result by Fomin et al. [31].

Proposition 7 [31] *For any $\epsilon > 0$, there exists an integer n_ϵ such that if G is an n-vertex graph with $n > n_\epsilon$ then*

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

where n_i is the number of vertices of degree i in G, for any $i \in \{3, 4, 5, 6\}$, and $n_{\ge 7}$ is the number of vertices of degree at least 7. Moreover, a path decomposition of the corresponding width can be constructed in polynomial time.

3 Inclusion/Exclusion-Based Branching

We will begin by showing that one can look at inclusion/exclusion from a branching perspective, as observed by Bax [4]. We do so by considering branching rules for the Set Cover and #Set Coverproblems; branching rules that easily generalise to other problems as well. In this way, we will see that inclusion/exclusion-based branching can be used to branch on elements in #Set Coverinstances in a way very similar to how one would normally branch on sets.

The canonical branching rule for Set Cover is branching on a set. Sets represent an *optional* property of the instance: either a set is in a solution, or it is not. If we branch on such an optional property of the instance, we obtain two branches in which the problem is simplified. In one branch, we *discard* the set decreasing the number of sets. In the other branch, we *take* the set in the solution decreasing the number of sets, and, since this set covers all its elements, its elements can also be removed from the instance, decreasing the number of elements as well. Any solution to the Set Cover instance is either returned by the discard branch, or returned by the take branch after we add the set on which we branched to it.

The counting problem #Set Covercan also be handled by branching steps of this type because the total number of solutions is the sum of the number of solutions obtained from each branch. Branching on a set (or any other *optional property*) can be thought of as adding up the number of solutions where it is *required* to take the set (required to have the property) and the number of solutions where it is *forbidden* to take the set (forbidden to have the property):

$$\textsc{Optional} = \textsc{Required} + \textsc{Forbidden}$$

When counting set covers of a fixed size κ, then we should count set covers of size $\kappa - 1$ in the required branch, as we have taken the set in the solutions that we count here. In the forbidden branch, we do not need to decrease κ.

We now consider branching on an element. Such a branching step is unusual and may appear strange at first sight as elements are not optional. Elements represent a requirement in the problem instance (a *required property*): the requirement to cover the element. However, when counting the number of solutions, we can rearrange the above formula to obtain the following branching rule that we call *inclusion/exclusion-based branching* or simply *IE-branching*:

$$\textsc{Required} = \textsc{Optional} - \textsc{Forbidden}$$

For a #SET COVERinstance, this formula expresses that the number of ways to cover a certain element is equal to the number of ways to *optionally* cover it, i.e., in which we are indifferent about covering it, minus the number of ways in which it is *forbidden* to cover it. This is interesting because this branching rule also simplifies the instance in both branches. In the branch where we make it *optional* to cover the element, we can remove the element from the instance: this removes an element and reduces the size of the sets in which it occurs. In the branch where we make it *forbidden* to cover the element, we have to remove the element and every set in which the element occurs. This leads to an even greater reduction in the size of the instance. When counting set covers of a fixed size κ, then we continue counting set covers of size κ in both branches since we do not select any set to be in the cover in either branch.

Having introduced inclusion/exclusion-based branching, we will show how this branching rule relates to other algorithms in the literature that use the principle of inclusion/exclusion. These algorithms typically use a variant of the following formula, which is often called the *inclusion/exclusion formula* (e.g., see [7, 8, 10, 12, 13, 48, 51, 58]):

$$c_\kappa = \sum_{X \subseteq \mathcal{U}} (-1)^{|X|} a(X)^\kappa \tag{3.1}$$

In this formula, c_κ is the number of ways in which we can pick κ sets from $\mathcal{S}$ such that all elements in $\mathcal{U}$ are covered by these sets (possibly picking sets multiple times), and $a(X)$, for any subset $X \subseteq \mathcal{U}$, is the number of sets in $\mathcal{S}$ that do not contain any element from X.

Consider a branching algorithm without reduction rules and without employing branch and bound. If the branching rules of the algorithm are based on an optional property of the problem, as is typically the case, the algorithm uses exhaustive search. Similarly, if the branching rules of the algorithm are based on a required property of the problem, then the algorithm uses the inclusion/exclusion formula.

To see this, consider a #SET COVERinstance $(\mathcal{S}, \mathcal{U})$. Let c'_κ be the number of set covers of size κ. Consider the branching tree constructed by exhaustively performing inclusion/exclusion-based branching. In the subproblems corresponding to the leafs of this branching tree, each element is either optional or forbidden. For each leaf of this tree, we consider the contribution of this leaf to the number computed in the root of the tree. Let X be the set of forbidden elements in a leaf, and notice that the $2^{|\mathcal{U}|}$ leaves of the tree represent the $2^{|\mathcal{U}|}$ different subsets $X \subseteq \mathcal{U}$. The number of set covers of size κ for the subproblem corresponding to a leaf is $\binom{a(X)}{\kappa}$: the number of ways to pick κ sets from the $a(X)$ available sets, i.e., the number of set covers of size κ where it is optional to cover each element not in X and forbidden to cover any element in X. A minus sign is put for each time we have entered a forbidden branch, so the total contribution of this leaf to the number computed in the root will be $(-1)^{|X|}$ times $\binom{a(X)}{\kappa}$. Summing over all leaves of the branching tree, this gives us the following expression for c'_κ:

$$c'_\kappa = \sum_{X \subseteq \mathcal{U}} (-1)^{|X|} \binom{a(X)}{\kappa} \tag{3.2}$$

We note that the difference between Eqs. (3.1) and (3.2) is that Eq. (3.1) counts set covers c_κ where sets may be picked multiple times while we do not allow this in Eq. (3.2).

The advantage of inclusion/exclusion-based branching over using the inclusion/exclusion formula is that we can use reduction rules to improve upon the standard running time of $\mathcal{O}^*(2^{|\mathcal{U}|})$ implied by evaluating every summand of the inclusion/exclusion formula. A standard method of obtaining similar improvements is *trimming*, where one predicts which summands in the inclusion/exclusion formula are non-zero in order to be able to skip the other summands. This method found applications for TRAVELLING SALESMAN PROBLEM and GRAPH COLOURING in graphs of bounded degree; see [11]. In our setting, trimming is equivalent to inclusion/exclusion-based branching in combination with a reduction rule that directly returns zero when it can be predicted that all summands enumerated from the current branch are zero. The main advantage of inclusion/exclusion-based branching over trimming is that inclusion/exclusion-based branching can easily be used interchangeably with traditional branching rules, and that standard methods of analysing such an algorithm, such as measure and conquer, can be applied to it. This will be demonstrated in Sects. 4–6.

3.1 Extended Inclusion/Exclusion-Based Branching

We continue by giving a generalisation of inclusion/exclusion-based branching that applies in the more general setting where not all required properties need to be satisfied, but a given number of them. To do so, we first formalise the above treatment of inclusion/exclusion-based branching.

Let $\mathcal{A}$ be a set containing the objects that we are counting, and let $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n$ be subsets of $\mathcal{A}$; these sets correspond to the properties that we are considering. In the #SET COVERexample, $\mathcal{A}$ equals all possible collections of sets from $\mathcal{S}$, i.e., $\mathcal{A} = 2^{\mathcal{S}}$, and we have a property $\mathcal{P}_S$ for every set $S \in \mathcal{S}$ and a property $\mathcal{P}_e$ for every element $e \in \mathcal{U}$. For a property associated with a set $S \in \mathcal{S}$, we let $\mathcal{P}_S$ contain the collections of sets in $\mathcal{A}$ that contain the set S, i.e., $\mathcal{P}_S = \{\mathcal{C} \in \mathcal{A} \mid S \in \mathcal{C}\}$. For a property associated with an element $e \in \mathcal{U}$, we let $\mathcal{P}_e$ contain the collections of sets in $\mathcal{A}$ that cover e, i.e., $\mathcal{P}_S = \{\mathcal{C} \in \mathcal{A} \mid e \in \bigcup_{S \in \mathcal{C}} S\}$.

Given $\mathcal{A}$ and a series of properties $\mathcal{P}_1, \mathcal{P}_2 \ldots, \mathcal{P}_n$, we let a partitioning (R, O, F) of $\{1, 2, \ldots, n\}$ define whether a property $\mathcal{P}_i$ is a *required property* ($i \in R$), an *optional property* ($i \in O$), or a *forbidden property* ($i \in F$). For such a partitioning (R, O, F), we define (similar to Bax [4]):

$$a(R, O, F) = \left| \left(\bigcap_{i \in R} \mathcal{P}_i \right) \Big\backslash \left(\bigcup_{i \in F} \mathcal{P}_i \right) \right| \tag{3.3}$$

That is, $a(R, O, F)$ counts the number of $\mathcal{C} \in \mathcal{A}$ that have all the required properties ($\mathcal{C} \in \mathcal{P}_i$ is counted when $i \in R$), and none of the forbidden properties ($\mathcal{C} \in \mathcal{P}_i$ is not counted when $i \in F$). Since (R, O, F) partitions $\{1, 2, \ldots, n\}$, all optional properties are ignored, i.e., a $\mathcal{C} \in \mathcal{A}$ is counted both if $\mathcal{C} \in \mathcal{P}_i$ and if $\mathcal{C} \notin \mathcal{P}_i$, for any $i \in O$. Note that in a #SET COVERinstance $(\mathcal{S}, \mathcal{U})$, initially, every set $S \in \mathcal{S}$ corresponds to an optional property since it can either be taken in a solution or not, and that every element $e \in \mathcal{U}$ corresponds to a required property since it must be covered. In this initial case, $a(R, O, F)$ counts the number of set covers of $(\mathcal{S}, \mathcal{U})$.

Through branching, the role of a property can change between being required, optional, and forbidden. It is easy to see that the two branching rules that correspond to branching on an optional or required property correspond to the following formulas:

$$\begin{array}{ll} \text{OPTIONAL:} & a(R, O \cup \{i\}, F) = a(R \cup \{i\}, O, F) + a(R, O, F \cup \{i\}) \\ \text{REQUIRED:} & a(R \cup \{i\}, O, F) = a(R, O \cup \{i\}, F) - a(R, O, F \cup \{i\}) \end{array}$$

We can generalise these two branching rules to the setting where not all required properties need to be satisfied, but only a given number of them. To do so, we define:

$$a_t(R, O, F) = |\{\mathcal{C} \in \mathcal{A} : |R[\mathcal{C}]| = t \wedge |F[\mathcal{C}]| = 0\}| \tag{3.4}$$

where $R[\mathcal{C}] = \{i \in R \mid \mathcal{C} \in \mathcal{P}_i\}$ and $F[\mathcal{C}] = \{i \in F \mid \mathcal{C} \in \mathcal{P}_i\}$.

Any $\mathcal{C} \in \mathcal{A}$ is counted in $a_t(R, O, F)$ if and only if it is counted in $a(R, O, F)$ and satisfies exactly t of the required properties (properties $\mathcal{P}_i$ with $i \in R$). Notice that if we now consider a required property $\mathcal{P}_i$, then it is possible that $\mathcal{C} \in \mathcal{A}$ is counted in $a_t(R, O, F)$ both when $\mathcal{P}_i$ is satisfied and when $\mathcal{P}_i$ is not satisfied.

We can branch on the choice whether a required property (now using the parameter t) will be satisfied or not. This leads to the following recurrence:

$$a_t(R \cup \{i\}, O, F) = a_t(R, O, F \cup \{i\}) + |\{\mathcal{C} \in \mathcal{P}_i : |R[\mathcal{C}]| = t \wedge |F[\mathcal{C}]| = 0\}| \tag{3.5}$$

Note that the second term on the right hand side represents the number of $\mathcal{C} \in \mathcal{A}$ counted in $a_t(R \cup \{i\}, O, F)$ that explicitly satisfy the property $\mathcal{P}_i$.

Since any $\mathcal{C} \in \mathcal{A}$ counted in the second term on the right hand side must be in $\mathcal{P}_i$, we can now apply the standard inclusion/exclusion-based branching rule to the required property $\mathcal{P}_i$. The composition of these two branching rules results in the extended inclusion/exclusion-based branching rule that we will use in this paper:

$$\begin{aligned} a_t(R \cup \{i\}, O, F) = {} & a_t(R, O, F \cup \{i\}) \\ & + (a_{t-1}(R, O \cup \{i\}, F) - a_{t-1}(R, O, F \cup \{i\})) \end{aligned} \tag{3.6}$$

where we set $a_t(R, O, F) = 0$ if $t > |R|$ or $t < 0$. The parameter t is decreased in the last two terms on the right hand side because the subtraction guarantees that the requirement $\mathcal{P}_i$ will be satisfied; therefore, we need to satisfy $t - 1$ remaining requirements from R. In the case that we no longer have the possibility to choose to satisfy a requirement or not, because $t = 0$ or $t = |R|$, then the rule is correct because we set $a_t(R, O, F) = 0$ for any $t < 0$ or $t > |R|$.

At first, the above branching rule does not look particularly efficient since, for each application of the branching rule, three instances have to be solved recursively. However, if we perform some bookkeeping, two recursive calls suffice. Observe that the first and last recursive call differ only in the subscript parameter t. We can exploit this by computing a_s for all $0 \leq s \leq n$ simultaneously whenever we need to compute a_t. This will only slow down the algorithm by a factor n, while it allows us to consider only two subproblems when branching: $a_t(R, O, F \cup \{i\})$ and $a_{t-1}(R, O, F \cup \{i\})$ will be computed in the same branch.

In the same way as we derived the inclusion/exclusion formula from the inclusion/exclusion-based branching rule, we can derive an extension of the inclusion/exclusion formula from the extended inclusion/exclusion formula-based branching rule. If one expands the recurrence that defines the extended inclusion/exclusion-based branching rule (Eq. (3.6)), then the following formula can be obtained for computing $a_t(R, O, F)$:

$$a_t(R, O, F) = \sum_{X \subseteq R} (-1)^{|X|-|R|+t} \binom{|X|}{|R|-t} a_0(\emptyset, O \cup (R \setminus X), F \cup X) \tag{3.7}$$

To see this, consider the branching tree constructed by exhaustively applying the extended inclusion/exclusion-based branching rule. Let X be a set of forbidden properties, i.e., the set of properties that go into the first or third branch (in Eq. (3.6)) when branching. We consider the set of leaves of this tree where X is the set of forbidden properties. For each such leaf, we have lowered the parameter t exactly t times, thus we have taken $|R|-t$ times the first branch and $|X|-|R|+t$ times the second branch. As a result, we have $\binom{|X|}{|R|-t}$ such leaves, and the contribution of each leaf to the sum in the root is multiplied exactly $|X|-|R|+t$ times by -1.

4 Polynomial-Space Algorithms for Domination Problems in Graphs

As a first application of inclusion/exclusion-based branching, we will consider a series of domination problems in graphs. For these problems, we will give exact exponential-time algorithms using polynomial space. We start by giving an algorithm for counting (partial) red-blue dominating sets in Sect. 4.1. This algorithm will then be used to give the currently fastest polynomial-space algorithms for #DOMINATING SET, PARTIAL DOMINATING SET, and some other problems in Sect. 4.2. In Sect. 4.3, we use the same algorithm as a subroutine to give the currently fastest polynomial-space algorithm for DOMATIC NUMBER. Finally, we will look back at our algorithm for counting partial red-blue dominating sets and state a symmetry relation between the red and blue vertices that is of independent interest in Sect. 4.4.

4.1 An Algorithm for Counting (Partial) Red-Blue Dominating Sets

In this section, we give an exponential-time and polynomial-space algorithm that, given a red-blue graph $G = (\mathcal{R} \cup \mathcal{B}, E)$ counts the number of red-blue dominating sets of size κ, for each value of κ with $0 \leq \kappa \leq |\mathcal{R}|$: Algorithm 1. We also give a slight modification of this algorithm that counts the number of partial red-blue dominating sets of size κ that dominate exactly t blue vertices, for each combination of values for κ and t with $0 \leq \kappa \leq |\mathcal{R}|$ and $0 \leq t \leq |\mathcal{B}|$. The algorithms combine the traditional branching approach that branches on red vertices (optional properties) with the inclusion/exclusion-based branching approach introduced in Sect. 3 that branches on blue vertices (required properties). On sparse instances, the algorithms will switch

Algorithm 1 An algorithm for counting the number of red-blue dominating sets of each size κ

```
Input: a red-blue graph $G = (\mathcal{R} \cup \mathcal{B}, E)$ and a set of annotated vertices $A$; initially $A = \emptyset$
Output: a list with the number of red-blue dominating sets in $G$ of size $\kappa$, for each value of $\kappa$
CountRBDS($G, A$):
1:  if there exists a vertex $v \in (\mathcal{R} \cup \mathcal{B}) \setminus A$ of degree at most one in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ then
2:      return CountRBDS($G, A \cup \{v\}$)
3:  else if there exist two vertices $v_1, v_2 \in (\mathcal{R} \cup \mathcal{B}) \setminus A$ both of degree two in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ that have the same two neighbours then
4:      return CountRBDS($G, A \cup \{v_1\}$)
5:  else
6:      Let $r \in \mathcal{R} \setminus A$ be a vertex such that $d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(r)$ is maximal
7:      Let $b \in \mathcal{B} \setminus A$ be a vertex such that $d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(b)$ is maximal
8:      if $d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(r) \leq 2$ and $d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(b) \leq 2$ then
9:          return CountRBDS-DP($G$)
10:     else if $d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(r) > d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(b)$ then
11:         Let $L_{\text{take}}$ = CountRBDS($G[(\mathcal{R} \cup \mathcal{B}) \setminus N[r]], A \setminus N(r)$) shifted down by one
12:         Let $L_{\text{discard}}$ = CountRBDS($[G(\mathcal{R} \cup \mathcal{B}) \setminus \{r\}], A$)
13:         return $L_{\text{take}} + L_{\text{discard}}$
14:     else
15:         Let $L_{\text{optional}}$ = CountRBDS($G[(\mathcal{R} \cup \mathcal{B}) \setminus \{b\}], A$)
16:         Let $L_{\text{forbidden}}$ = CountRBDS($G[(\mathcal{R} \cup \mathcal{B}) \setminus N[b]], A \setminus N(b)$)
17:         return $L_{\text{optional}} - L_{\text{forbidden}}$
```

to a treewidth-based dynamic programming approach. The algorithm will be used to prove the main results of Sects. 4.2 and 4.3.

Algorithm 1 uses a set of annotated vertices A which is initially empty. Intuitively, annotating a vertex corresponds to ignoring the vertex when selecting a vertex to branch on, not only by not considering it for branching, but also by ignoring it as a neighbour for determining the degree of vertices that can be branched on. Annotated vertices, however, cannot simply ignored when the algorithm branches since they still matter for the outcome: here they are treated as ordinary vertices. During the execution of the algorithm, Algorithm 1 annotates any vertex of degree at most one in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$. Furthermore, it annotates a degree two vertex in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ if there exists another degree two vertex in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ with the same two neighbours. These two annotation rules are effectively the reduction rules of the algorithm.

If no vertex can be annotated, Algorithm 1 selects a red vertex and a blue vertex that are both of maximum degree among the vertices of their respective colour in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$. If the degree in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ of any vertex is at most two, then the algorithm switches to a different approach and solves the problem by dynamic programming by calling the procedure CountRBDS(G). This procedure generates a list containing the number of red-blue dominating sets in G of each size κ, $0 \leq \kappa \leq n$, in polynomial time and will be described later. For now, it suffices to say that the annotation procedure guarantees that any red-blue graph G on which CountRBDS(G) is called is simple enough to be dealt with in polynomial time. Otherwise, the maximum degree in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ is at least three. In this case, Algorithm 1 branches on a vertex of maximum degree in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ while it prefers a blue vertex if vertices of maximum degree of both colours exist.

If Algorithm 1 decides to branch on a red vertex $r \in \mathcal{R}$, then it generates two subproblems: one in which it counts red-blue dominating sets that contain r and one in which it counts red-blue dominating sets that do not contain r. This branching corresponds to branching on an optional property, as discussed in Sect. 3. In the first subproblem, r can be removed together with all neighbouring blue vertices since they are now dominated. In the second subproblem, only r can be removed. The algorithm computes the required values by a component-wise summation of the lists returned from both branches. This clearly works if we increase the sizes of the red-blue dominating sets counted in the first branch by one because we have taken the vertex r in these red-blue dominating sets; this is done by the algorithm accordingly (specifically, before we write the list to L_{take} we shift its indices by one).

If Algorithm 1 decides to branch on a blue vertex $b \in \mathcal{B}$, then we perform inclusion/exclusion-based branching; we branch on a required property, as discussed in Sect. 3. The number of red-blue dominating sets in G equals the number of red-blue dominating sets that can either dominate b or not (*optional branch*) minus the number of red-blue dominating sets that do not dominate b (*forbidden branch*). In the first branch, b is removed. In the other branch, b and all red neighbours of b are removed, i.e., $N[b]$ is removed. For each size κ, the algorithm subtracts the result from the second branch from the result of the first branch, correctly computing the number of red-blue dominating sets of each size κ. Again, this is done by component-wise operations on the lists returned from both branches.

Notice that the effects of both branching rules on the red-blue graph G are symmetric. However, we note that this symmetry is not complete because for other purposes red vertices and blue vertices are not symmetric. Examples of the differences between the roles of red vertices and blue vertices can be found in Sect. 7.1 where we reconsider algorithms for #DOMINATING SET and RED-BLUE DOMINATING SET that are allowed to use exponential space.

This concludes the description of the branching of the algorithm. Since the annotation procedure does not affect the output of the Algorithm 1, we can conclude that the algorithm is correct based on the correctness of the branching rules.

We continue by explaining the function of the annotation procedure by giving details on the procedure CountRBDS-DP(G). This procedure can solve remaining problem instances in polynomial time because any red-blue graph G to which it is applied has treewidth at most two. This is formalised by the following lemma that builds upon Proposition 6.

Lemma 8 *When Algorithm 1 makes a call to CountRBDS-DP(G), then the treewidth of G is at most two.*

Proof Let A be the set of annotated vertices maintained by Algorithm 1 when it makes a call to CountRBDS-DP(G). Notice that $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ cannot contain vertices of degree at most one since these would have been annotated by Algorithm 1. Therefore, all vertices in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ are of degree two as $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ is of maximum degree two when the call to CountRBDS-DP(G) is made. Consequently, $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ has treewidth at most two since it is a (possibly empty) collection of cycles.

We will now remove the annotations from the vertices in A in reverse order of their moment of annotation by Algorithm 1. By doing so, each step consists of either adding a vertex of degree at most one to $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$, or adding a degree two vertex v to $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ for which there exist another degree two vertex $u \notin A$ with $N(u) = N(v)$ in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$. We notice that the operation of adding the degree two vertex is identical to first contracting an edge incident to v, then doubling the contracted edge, and then subdividing both copies of the edge again. Hence, both operations do not increase the treewidth of $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ above two by Proposition 6. We conclude that G has treewidth at most two. □

Alternatively, one could say that when Algorithm 1 calls CountRBDS-DP(G), then G is a generalised series-parallel graph.

Corollary 9 *The procedure CountRBDS-DP(G) in Algorithm 1 can be implemented in polynomial time.*

Proof Standard dynamic programming on tree decompositions or generalised series-parallel graphs. For an idea of how such an algorithm works see for example [15, 68]. □

We will not analyse the running time required to solve RED-BLUE DOMINATING SET using Algorithm 1, as the result would not improve the $\mathcal{O}(1.2279^n)$-time and polynomial-space algorithm published in [65], even though such an analysis would give the currently fastest algorithm for #RED-BLUE DOMINATING SET. Instead, we will use Algorithm 1 to both count the number of dominating sets in a graph and as a subroutine to compute the domatic number. In both cases, the input to Algorithm 1 is different compared to the number of vertices in a problem instance. Therefore, we need two different running-time analyses, which we defer to Sects. 4.2 and 4.3.

We conclude by noticing that Algorithm 1 can easily be modified such that it counts the number of partial red-blue dominating sets of size κ that dominate exactly t blue vertices, for each combination of values for κ and t with $0 \leq \kappa \leq |\mathcal{R}|$ and $0 \leq t \leq |\mathcal{B}|$.

Proposition 10 *Algorithm 1 can be modified such that it counts the number of partial red-blue dominating sets of size κ that dominate exactly t blue vertices, for each κ, t with $0 \leq \kappa \leq |\mathcal{R}|$ and $0 \leq t \leq |\mathcal{B}|$, while generating exactly the same graphs as subproblems.*

Proof We modify the algorithm such that it does not compute lists L with the number of red-blue dominating sets of each size, but matrices M with for each value of κ and t the number of subsets of $\mathcal{R}$ of size κ that dominate exactly t vertices from $\mathcal{B}$.

It is easy to see that this modification has no influence on the annotation procedure, the running time of the procedure CountRBDS-DP(G), or the branching on a red vertex (where two matrices instead of lists are now added up). When branching on blue vertices, we replace the inclusion/exclusion-based branching rule in lines 15–17 by an extended inclusion/exclusion-based branching rule that is formalised in the following pseudocode:

```
15:    Let $M_{\text{optional}} = \text{CountRBDS}(G[(\mathcal{R} \cup \mathcal{B}) \setminus \{b\}], A)$
16:    Let $M_{\text{not}} = M_{\text{forbidden}} = \text{CountRBDS}(G[(\mathcal{R} \cup \mathcal{B}) \setminus N[b]], A \setminus N(e))$
17:    Decrease the parameter $t$ by one in $M_{\text{optional}}$ and $M_{\text{forbidden}}$
18:    **return** $M_{\text{not}} + M_{\text{optional}} - M_{\text{forbidden}}$
```

Compare this to Eq. (3.6) in Sect. 3.1. In the above pseudocode, M_{not} corresponds to the branch where the requirement to dominate the vertex b is not satisfied, while M_{optional} and $M_{\text{forbidden}}$ represent the two branches generated for the case where this requirement is satisfied, and where the parameter t is adjusted correspondingly. Since M_{not} and $M_{\text{forbidden}}$ correspond to the same subproblem (as in Eq. (3.6)), it follows that this branching rule generates exactly the same graphs as subproblems as the inclusion/exclusion-based branching rule that it replaces. Correctness follows from the correctness of Eq. (3.6). □

4.2 Counting (Partial) Dominating Sets

We now give our first application of Algorithm 1: we use it to count dominating sets and partial dominating sets in general graphs (not red-blue graphs). As a result, we obtain the currently fastest polynomial-space algorithms for problems such as #DOMINATING SET, PARTIAL DOMINATING SET, and MINIMUM WEIGHT DOMINATING SET for the case where the size of the set of possible weight sums is polynomially bounded. We note that the annotation procedure used in Algorithm 1 plays an important role in the running-time analyses of these algorithms.

Theorem 11 *There exists an algorithm that counts the number of dominating sets of size κ in a graph G, for each value $0 \le \kappa \le n$, in $\mathcal{O}(1.5673^n)$ time and polynomial space.*

Proof Recall the simple transformation from a graph H to a red-blue graph $G = (\mathcal{R} \cup \mathcal{B}, E)$ given in Proposition 2, and notice that there exists a size-preserving one-to-one correspondence between the dominating sets in H and the red-blue dominating sets in G. After applying this transformation, we can use Algorithm 1 to count the number of red-blue dominating sets in G of each size κ to obtain the number of dominating sets in H of each size κ, as required.

We use *measure and conquer* to perform the run-time analysis of Algorithm 1 when applied to a red-blue graph G obtained in this way. Doing so, we introduce weight functions $v, w : \mathbb{N} \to \mathbb{R}_+$ and use the following measure k on the size of a subproblem (G, A) where $G = (\mathcal{R} \cup \mathcal{B}, E)$:

$$k := \sum_{b \in \mathcal{B}, b \notin A} v(d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(b)) + \sum_{r \in \mathcal{R}, r \notin A} w(d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(r))$$

Notice that the transformation from the n-vertex graph H results in a red-blue graph G with n red vertices and n blue vertices. As a result, G has measure at most $(v_{\max} + w_{\max})n$, where $v_{\max} = \max_{i \in \mathbb{N}} v(i)$ and $w_{\max} = \max_{i \in \mathbb{N}} w(i)$.

Let $\Delta v(i) = v(i) - v(i-1)$ and $\Delta w(i) = w(i) - w(i-1)$. We impose the following constraints on the weights functions v, w:

1. $v(0) = v(1) = 0$
2. $\Delta v(i) \geq 0$ for all $i \geq 2$
3. $\Delta v(i) \geq \Delta v(i+1)$ for all $i \geq 2$
4. $2\Delta v(3) \leq v(2)$
5. $w(0) = w(1) = 0$
6. $\Delta w(i) \geq 0$ for all $i \geq 2$
7. $\Delta w(i) \geq \Delta w(i+1)$ for all $i \geq 2$
8. $2\Delta w(4) \leq w(2)$

Notice that annotating a vertex reduces the measure of an instance, and that annotated vertices do not contribute to the measure. Constraints 1 and 5 represent the fact that if the degree of a vertex becomes at most one in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$, then it is annotated and, as such, does not contribute to the measure. Constraints 2 and 6 represent the fact that we want vertices with a higher degree to contribute more to the measure of an instance. Furthermore, Constraints 3 and 7 are non-restricting steepness inequalities that make the formulation of the problem easier. The function of Constraints 4 and 8 is explained later.

We formulate a series of recurrence relations to analyse the branching of the algorithm. Consider branching on a red vertex $r \in \mathcal{R}$ with b_i neighbours of degree i in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$. In the branch where r is taken in the red-blue dominating sets, we remove r decreasing the measure by $w(d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(r))$, we remove all its neighbours decreasing the measure by $\sum_{i=2}^{\infty} b_i v(i)$, and we reduce the degrees of all vertices at distance two from r. We can bound the decrease in measure due to this last reduction by $\Delta w(d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(r)) \sum_{i=2}^{\infty} b_i (i-1)$ because Constraint 7 ensures that, each time we reduce the degree of any red vertex by one, the measure decreases by at least $\Delta w(d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(r))$. Notice that if the degree of a red vertex different from r is lowered to at most one, then we could in principle decrease the measure by too much, as $w(1) = 0$. If $d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(r) \geq 4$, then this is prevented by Constraint 8. If $d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(r) = 3$, then we know that all neighbours of r have degree two since Algorithm 1 prefers to branch on red vertices. Also, no two of these degree two neighbours of r in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ can have the same neighbours since this would force them to be annotated. Hence, we still have a decrease of measure of at least $\Delta w(d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(r)) \sum_{i=2}^{\infty} b_i (i-1)$. In the branch where r is not taken in the red-blue dominating sets (where r is discarded), we remove r decreasing the measure by $w(d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(r))$, and we reduce the degrees of the neighbours of r decreasing the measure by $\sum_{i=2}^{\infty} b_i \Delta v(i)$.

Let Δk_{take} and $\Delta k_{\text{discard}}$ be the decrease of the measure in the branch where we take r in the red-blue dominating sets and where we discard it, respectively. We have shown that:

$$\Delta k_{\text{take}} \geq w(d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(r)) + \sum_{i=2}^{\infty} b_i v(i) + \Delta w(d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(r)) \sum_{i=2}^{\infty} b_i (i-1)$$

$$\Delta k_{\text{discard}} \geq w(d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(r)) + \sum_{i=2}^{\infty} b_i \Delta v(i)$$

Now, consider branching on a blue vertex b in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ with r_i neighbours of degree i in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$. Let $\Delta k_{\text{optional}}$ and $\Delta k_{\text{forbidden}}$ be the decrease of the measure in the branch where it is optional to dominate the blue vertex b and forbidden to dominate it, respectively. In the same way, we deduce the following symmetrical inequalities:

$$\Delta k_{\text{optional}} \geq v(d_{(\mathcal{R}\cup\mathcal{B})\setminus A}(b)) + \sum_{i=2}^{\infty} r_i \Delta w(i)$$

$$\Delta k_{\text{forbidden}} \geq v(d_{(\mathcal{R}\cup\mathcal{B})\setminus A}(b)) + \sum_{i=2}^{\infty} r_i w(i) + \Delta v(d_{(\mathcal{R}\cup\mathcal{B})\setminus A}(b)) \sum_{i=2}^{\infty} r_i (i-1)$$

Considering all possible cases for the branching of Algorithm 1, we obtain the following set of recurrence relations. Let $N(k)$ be the number of subproblems generated on a problem of measure k. For all $d_{(\mathcal{R}\cup\mathcal{B})\setminus A}(r) \geq 3$ and b_i such that $\sum_{i=2}^{d_{(\mathcal{R}\cup\mathcal{B})\setminus A}(r)-1} b_i = d_{(\mathcal{R}\cup\mathcal{B})\setminus A}(r)$, and all $d_{(\mathcal{R}\cup\mathcal{B})\setminus A}(b) \geq 3$ and r_i such that $\sum_{i=2}^{d_{(\mathcal{R}\cup\mathcal{B})\setminus A}(b)} r_i = d_{(\mathcal{R}\cup\mathcal{B})\setminus A}(b)$, we have:

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

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

An upper bound on the solution to this set of recurrence relations is of the form α^k. Thus, we have proven that Algorithm 1 can be used to count the number of dominating sets of size κ in G, for each value $0 \leq \kappa \leq n$, in $\mathcal{O}(\alpha^{(v_{\max}+w_{\max})n})$ time and polynomial space.

The last step is to compute weight functions v and w that minimise $\alpha^{v_{\max}+w_{\max}}$. We have done so by computer and have obtained $\alpha = 1.205693$ using the following weights:

i	2	3	4	5	6	>6
$v(i)$	0.640171	0.888601	0.969491	0.998628	1.000000	1.000000
$w(i)$	0.819150	1.218997	1.362801	1.402265	1.402265	1.402265

This proves a running time of $\mathcal{O}(1.205693^{(1+1.402265)n}) = \mathcal{O}(1.5673^n)$. □

The result of Theorem 11 can be used to give faster exact exponential-time polynomial-space algorithms for a number of domination problems in graphs.

Corollary 12 *There exist $\mathcal{O}(1.5673^n)$-time polynomial-space algorithms for the following problems*:

1. #DOMINATING SET
2. PARTIAL DOMINATING SET
3. MINIMUM WEIGHT DOMINATING SET *for the case where the set of possible weight sums is polynomially bounded*

Proof (1.) Follows directly from Theorem 11: use the same algorithm and output the number of dominating sets for the smallest value of κ for which this number is non-zero.

(2.) Using Proposition 10, we can modify Algorithm 1 such that it counts the number of partial red-blue dominating sets of size κ that dominate exactly t blue vertices, for each κ, t with $0 \leq \kappa \leq |\mathcal{R}|$ and $0 \leq t \leq |\mathcal{B}|$, while generating exactly the same graphs as subproblems. Observe that the simple transformation from a graph H to a red-blue graph $G = (\mathcal{R} \cup \mathcal{B}, E)$ given in Proposition 2 also induces a size-preserving one-to-one correspondence between the partial dominating sets in H dominating exactly t vertices and the partial red-blue dominating sets in G dominating exactly t vertices. Consequently, we can use the modified version of Algorithm 1 to solve PARTIAL DOMINATING SET. The running time follows from the proof of Theorem 11.

(3.) It is easy to modify Algorithm 1 in such a way that it counts the number of red-blue dominating sets of total weight κ, for each possible value of κ. If the set of possible values of κ is polynomially bounded, i.e., the number of possible weight sums is polynomially bounded, then clearly this slows the algorithm down by only a polynomial factor. Therefore, the result follows from the proof of Theorem 11 by reasoning similar to (2). □

In the case of PARTIAL DOMINATING SET and MINIMUM WEIGHT DOMINATING SET, we note that one can construct the associated (partial) dominating sets in H by repeating the counting algorithms a polynomial number of times in the following way. First compute the minimum size/weight k of a solution. Then, assign a vertex v in H to the (partial) dominating set and test if this alters the minimum size/weight of a solution: this can be done by applying Algorithm 1 to an input red-blue graph G that is modified to represent the assignment of vertices in H, i.e., an input where all blue vertices corresponding to already dominated vertices in H are removed, and where all red vertices corresponding to selected/discarded vertices in H are removed also. If the minimum size/weight of the solution changes, we discard the vertex v; otherwise, we keep it in the solution that we are constructing. We repeat this process for the next vertex until we have decided for all vertices in H whether they belong to the solution under construction.

4.3 Computing the Domatic Number

The second result we obtain using Algorithm 1 is an $\mathcal{O}(2.7139^n)$-time polynomial-space algorithm for DOMATIC NUMBER. For this result, we cannot apply the algorithm as direct as in Theorem 11. Instead, we will use the algorithm as a subroutine in the inclusion/exclusion approach to set partitioning by Björklund et al. [13].

We will first introduce the reader to this inclusion/exclusion approach to set partitioning by Björklund et al. It uses the following result to compute the domatic number.

Proposition 13 [13] *Let $\mathcal{D}$ be a family of sets over the universe $\mathcal{V}$, and let $k \in \mathbb{N}$. The number of ways $p_\kappa(\mathcal{D})$ to partition $\mathcal{V}$ into k sets from $\mathcal{D}$ equals*:

$$p_k(\mathcal{D}) = \sum_{X \subseteq \mathcal{V}} (-1)^{|X|} a_k'(X) \tag{4.1}$$

where $a'_k(X)$ *equals the number of* k*-tuples* $(S_1, S_2, \dots, S_k)$ *with* $S_i \in \{D \in \mathcal{D} \mid D \cap X = \emptyset\}$ *and for which* $\sum_{i=1}^{k} |S_i| = |\mathcal{V}|$.

Proof Direct application of the inclusion/exclusion formula. The formula for $p_k(\mathcal{D})$ equals the right-hand side of Eq. (3.1) with $a(X)^\kappa$ replaced by $a'_k(X)$. For details, see [13]. □

If we let $\mathcal{V} = V$ and let $\mathcal{D}$ be the set of dominating sets in H, then Proposition 13 can be used to check whether there exists a domatic k-partition, i.e., whether the domatic number of a graph is at least k, by checking whether $p_k(\mathcal{D}) > 0$ or not. To do so, we would need to compute the numbers $a'_k(X)$. In other words, for every $X \subseteq V$, we would need to compute the number of k-tuples $(S_1, S_2, \dots, S_k)$ with $S_i \in \{D \in \mathcal{D} \mid D \cap X = \emptyset\}$ and for which $\sum_{i=1}^{k} |S_i| = n$.

Let $d_\kappa(V')$ be the number of dominating sets of size κ in H using only vertices from $V' \subseteq V$. Björklund et al. [13] show how to compute the values $a'_k(X)$ required to use Proposition 13 from the numbers $d_\kappa(V')$. To this end, they give the following recurrence that can be used for dynamic programming.

Let $a'_X(i, j)$ be the number of i-tuples $(S_1, S_2, \dots, S_i)$ where each S_i is a dominating set in H using only vertices from $V \setminus X$ and such that $\sum_{i=1}^{k} |S_i| = j$ and $S_l \cap X = \emptyset$ (where $1 \le l \le k$). The values $a'_X(i, j)$ can be computed by dynamic programming over the following recurrence summing over the possible sizes for the i-th element of the i-tuple $(S_1, S_2, \dots, S_i)$:

$$a'_X(i, j) = \sum_{l=0}^{j} a'_X(i-1, j-l) \cdot d_l(V \setminus X) \tag{4.2}$$

Björklund et al. obtain the required values by observing that $a'_k(X) = a'_X(k, n)$.

We are now ready to prove our next result.

Theorem 14 *The domatic number of a graph can be computed in* $\mathcal{O}(2.7139^n)$ *time and polynomial space.*

Proof We use Proposition 13 for increasing $k = 3, 4, \dots$ to find the domatic number of H. To do so, we need to compute the required values $a'_k(X)$.

For each $V' \subseteq V$, we use Algorithm 1 on $G = (\mathcal{R} \cup \mathcal{B}, E)$, where $\mathcal{R} = V'$, $\mathcal{B} = V$, and $E = \{(r, b) \mid r \in \mathcal{R}, b \in \mathcal{B}, b \in N[r] \text{ in } H\}$ to produce a list containing, for each value $0 \le \kappa \le n$, the number of dominating sets in H of size κ using only vertices in V'. We observe that each such list can be used to compute a single value $a'_k(X)$ from Eq. (13) by the dynamic programming procedure shown above.

We obtain a polynomial-space algorithm for DOMATIC NUMBER by producing these lists one at a time and summing their corresponding contributions to the formula. Since the 2^n calls to Algorithm 1 are on instances of different sizes, the total number of subproblems generated by Algorithm 1 over all 2^n calls can be bounded from above by:

$$\sum_{i=0}^{n}\binom{n}{i}\alpha^{v_{\max}n+w_{\max}i}=\left(\alpha^{v_{\max}}(1+\alpha^{w_{\max}})\right)^n$$

Here, we use the notation from the proof of Theorem 11. Because this number of subproblems equals the exponential factor in the running time, we conclude that our algorithm for DOMATIC NUMBER runs in $\mathcal{O}^*((\alpha^{v_{\max}}(1+\alpha^{w_{\max}}))^n)$ time.

We recompute the weight functions used in the proof of Theorem 11 minimising $\alpha^{v_{\max}}(1+\alpha^{w_{\max}})$ and obtain $\alpha = 1.099437$ using the following set of weights:

i	2	3	4	5	6	7	>7
$v(i)$	0.822647	0.971653	1.000000	1.000000	1.000000	1.000000	1.000000
$w(i)$	2.039702	3.220610	3.711808	3.919500	4.016990	4.052718	4.052729

The running time is $\mathcal{O}((1.099437(1+1.099437^{4.052729}))^n)=\mathcal{O}(2.7139^n)$. □

Again, one can construct the associated domatic partitions (dominating sets) by repeating the counting algorithms a polynomial number of times. This is slightly more complex compared to the same results for PARTIAL DOMINATING SET and MINIMUM WEIGHT DOMINATING SET as it requires that, for each value $a'_k(X)$ from Eq. (13), Algorithm 1 is called a different input, each corresponding to one of the k thus far constructed domatic partitions. But, this difference increases the running time only by a polynomial factor.

4.4 On Symmetry in Counting Partial Red-Blue Dominating Sets

We conclude this section by considering an interesting property that arises from the symmetry between the roles of the red and blue vertices when counting partial red-blue dominating sets. That such symmetry exists is visible from the fact that branching on a red vertex or a blue vertex has similar effects on a red-blue graph. The symmetry property stated in Proposition 15 below follows from our extension of the usual inclusion/exclusion approach to partial requirements and is of independent interest. A modulo 2 variant was used in a reduction in recent work [21], see also for example [57]. However, it will not be used to obtain any of the other results in this paper.

Given a red-blue graph $G=(\mathcal{R}\cup\mathcal{B},E)$, let $b_{i,j}(\mathcal{R},\mathcal{B})$ is the number of partial red-blue dominating sets of size i that dominate exactly j blue vertices. That is:

$$b_{i,j}(\mathcal{R},\mathcal{B})=\left|\{X\subseteq\mathcal{R}:|X|=i\wedge|N[X]\cap\mathcal{B}|=j\}\right| \tag{4.3}$$

Given a red-blue graph $G=(\mathcal{R}\cup\mathcal{B},E)$ with red vertex set $\mathcal{R}$ and blue vertex set $\mathcal{B}$, we define its *flipped graph* G' to be G with the colours of every vertex flipped between red and blue, i.e., G' is the graph G with red vertex set $\mathcal{B}$ and blue vertex set $\mathcal{R}$.

From the symmetry between the roles of the red vertices and blue vertices the following proposition follows. This proposition shows that the matrix M_G that contains the values $b_{i,j}(\mathcal{R},\mathcal{B})$, for all i and j, associated with the graph G can be transformed into the matrix $M_{G'}$ associated with the flipped graph G' in polynomial time.

Proposition 15 *Given a red-blue graph $G = (\mathcal{R} \cup \mathcal{B}, E)$, let the values $b_{i,j}(\mathcal{R}, \mathcal{B})$ associated with G for $0 \leq i \leq |\mathcal{R}|$ and $0 \leq j \leq |\mathcal{B}|$ be as defined above, and let $b_{j,i}(\mathcal{B}, \mathcal{R})$ be the same values associated to the flipped graph G' of G. We have that*:

$$b_{i,j}(\mathcal{R}, \mathcal{B}) = \sum_{k=0}^{|\mathcal{R}|} \sum_{l=0}^{|\mathcal{B}|} (-1)^{l+j-|\mathcal{B}|} \binom{l}{|\mathcal{B}| - j} \binom{|\mathcal{R}| - k}{i} b_{l,k}(\mathcal{B}, \mathcal{R}) \tag{4.4}$$

Proof In Sect. 3.1, we gave the following formula obtained by expanding the recurrence of the extended inclusion/exclusion-based branching rule (Eq. (3.7)):

$$a_t(R, O, F) = \sum_{X \subseteq R} (-1)^{|X|-|R|+t} \binom{|X|}{|R| - t} a_0(\emptyset, O \cup (R \setminus X), F \cup X) \tag{4.5}$$

We will give a similar formula here, where $a_t(R, O, F)$ is replaced by $b_{i,j}(\mathcal{R}, \mathcal{B})$.

Notice that $\mathcal{B}$ now corresponds to the set of properties that initially are required properties. If we choose to make a subset $X \subseteq \mathcal{B}$ of these properties forbidden, then $\mathcal{R}$ is influenced by this choice of X as no subsets of $\mathcal{R}$ containing a vertex that is a neighbour of a forbidden vertex may be selected. In this way, we obtain the following formula summing over all sets of vertices $X \subseteq \mathcal{B}$ that correspond to the blue vertices that we forbid to be dominated:

$$b_{i,j}(\mathcal{R}, \mathcal{B}) = \sum_{X \subseteq \mathcal{B}} (-1)^{|X|-|\mathcal{B}|+j} \binom{|X|}{|\mathcal{B}| - j} b_{i,0}(\mathcal{R} \setminus N(X), \emptyset) \tag{4.6}$$

$$= \sum_{X \subseteq \mathcal{B}} (-1)^{|X|-|\mathcal{B}|+j} \binom{|X|}{|\mathcal{B}| - j} \binom{|\mathcal{R} \setminus N(X)|}{i} \tag{4.7}$$

The second equality here follows from the definition of $b_{i,j}(\mathcal{R}, \mathcal{B})$: $\binom{|\mathcal{R} \setminus N(X)|}{i}$ equals the number of ways to choose i red vertices that have no forbidden neighbours.

If we now group the summands of the summation by the size of $\mathcal{R} \cap N(X)$ and the size of X, then we obtain the following equation:

$$b_{i,j}(\mathcal{R}, \mathcal{B}) = \sum_{k=0}^{|\mathcal{R}|} \sum_{l=0}^{|\mathcal{B}|} \sum_{\substack{X \subseteq \mathcal{B} \\ |\mathcal{R} \cap N(X)|=k, |X|=l}} (-1)^{l-|\mathcal{B}|+j} \binom{l}{|\mathcal{B}| - j} \binom{|\mathcal{R}| - k}{i} \tag{4.8}$$

$$= \sum_{k=0}^{|\mathcal{R}|} \sum_{l=0}^{|\mathcal{B}|} (-1)^{l-|\mathcal{B}|+j} \binom{l}{|\mathcal{B}| - j} \binom{|\mathcal{R}| - k}{i} \sum_{\substack{X \subseteq \mathcal{B} \\ |\mathcal{R} \cap N(X)|=k, |X|=l}} 1 \tag{4.9}$$

$$= \sum_{k=0}^{|\mathcal{R}|} \sum_{l=0}^{|\mathcal{B}|} (-1)^{l+j-|\mathcal{B}|} \binom{l}{|\mathcal{B}| - j} \binom{|\mathcal{R}| - k}{i} b_{l,k}(\mathcal{B}, \mathcal{R}) \tag{4.10}$$

The last equality follows again from the definition of $b_{j,i}(\mathcal{B}, \mathcal{R})$. □

5 Parameterised Algorithms for Set Splitting Problems

In the previous section, we have shown that our (extended) inclusion/exclusion-based branching technique can be used to obtain faster exact exponential-time algorithms. In this section, we show that the same technique can also be used in the setting of parameterised algorithms. Specifically, we give faster parameterised algorithms for the well-studied problem k-SET SPLITTING and its generalisation k-NOT-ALL-EQUAL SATISFIABILITY. Our algorithms use the fact that small linear kernels exists for these problems. The use of such small linear kernels is an approach that often is used to obtain fast parameterised algorithms based on techniques from exact exponential-time algorithms; for more examples see [18, 22, 66].

5.1 A Faster Parameterised Algorithm for k-Not-All-Equal SAT

We will now present our faster parameterised algorithm for k-NOT-ALL-EQUAL SATISFIABILITY. This algorithm consists of three steps: first we apply a kernelisation step and an additional preprocessing step that can both be found in the literature and are given in Propositions 16 and 17 below. Thereafter, we solve the remaining instance using a branch-and-reduce algorithm that uses the extended inclusion/exclusion-based branching rule and is analysed using measure and conquer.

We say that a clause is *split* by an assignment if the assignment sets both a literal to true and a literal to false. Also we will work with the incidence graph of a CNF-formula in a way similar to the incidence graph of a set system: The incidence graph of a CNF-formula is a bipartite graph where the color classes represent the variables and clauses respectively. Different than the incidence graph of a set system we label the edges with a $+$ or $-$, depending on whether the variable occurs as a negative literal. Then for a vertex v of the incidence graph, we use $N(v)$ to denote its neighbourhood as usual, but also $N^+(v)$ to denote all vertices adjacent to v with an edge labelled with a $+$, and $N^-(v)$ to denote all vertices adjacent to v with an edge labelled with a $-$.

A *kernel* for a parameterised problem P is a polynomial-time algorithm that, given an instance x of P with parameter k, transforms x into an equivalent instance x' of P with parameter $k' \leq g(k)$ in such a way that x' has size at most $f(k)$ for some computable function $f(k)$.

Our algorithm first uses the following kernel by Lokshtanov and Saurabh [54].

Proposition 16 [54] k-NOT-ALL-EQUAL SATISFIABILITY *admits a kernel with at most* $2k$ *clauses and at most* k *variables.*

This result allows us to perform a polynomial-time kernelisation step that transforms a given instance into an equivalent instance that has at most $2k$ clauses and at most k variables.

After this kernelisation step, we apply the following test as a preprocessing step that allows us to directly decide that instances containing sufficiently many large clauses are YES-instances.

Proposition 17 [20] *Let s_i be the number of clauses of size i in a given k-*NOT-ALL-EQUAL SATISFIABILITY *instance. The instance is a* YES*-instance if*:

$$k \leq \frac{1}{2}s_2 + \frac{3}{4}s_3 + \frac{7}{8}s_4 + \frac{15}{16}s_5 + \frac{31}{32}s_6 + \frac{63}{64}s_7 + \cdots + \left(1 - \frac{1}{2^{i-1}}\right)s_i + \cdots \quad (5.1)$$

Proof Consider a random assignment of the variables, setting it to true with probability $\frac{1}{2}$ for each variable independently. For any clause of cardinality i, we consider the probability that the clause is split. This probability is independent of whether the first literal is negated or not, and, for each additional literal, this probability is multiplied by $\frac{1}{2}$. Hence, a clause of cardinality i has a probability of $\frac{1}{2^{i-1}}$ of not being split and hence a probability of $1 - \frac{1}{2^{i-1}}$ of being split.

By linearity of expectation, we find that the right hand side of the inequality in the proposition is the expected number of clauses split by a random variable assignment. Because there must exist an assignment that splits has at least this number of clauses, we have a YES-instance if the instance satisfies the stated inequality. □

If the kernelisation and preprocessing steps do not solve our problem instance, then we apply Algorithm 2 to the incidence graph of the instance. This algorithm computes a list with the number of variable assignments splitting exactly l clauses, for each $0 \leq l \leq 2k$. From this list, it is easy to see whether there exists a variable assignment splitting at least k sets, as required.

Algorithm 2 takes the incidence graph $I = (\mathcal{C} \cup \mathcal{V}, E)$ of the k-NOT-ALL-EQUAL SATISFIABILITY instance, a set of annotated vertices A, and a partitioning of the set of clauses $\mathcal{C}$ into three sets $\mathcal{T}$, $\mathcal{F}$, and $\mathcal{W}$ as input. This partitioning of $\mathcal{C}$ allows the algorithm to remove a vertex that represents a clause as soon as the assignment of a variable is fixed by the algorithm. When a vertex $v \in \mathcal{V}$ representing an variable is removed from I, then the partitioning will be updated to keep track of the fact that all sets that contained v contain a true or false literal. More precisely, clauses $C \in \mathcal{C}$ are put in $\mathcal{T}$ (*true*) or in $\mathcal{F}$ (*false*) if C has a positively or negatively set literal, respectively. Initially, all clauses $C \in \mathcal{C}$ are in $\mathcal{W}$ (*white/without colour*); this set contains all clauses of which the variables have not been assigned thus far. When a clause $C \in \mathcal{C}$ is split and thus contains both positively and negatively set clauses, then the vertex that represents C is removed from I.

Algorithm 2 uses the same annotation procedure as Algorithm 1 from Sect. 4.1 that annotates low degree vertices such that they will be ignored by the branching rules. Because this procedure is identical, we know by Lemma 8 that the treewidth of I is at most two when NS-DP$(I, \mathcal{T}, \mathcal{F}, \mathcal{W})$ is called. Hence, the subroutine NS-DP$(I, \mathcal{T}, \mathcal{F}, \mathcal{W})$ can be implemented in polynomial time using dynamic programming on tree decompositions (similarly to Corollary 9).

There are three different branching rules that Algorithm 2 uses, two of which use extended inclusion/exclusion-based branching as defined in Sect. 3.1.

1. **Branching on a vertex representing a variable** (lines 11–15 of Algorithm 2). We recursively solve two subproblems, one in which the variable is set to true and one in which variable is set to false. Hereafter, we shift the computed lists of numbers to update the number of split clauses. Finally, we compute the required results by adding the two computed lists component wise.

Algorithm 2 An algorithm for counting variable assignments that split exactly k clauses

Input: the incidence graph $I = (\mathcal{C} \cup \mathcal{V}, E)$ of a k-NOT-ALL-EQUAL SATISFIABILITY instance $(\mathcal{C}, \mathcal{V})$, a partitioning of the clauses $\mathcal{C} = \mathcal{T} \cup \mathcal{F} \cup \mathcal{W}$ where $\mathcal{T}$ and $\mathcal{F}$ are the sets of clauses that contain a literal that is set to true and false, respectively, and $\mathcal{W}$ is the set of clauses without yet assigned variables, and a set $A \subset (\mathcal{C} \cup \mathcal{V})$ of annotated vertices; initially $\mathcal{W} = \mathcal{C}, \mathcal{T} = \mathcal{F} = \emptyset$, $A = \emptyset$

Output: a list with, for each k, the number of assignments of $\mathcal{V}$ splitting exactly k clauses.

```
NS($I, \mathcal{T}, \mathcal{F}, \mathcal{W}, A$):
1: if there exists a vertex $v \in (\mathcal{C} \cup \mathcal{V}) \setminus A$ of degree at most one in $I[(\mathcal{C} \cup \mathcal{V}) \setminus A]$ then
2:     return NS($I, \mathcal{T}, \mathcal{F}, \mathcal{W}, A \cup \{v\}$)
3: else if there exist two vertices $v_1, v_2 \in (\mathcal{C} \cup \mathcal{V}) \setminus A$ both of degree two in $I[(\mathcal{C} \cup \mathcal{V}) \setminus A]$ that have the same two neighbours then
4:     return NS($I, \mathcal{T}, \mathcal{F}, \mathcal{W}, A \cup \{v_1\}$)
5: else
6:     Let $v \in \mathcal{V} \setminus A$ be a vertex such that $d_{(\mathcal{C} \cup \mathcal{V}) \setminus A}(v)$ is maximum
7:     Let $w \in \mathcal{W} \setminus A$ be a vertex such that $d_{(\mathcal{C} \cup \mathcal{V}) \setminus A}(w)$ is maximum
8:     Let $c \in (\mathcal{T} \cup \mathcal{F}) \setminus A$ be a vertex such that $d_{(\mathcal{C} \cup \mathcal{V}) \setminus A}(c)$ is maximum
9:     if $\max\{d_{(\mathcal{C} \cup \mathcal{V}) \setminus A}(v),\ d_{(\mathcal{C} \cup \mathcal{V}) \setminus A}(w),\ d_{(\mathcal{C} \cup \mathcal{V}) \setminus A}(c)\} \le 2$ then
10:        return NS-DP($I, \mathcal{T}, \mathcal{F}, \mathcal{W}$)
11:    else if $d_{(\mathcal{C} \cup \mathcal{V}) \setminus A}(v) = \max\{d_{(\mathcal{C} \cup \mathcal{V}) \setminus A}(v),\ d_{(\mathcal{C} \cup \mathcal{V}) \setminus A}(w),\ d_{(\mathcal{C} \cup \mathcal{V}) \setminus A}(c)\}$ then
12:        Let $L_{\text{true}} = \text{NS}(I[(\mathcal{C} \cup \mathcal{V}) \setminus N_{\mathcal{F} \cup \mathcal{V}}[v]], (\mathcal{T} \cup N^+_{\mathcal{W}}(v)) \setminus N^-_{\mathcal{W}}(v), (\mathcal{T} \cup N^-_{\mathcal{W}}(v)) \setminus N^+_{\mathcal{W}}(v), \mathcal{W} \setminus N(v), A \setminus N_{\mathcal{F}}(v))$
13:        Let $L_{\text{false}} = \text{NS}(I[(\mathcal{C} \cup \mathcal{V}) \setminus N_{\mathcal{T} \cup \mathcal{V}}[v]], (\mathcal{T} \cup N^-_{\mathcal{W}}(v)) \setminus N^+_{\mathcal{W}}(v), (\mathcal{F} \cup N^+_{\mathcal{W}}(v)) \setminus N^-_{\mathcal{W}}(v), \mathcal{W} \setminus N(v), A \setminus N_{\mathcal{T}}(v))$
14:        Update the number of split sets in $L_{\text{true}}$ and $L_{\text{false}}$
15:        return $L_{\text{true}} + L_{\text{false}}$
16:    else if $d_{(\mathcal{C} \cup \mathcal{V}) \setminus A}(w) = \max\{d_{(\mathcal{C} \cup \mathcal{V}) \setminus A}(v),\ d_{(\mathcal{C} \cup \mathcal{V}) \setminus A}(w),\ d_{(\mathcal{C} \cup \mathcal{V}) \setminus A}(c)\}$ then
17:        $L_{\text{optional}} = \text{NS}(I[(\mathcal{C} \cup \mathcal{V}) \setminus \{w\}], \mathcal{T}, \mathcal{F}, \mathcal{W} \setminus \{w\}, A)$
18:        Make all literals of the clause $w$ positive by accordingly flipping variables and merge all variables of $w$ into a single variable.
19:        $L_{\text{not}} = L_{\text{forbidden}} = \text{NS}(I[(\mathcal{C} \cup \mathcal{V}) \setminus \{w\}], \mathcal{T}, \mathcal{F}, \mathcal{W} \setminus \{w\}, A \setminus N(w))$
20:        Decrease the parameter $k$ by one in $L_{\text{optional}}$ and $L_{\text{forbidden}}$
21:        return $L_{\text{not}} + L_{\text{optional}} - L_{\text{forbidden}}$
22:    else // $d_{(\mathcal{C} \cup \mathcal{V}) \setminus A}(c) > d_{(\mathcal{C} \cup \mathcal{V}) \setminus A}(v), d_{(\mathcal{C} \cup \mathcal{V}) \setminus A}(w)$
23:        $L_{\text{optional}} = \text{NS}(I[(\mathcal{C} \cup \mathcal{V}) \setminus \{c\}], \mathcal{T} \setminus \{c\}, \mathcal{F} \setminus \{c\}, \mathcal{W}, A)$
24:        if $c \in \mathcal{T}$ then
25:            $L_{\text{not}} = L_{\text{forbidden}} =$
                   $\text{NS}(I[(\mathcal{C} \cup \mathcal{V}) \setminus N^2_{\mathcal{V} \cup \mathcal{F}}[c]], (\mathcal{T} \setminus \{c\}) \cup N^2_{\mathcal{W}}(c), \mathcal{F} \setminus N^2(c), \mathcal{W} \setminus N^2(c),$
                       $A \setminus N^2_{\mathcal{V} \cup \mathcal{F}}[c])$
26:        else // $c \in \mathcal{F}$
27:            $L_{\text{not}} = L_{\text{forbidden}} =$
                   $\text{NS}(I[(\mathcal{C} \cup \mathcal{V}) \setminus N^2_{\mathcal{V} \cup \mathcal{T}}[c]], \mathcal{T} \setminus N^2(c), (\mathcal{F} \setminus \{c\}) \cup N^2_{\mathcal{W}}(c), \mathcal{W} \setminus N^2(c),$
                       $A \setminus N^2_{\mathcal{V} \cup \mathcal{T}}[c])$
28:        Update the number of split clauses in $L_{\text{not}}$ and $L_{\text{forbidden}}$
29:        Decrease the parameter $k$ by one in $L_{\text{optional}}$ and $L_{\text{forbidden}}$
30:        return $L_{\text{not}} + L_{\text{optional}} - L_{\text{forbidden}}$
```

2. **Branching on a vertex representing a clause without assigned variables** (lines 16–21 of Algorithm 2). We apply the extended inclusion/exclusion-based branching rule as defined in Sect. 3.1. Because we are computing lists of numbers with the number of assignments splitting exactly l clauses, for each $0 \leq l \leq 2k$, we need to compute only two of these lists: one corresponding to the subproblem where it is *optional* to split the clause, and one corresponding to the subproblem where it is *forbidden* to split the clause.

   In the optional branch, we remove the vertex representing the clause. As a result, we are indifferent about splitting the corresponding clause in this branch. In the forbidden branch, we remove the vertex representing the clause and merge all vertices that represent the variables in the clause to a single vertex (here we assume that a clause does not contain one variable twice, this can be done since if this is the case it is automatically split). A solution to this instance corresponds to a solution of the original instance in which all literals in the clause are either true or false, i.e., in which the clause is not split.
3. **Branching on a vertex representing a clause with assigned variables** (lines 22–30 of Algorithm 2). Similar to branching on vertices that represent clauses with assigned variables, we use the extended inclusion/exclusion-based branching rule and generate two subproblems. In the optional branch, we again remove the vertex corresponding to the clause since we are indifferent about splitting it. In the forbidden branch, we generate an instance equivalent to make sure that the corresponding clause will not be split. Since the clause representing the set already has an assigned variable, giving a true or false litereal, this means that we must assign all variables in the set such that all literals in the clause are either true or all false. Note that other clauses can be split as a result of this operation.

To analyse the worst-case running time of our algorithm for k-NOT-ALL-EQUAL SATISFIABILITY, we again use measure and conquer. To this end, we introduce the following measure μ on a subproblem $(I, \mathcal{T}, \mathcal{F}, \mathcal{W}, A)$. Note that, in contrast to the rest of this paper, we use μ instead of k for the measure to avoid confusion with the parameter k of k-NOT-ALL-EQUAL SATISFIABILITY.

$$\mu := \sum_{v \in \mathcal{V}, v \notin A} v(d_{(\mathcal{C} \cup \mathcal{V}) \setminus A}(v)) + \sum_{w \in \mathcal{W}, w \notin A} x(d_{(\mathcal{C} \cup \mathcal{V}) \setminus A}(w)) + \sum_{c \in (\mathcal{T} \cup \mathcal{F}), c \notin A} y(d_{(\mathcal{C} \cup \mathcal{V}) \setminus A}(c))$$

We use the following values for the weights:

i	≤ 1	2	3	4	5	6	> 6
$v(i)$	0.000000	0.726515	1.000000	1.000000	1.000000	1.000000	1.000000
$x(i)$	0.000000	0.525781	0.903938	1.054594	1.084859	1.084885	1.084885
$y(i)$	0.000000	0.387401	0.617482	0.758071	0.792826	0.792852	0.792852

We remind the reader that, similar to previous analyses, the weight functions v, x, and y have been chosen in such a way that the running time claimed in Theorem 19 is as fast as possible.

Lemma 18 *Algorithm 2 runs in $\mathcal{O}(1.31242^{\mu})$ time on an input of measure μ.*

Proof Let $N(\mu)$ be the number of subproblem generated on an input of measure μ. We generate a large set of recurrence relations of the form $N(\mu) \leq N(\mu - \Delta\mu_1) + N(\mu - \Delta\mu_2)$ representing all possible cases in which Algorithm 2 can branch.

Analogous to the previous measure-and-conquer analyses, we let $\Delta v(i) = v(i) - v(i-1)$, $\Delta x(i) = x(i) - x(i-1)$, $\Delta y(i) = y(i) - y(i-1)$, and notice that the weight functions v, x, and y satisfy the following constraints:

1. $\Delta v(i), \Delta x(i), \Delta y(i) \geq 0$ for all i
2. $y(i) \leq x(i)$ for all i
3. $\Delta v(i) \geq \Delta v(i+1)$ for all $i \geq 2$
4. $\Delta x(i) \geq \Delta x(i+1)$ for all $i \geq 2$
5. $\Delta y(i) \geq \Delta y(i+1)$ for all $i \geq 2$
6. $2\Delta v(3) \leq v(2)$
7. $2\min\{\Delta y(4),\ \Delta x(3)\} \leq y(2)$

These constraints are similar to those used in the proof of Theorem 11. That is, Constraint 1 represents the fact that we want vertices with a higher degree in $I[(\mathcal{C} \cup \mathcal{V}) \setminus A]$ to contribute more to the measure of an instance. Constraint 2 represents the fact that moving a set from $\mathcal{W}$ to either $\mathcal{T}$ or $\mathcal{F}$ decreases the measure of an instance. Furthermore, Constraints 3, 4, and 5 are non-restricting steepness inequalities that make the formulation of the problem easier. Finally, Constraints 6 and 7 exist to make sure that we do not decrease the measure too much when using the maximum degree to bound the decrease of the measure due to reducing the degree of a vertex by one; this is similar to the function of Constraints 4 and 8 in the proof of Theorem 11.

We first consider branching on a vertex v that represents a variable. Let this vertex have w_i, t_i, and f_i neighbours of degree i in $I[(\mathcal{C} \cup \mathcal{V}) \setminus A]$ whose corresponding clauses are elements of $\mathcal{W}$, $\mathcal{T}$, and $\mathcal{F}$, respectively. Consider the branch where we assign the variable v to *true*. In this branch, the measure decreases by $v(d_{(\mathcal{C}\cup\mathcal{V})\setminus A}(v))$ due to the fact that the vertex representing the variable is removed. Because the t_i clauses in $\mathcal{T}$ containing the variable represented by v are split, the measure decreases by an additional $t_i y(i)$. The f_i vertices of degree i in $I[(\mathcal{C} \cup \mathcal{V}) \setminus A]$ that represent clauses in $\mathcal{F}$ that contain the variable have their sizes reduced, which decreases the measure by $f_i \Delta y(i)$. Furthermore, the w_i vertices of degree i in $I[(\mathcal{C} \cup \mathcal{V}) \setminus A]$ each represent sets in $\mathcal{W}$ that are now put in $\mathcal{F}$ and reduced in size; this decreases the measure by an additional $w_i(x(i) - y(i-1))$. Finally, since the t_i vertices of degree i in $I[(\mathcal{C} \cup \mathcal{V}) \setminus A]$ that represent sets in $\mathcal{T}$ containing v are removed, other elements are reduced in frequency: this reduces the measure by an additional $\Delta v(d_{(\mathcal{C}\cup\mathcal{V})\setminus A}(v)) \sum_{i=2}^{\infty} t_i(i-1)$ by Constraints 3 and 6 and the fact that v is of maximum degree among the vertices that represent elements. Here, we use Constraint 6 to prevent that we decrease the measure by too much if we reduce the degree of a degree i vertex representing an element a total of i times; this is identical to what Constraints 4 and 8 do in the proof of Theorem 11.

Let $\Delta\mu_{\text{true}}$ and $\Delta\mu_{\text{false}}$ be the decrease of the measure in the subproblem where the variable is assigned to true or false, respectively. Since the above analysis is symmetric if we assign v to either true or false, we have deduced the following inequalities:

$$\Delta\mu_{\text{false}} \geq v(d_{(\mathcal{C}\cup\mathcal{V})\setminus A}(v)) + \sum_{i=2}^{\infty}\bigl(t_i y(i) + f_i \Delta y(i) + w_i(x(i) - y(i-1))\bigr)$$

$$+ \Delta v(d_{(\mathcal{C}\cup\mathcal{V})\setminus A}(v)) \sum_{i=2}^{\infty} t_i(i-1)$$

$$\Delta\mu_{\text{true}} \geq v(d_{(\mathcal{C}\cup\mathcal{V})\setminus A}(v)) + \sum_{i=2}^{\infty}\bigl(f_i y(i) + t_i \Delta y(i) + w_i(x(i) - y(i-1))\bigr)$$

$$+ \Delta v(d_{(\mathcal{C}\cup\mathcal{V})\setminus A}(v)) \sum_{i=2}^{\infty} f_i(i-1)$$

Now, consider branching on a vertex s representing a clause in $\mathcal{T}$ or $\mathcal{F}$ containing e_i variables whose corresponding vertices have degree i in $I[(\mathcal{C}\cup\mathcal{V})\setminus A]$. In the optional branch, s is removed and the vertices representing the variables contained in the clause have their degrees reduced: this decreases the measure by $y(d_{(\mathcal{C}\cup\mathcal{V})\setminus A}(s)) + \sum_{i=2}^{\infty} e_i \Delta v(i)$. In the forbidden branch, s is removed together with the vertices representing the variables contained in the set: this decreases the measure by $y(s) + \sum_{i=2}^{\infty} e_i v(i)$. Furthermore, removing the vertices that represent these variables reduces the degrees of other vertices that represent clauses. Because of the branching order and Constraints 4 and 5, this decreases the measure by either at least $\Delta y(d_{(\mathcal{C}\cup\mathcal{V})\setminus A}(s))$ if the clause is in $\mathcal{T}$ or $\mathcal{F}$ and by at least $\Delta x(d_{(\mathcal{C}\cup\mathcal{V})\setminus A}(s) - 1)$ if the clause is in $\mathcal{W}$. In total, this gives a decrease in the measure of at least $\min\{\Delta y(d_{(\mathcal{C}\cup\mathcal{V})\setminus A}(s)),\ \Delta x(d_{(\mathcal{C}\cup\mathcal{V})\setminus A}(s) - 1)\}$ for each time the degree of a vertex is reduced. Similar to how we used Constraint 6 when deriving the formula associated with branching on a variable v, we use Constraint 7 here to make sure that we do not decrease the measure by too much if we reduce the degree of a vertex of degree i a total of i times.

Let $\Delta\mu_{\text{optional}}$ and $\Delta\mu_{\text{forbidden}}$ be the decrease of the measure in the subproblem where splitting the clause is made optional and forbidden, respectively. We have deduced:

$$\Delta\mu_{\text{optional}} \geq y(d_{(\mathcal{C}\cup\mathcal{V})\setminus A}(s)) + \sum_{i=2}^{\infty} e_i \Delta v(i)$$

$$\Delta\mu_{\text{forbidden}} \geq y(d_{(\mathcal{C}\cup\mathcal{V})\setminus A}(s)) + \sum_{i=2}^{\infty} e_i v(i)$$

$$+ \min\{\Delta y(d_{(\mathcal{C}\cup\mathcal{V})\setminus A}(s)),\ \Delta x(d_{(\mathcal{C}\cup\mathcal{V})\setminus A}(s) - 1)\} \cdot \sum_{i=2}^{\infty} e_i(i-1)$$

Finally, we consider branching on a vertex representing a clause in $\mathcal{W}$. Analogous to the above, we derive:

$$\Delta\mu_{\text{optional}} \geq x(d_{(\mathcal{C}\cup\mathcal{V})\setminus A}(s) + \sum_{i=2}^{\infty} e_i \Delta v(i)$$

$$\Delta\mu_{\text{forbidden}} \geq x(d_{(\mathcal{C}\cup\mathcal{V})\setminus A}(s)) + \sum_{i=2}^{\infty} e_i v(i) - v(\sum_{i=2}^{\infty} e_i(i-1))$$

The main difference in the derivation compared to the case where we branch on a vertex representing a clause in $\mathcal{F}$ or $\mathcal{T}$ is the term $-v(\sum_{i=2}^{\infty} e_i(i-1))$. This term accounts for the fact that if we decide that it is forbidden to split a clause from $\mathcal{W}$, then we do not remove the vertices representing the variables in this clause, but merge these variables (and the corresponding vertices) to a single variable (vertex) instead. This causes a new vertex of degree at most $\sum_{i=2}^{\infty} e_i(i-1)$ to be created, which has measure at most $v(\sum_{i=2}^{\infty} e_i(i-1))$.

Having derived lower bounds on the decrease of the measure in every subproblem, we can now solve the large resulting set of recurrence relations with the given weights. As a result, we have found that $N(\mu) \leq 1.31242^{\mu}$.

Since both the annotation procedure and the procedure NS-DP($I, \mathcal{T}, \mathcal{F}, \mathcal{W}$) can be implemented in polynomial time, the running time is dominated by the exponentially number of generated subproblems. We conclude that the algorithm runs in $\mathcal{O}(1.31242^{\mu})$ time. □

We are now ready to prove the main result of this section.

Theorem 19 *There exists an $\mathcal{O}^*(1.8213^k)$-time and polynomial-space algorithm for k-*NOT-ALL-EQUAL SATISFIABILITY.

Proof The claimed algorithm is the three step algorithm described above. This algorithm first applies the preprocessing given by Propositions 16 and 17, and then applies Algorithm 2 to the incidence graph of the resulting instance $(\mathcal{C}, \mathcal{V})$. Algorithm 2 then produces a list containing the number of variable assignments of the variables in $\mathcal{V}$ that split exactly l clauses from $\mathcal{C}$, for each $0 \leq l \leq 2k$. From this list, the algorithm can directly determine if there exists a variable assignment splitting at least k clause.

The exponential part of the running time of this algorithm comes from the call to Algorithm 2 since the preprocessing is done in polynomial time. We use Lemma 18 to compute this running time. By Proposition 16, an instance that is given to Algorithm 2 can have at most k variables; these variables each have measure at most 1. By Proposition 17, an instance that is given to Algorithm 2 must satisfy $k > \sum_{i=2}^{\infty}(1 - \frac{1}{2^{i-1}})s_i$, where s_i is the number of clauses of cardinality i. Therefore, the maximum measure z of such an instance is bounded by the solution of:

$$z = k + \max \sum_{i=2}^{\infty} x(i)s_i \quad \text{with the restriction} \quad \sum_{i=2}^{\infty}\left(1 - \frac{1}{2^{i-1}}\right)s_i < k \qquad (5.2)$$

We find that the maximum measure of an instance given to Algorithm 2 is obtain by setting s_3 to the maximum s_3 such that $s_3 < k/0.75$ and $s_i = 0$ for $i \neq 3$ and gives $z < 2.205251k$. The claimed running time follows from Lemma 18 since $1.31242^z < 1.31242^{2.205251k} < 1.8213^k$. □

Since k-SET SPLITTING is the special case of k-NOT-ALL-EQUAL SATISFIABILITY where all literals are positive, we directly obtain the following from Theorem 19:

Corollary 20 *There exists an $\mathcal{O}^*(1.8213^k)$-time and polynomial-space algorithm for k-SET SPLITTING.*

6 Improvements Using Exponential-Space

Several results from the previous sections can be improved at the cost of using exponential space. This can be done by letting the branching algorithms switch to dynamic programming methods when the generated subproblems are sufficiently sparse. This works because, in general, branching works well if the graph contains high-degree vertices while dynamic programming can be applied more efficiently to sparse graphs. This combination was first used by Fomin et al. [31]. In this section, we improve our algorithms in the same way as presented in [31].

This section is organised as follows. In Sect. 6.1, we improve our algorithm for counting (partial) dominating sets from Sect. 6.1 using dynamic programming on sparse graphs. This results in the currently fastest algorithm for PARTIAL DOMINATING SET. Hereafter, we apply the same improvements to our algorithms for k-SET SPLITTING and k-NOT-ALL-EQUAL SATISFIABILITY in Sect. 6.2.

6.1 Counting (Partial) Dominating Sets

Below, we show how to improve the running time of the algorithm of Theorem 11 at the cost of using exponential space in the same way as introduced by Fomin et al. in [31]. We do so by letting the algorithm that counts (partial) red-blue dominating sets (Algorithm 1) switch to a dynamic-programming-based approach when the red-blue graph is sparse enough. This dynamic programming approach is based on dynamic programming on tree decompositions, where the tree decompositions are obtained through using Proposition 7. As a result, we will obtain the currently fastest algorithm for (the counting version of) PARTIAL DOMINATING SET.

Consider Algorithm 3. This is Algorithm 1 modified in two ways: it is modified to count partial red-blue dominating sets as described in Proposition 10, and it is modified to switch to a dynamic programming approach at an earlier stage. More precisely, Algorithm 3 switches to this dynamic programming approach when G is of maximum degree four and has no blue vertices in $\mathcal{B} \setminus A$ that have a degree four neighbour in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ nor blue vertices in $\mathcal{B} \setminus A$ that have four degree three neighbours in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$. Notice that blue vertices of degree four are treated differently depending on the degrees of their neighbours. This is because, in the analysis below, these local differences can be used: in some cases, it is more efficient to

Algorithm 3 An exponential-space algorithm for counting partial red-blue dominating sets

```
Input: a red-blue graph $G = (\mathcal{R} \cup \mathcal{B}, E)$ and a set of annotated vertices $A$; initially
    $A = \emptyset$
Output: a matrix containing the number of partial red-blue dominating sets of size $\kappa$
    dominating exactly $t$ blue vertices for each value of $\kappa$ and $t$
CountRBDS($G, A$):
 1: if there exists a vertex $v \in (\mathcal{R} \cup \mathcal{B}) \setminus A$ of degree at most one in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$
    then
 2:   return CountRBDS($G, A \cup \{v\}$)
 3: else if there exist two vertices $v_1, v_2 \in (\mathcal{R} \cup \mathcal{B}) \setminus A$ both of degree two in
    $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ that have the same two neighbours then
 4:   return CountRBDS($G, A \cup \{v_1\}$)
 5: else
 6:   Let $r \in \mathcal{R} \setminus A$ be a vertex such that $d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(r)$ is maximal
 7:   Let $b \in \mathcal{B} \setminus A$ be a vertex such that $d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(b)$ is maximal
 8:   if $d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(r) \leq 4$ and $d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(b) \leq 4$ and there exist no vertex $b \in \mathcal{B}$ with
      a neighbour of degree four in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ and there exist no vertex $b \in \mathcal{B}$
      with four neighbours of degree three then
 9:     return CountRBDS-DP($G$)
10:   else if $d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(r) > d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(b)$ then
11:     Let $M_{\text{take}}$ = CountRBDS($G[(\mathcal{R} \cup \mathcal{B}) \setminus N[r]], A \setminus N(r)$)
12:     Let $M_{\text{discard}}$ = CountRBDS($G[(\mathcal{R} \cup \mathcal{B}) \setminus \{r\}], A$)
13:     Increase the cardinalities $\kappa$ in $M_{\text{take}}$ by one
14:     return $M_{\text{take}} + M_{\text{discard}}$
15:   else
16:     Let $M_{\text{optional}}$ = CountRBDS($G[(\mathcal{R} \cup \mathcal{B}) \setminus \{b\}], A$)
17:     Let $M_{\text{not}} = M_{\text{forbidden}}$ = CountRBDS($G[(\mathcal{R} \cup \mathcal{B}) \setminus N[b]], A \setminus N(e)$)
18:     Decrease the parameter $t$ by one in $M_{\text{optional}}$ and $M_{\text{forbidden}}$
19:     return $M_{\text{not}} + M_{\text{optional}} - M_{\text{forbidden}}$
```

branch on the blue vertex, while in other cases it is more efficient to process them while dynamic programming.

The rest of this section is used to prove that Algorithm 3 can be used to solve #PARTIAL DOMINATING SET in $\mathcal{O}(1.5014^n)$ time.

We reconsider the measure k used in the proof of Theorem 11:

$$k := \sum_{b \in \mathcal{B}, b \notin A} v(d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(b)) + \sum_{r \in \mathcal{R}, r \notin A} w(d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(r))$$

Here, we reuse it, but with the following different set of weights:

i	2	3	4	5	6	> 6
$v(i)$	0.498964	0.750629	0.913191	0.975726	0.999999	1.000000
$w(i)$	0.673168	1.046831	1.261070	1.352496	1.382025	1.386987

We start by proving the following lemma, analogous to [31].

Lemma 21 *For any $\epsilon > 0$, there exists an integer n_ϵ such that the following holds for any graph $G = (\mathcal{R} \cup \mathcal{B}, E)$ associated with a subproblem of measure at most k that Algorithm 3 solves by dynamic programming: if $|\mathcal{R} \cup \mathcal{B}| > n_\epsilon$, then the treewidth of G is at most $(0.245614 + \epsilon)k$. A tree decomposition of this width can be computed in polynomial time.*

Proof Let A be the set of annotated vertices in a subproblem with the red-blue graph $G = (\mathcal{R} \cup \mathcal{B}, E)$. Furthermore, let $\epsilon > 0$ be fixed, and let n_ϵ be as in Proposition 7. We will show that the pathwidth of $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ is at most $(0.245614 + \epsilon)k$ and that a path decomposition of this width can be obtained in polynomial time using Proposition 7. The result then follows because a path decomposition also is a tree decomposition, and because we can add all vertices in A without increasing the treewidth by the argument in the proof of Lemma 8.

Similar to [31], we formulate a linear program to compute the maximum width of a tree decomposition of G obtained in the way described above. In this linear program, all variables have the domain $[0, \infty)$.

$$\max\ z = \frac{1}{6}(x_3 + y_3) + \frac{1}{3}(x_4 + y_4) \tag{6.1}$$

$$\text{such that:}\quad 1 = \sum_{i=2}^{4} w(i)x_i + \sum_{i=2}^{4} v(i)y_i \tag{6.2}$$

$$\sum_{i=2}^{4} i x_i = \sum_{i=2}^{4} i y_i \tag{6.3}$$

$$y_4 = p_0 + p_1 + p_2 + p_3 \tag{6.4}$$

$$x_2 \geq \frac{4}{2}p_0 + \frac{3}{2}p_1 + \frac{2}{2}p_2 + \frac{1}{2}p_3 \tag{6.5}$$

$$x_3 \geq \frac{1}{3}p_1 + \frac{2}{3}p_2 + \frac{3}{3}p_3 \tag{6.6}$$

The objective function of this linear program represents the maximum pathwidth z of the graph $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ obtained using Proposition 7 per unit of the measure k. Notice that, in a subproblem solved by dynamic programming, $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ can have vertices of degree at most four. The variables x_i and y_i represent the number of red vertices of degree i and blue vertices of degree i per unit of the measure in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$, respectively. Since blue vertices of degree four can exist only if their neighbours have specific combinations of degrees, we introduce the variables p_i. Such a variable p_i represents the number of blue vertices of degree four that have i neighbours of degree three and $4 - i$ neighbours of degree two in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ per unit of the measure k. Form the branching rules of Algorithm 3, we can directly see that we need p_i only for $i \in \{0, 1, 2, 3\}$.

Consider the constraints of the above linear program. Constraint 6.2 guarantees that the variables x_k and y_k use exactly one unit of the measure k. Constraint 6.3

guarantees that the vertices with both colours are incident to the same total number of edges. Constraint 6.4 makes sure that the p_i sum up to y_4, which is the total number of blue vertices of degree four per unit of the measure. Finally, Constraints 6.5 and 6.6 make sure that if a blue vertex of degree four exists in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$, then sufficiently many red neighbours with the appropriate degrees exists, as required by definition of the variables p_i. That is, by definition of the variables p_i, a blue vertex of degree four in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ has i neighbours of degree three and $4-i$ neighbours of degree two per unit of the measure in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$. Therefore, the number of edges incident to a degree two vertex $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ is at least $4p_0 + 3p_1 + 2p_2 + 1p_3$. Since each such degree two vertex has two endpoints, Constraints 6.5 follows. Constraint 6.6 follows similarly.

The solution to this linear program is $z = 0.245614$ with all variables equal to zero, except: $x_4 = 0.589473$ and $y_3 = 0.442104$. The bound of $(0.245614 + \epsilon)k$ now follows from Proposition 7 and the argument in the first paragraph of this proof. □

Theorem 22 *There exists an algorithm that solves* #PARTIAL DOMINATING SET *in* $\mathcal{O}(1.5014^n)$ *time and space.*

Proof We again use the simple transformation from a graph H to a red-blue graph $G = (\mathcal{R} \cup \mathcal{B}, E)$ given in Proposition 2 to solve #PARTIAL DOMINATING SET on H by considering partial red-blue dominating sets in G. We use Algorithm 3 to count the number of partial red-blue dominating sets of size κ that dominate exactly t blue vertices, for each value of κ and t. The result follows from the running time analysis of this approach.

Let $N_h(k)$ be the number of subproblems of measure h generated by Algorithm 3 through branching when starting from an input with measure k. By Lemma 21, any subproblem of measure h solved by dynamic programming has treewidth at most $(0.245614 + \epsilon)h$. If we fix $\epsilon > 0$ small enough, then such a subproblem is solved in $\mathcal{O}^*(2^{(0.245614+\epsilon)h}) = \mathcal{O}(1.1856^h)$ time by Proposition 4.

To bound the number of subproblems generated by branching, we use the same recurrence relations as used in the proof of Theorem 11. However, because we now consider using Algorithm 3 instead of Algorithms 1, we remove those recurrence relations that do not correspond to the modified branching rule. I.e., we remove any recurrence corresponding to branching on a vertex of degree three or four, unless it corresponds to branching on a blue vertex of degree four with either a red neighbour of degree four or only red neighbours of degree three. If we solve the resulting set of recurrences with the measure defined above Lemma 21, we obtain: $N_h(k) \leq 1.1856^{k-h}$.

We now combine the analysis of the branching and the analysis of the dynamic programming and find that the total running time $T(k)$ on an input of measure k satisfies:

$$T(k) \leq \sum_{h \in H_k} N_h(k) \cdot \mathcal{O}(1.1856^h) \leq \sum_{h \in H_k} 1.1856^{k-h} \cdot \mathcal{O}(1.1856^h) = \sum_{h \in H_k} \mathcal{O}(1.1856^k)$$

where H_k is the set of all possible measures of subproblems generated when starting from a problem with measure k. We conclude that $T(k) = \mathcal{O}(1.1856^k)$ because we use only a finite number of weights, which makes $|H_k|$ polynomially bounded and thus disappear in the big-$\mathcal{O}$ notation by the decimal rounding of the base of the exponent.

Let $v_{\max} = \max_{i \in \mathbb{N}} v(i)$ and $w_{\max} = \max_{i \in \mathbb{N}} w(i)$. As the input instance given to Algorithm 3 used to solve an instance of #PARTIAL DOMINATING SET contains n red vertices and n blue vertices, this proves a running time of $\mathcal{O}(1.1856^{(v_{\max}+w_{\max})n}) = \mathcal{O}(1.1856^{(1+1.386987)n}) = \mathcal{O}(1.5014^n)$. □

We note that a variation of this algorithm can also be used to solve #DOMINATING SET and MINIMUM WEIGHT DOMINATING SET for the case where the set of possible weight sums is polynomially bounded. We do not state such results here as we will give slightly faster algorithms for these problems in Sect. 7.1.

6.2 k-Set Splitting and k-Not-All-Equal Satisfiability

We will now apply the same approach to improve our algorithms for k-SET SPLITTING and k-NOT-ALL-EQUAL SATISFIABILITY. Our presentation involves only the k-NOT-ALL-EQUAL SATISFIABILITY problem, the same result for k-SET SPLITTING follows analogously.

We first need a result that shows in which running time we can solve a subproblem consisting of an incidence graph $I = (\mathcal{S} \cup \mathcal{U}, E)$ of a k-NOT-ALL-EQUAL SATISFIABILITY instance $(\mathcal{C}, \mathcal{V})$ and a partitioning of the sets $\mathcal{C} = \mathcal{T} \cup \mathcal{F} \cup \mathcal{W}$ as in Algorithm 2 when we are given an instance with a tree decomposition of I. This result is given by the following proposition. We will not go into details of the proof as that would involve an in-depth treatment of dynamic programming on tree decompositions.

Proposition 23 *Given a tree decomposition of an incidence graph $I = (\mathcal{C} \cup \mathcal{V}, E)$ of a k-NOT-ALL-EQUAL SATISFIABILITY instance I of width tw and a partitioning of the sets $\mathcal{C} = \mathcal{T} \cup \mathcal{F} \cup \mathcal{W}$, the number of assignments of the variables $\mathcal{V}$ that split exactly k clauses, for each value $0 \leq k \leq |\mathcal{C}|$, can be computed in $\mathcal{O}^*(3^{tw})$ time.*

Sketch of Proof In general, algorithms on tree decompositions use $\mathcal{O}^*(\alpha^{tw})$ space where α equals the number of 'states' required to represent partial solutions; for examples, see [14]. For many problems (including this one), these algorithms can be made to run in the same time using the techniques in [68]. We give an argument for the $\mathcal{O}^*(3^{tw})$ time bound by considering the different roles that vertices in I can have and arguing what number of 'states' these vertices require.

A vertex in the incidence graph I can either represent an element $v \in \mathcal{V}$, a clause with assigned variable $c \in \mathcal{R} \cup \mathcal{G}$, or a clause without assigned variables $w \in \mathcal{W}$. For a variable, we need two states: assigned to true and assigned to false. For a clause with an assigned variable, we also need two states: the clause is split or not. For a clause without an assigned variable, we need three states: the set has only true literals, only false literals, or the clause is split. □

We modify Algorithm 2 such that it starts with dynamic programming as soon as the incidence graph I has maximum degree three. That is, lines 9–10 are replaced by the pseudocode below.

```
 9: if $\max\{d_{(\mathcal{C}\cup\mathcal{V})\setminus A}(v),\ d_{(\mathcal{C}\cup\mathcal{V})\setminus A}(w),\ d_{(\mathcal{C}\cup\mathcal{V})\setminus A}(c)\} \leq 3$ then
10:     return NEQSAT-DP($I, \mathcal{T}, \mathcal{F}, \mathcal{W}$)
```

Furthermore, the procedure NEQSAT-DP(I, $\mathcal{R}$, $\mathcal{G}$, $\mathcal{W}$) is now implemented using the above proposition together with Proposition 7.

Theorem 24 *There exists an $\mathcal{O}^*(1.7171^k)$-time and space algorithm for k-*NOT-ALL-EQUAL SATISFIABILITY.

Proof We will prove that the combination consisting of Algorithm 2, when modified as described above, the kernel from Proposition 16, and the preprocessing of Proposition 17 runs in $\mathcal{O}^*(1.7171^k)$-time and space.

To analyse the worst-case running time of our algorithm, we instantiate the measure μ for a subproblem $(I, \mathcal{T}, \mathcal{F}, \mathcal{W}, A)$ from Sect. 5.1 in the following way:

$$\mu := \sum_{v\in\mathcal{V}, v\notin A} v(d_{(\mathcal{C}\cup\mathcal{V})\setminus A}(v)) + \sum_{w\in\mathcal{W}, w\notin A} x(d_{(\mathcal{C}\cup\mathcal{V})\setminus A}(w)) + \sum_{c\in(\mathcal{C}\cup\mathcal{V}), c\notin A} y(d_{(\mathcal{C}\cup\mathcal{V})\setminus A}(c))$$

i	≤ 1	2	3	4	5	6	> 6
$v(i)$	0.000000	0.614383	0.889666	1.000000	1.000000	1.000000	1.000000
$x(i)$	0.000000	0.525781	0.861526	1.008031	1.076907	1.098145	1.098145
$y(i)$	0.000000	0.386212	0.568178	0.719229	0.798112	0.813954	0.813954

The set of recurrence relations that correspond to the branching of our modified algorithm are the recurrence relations from Lemma 18 for which $d_{(\mathcal{C}\cup\mathcal{V})\setminus A}(v) \geq 3$ when branching on an element $v \in \mathcal{V}$ and $d_{(\mathcal{C}\cup\mathcal{V})\setminus A}(c) \geq 3$ when branching on a clause $c \in \mathcal{C}$. When we solve this set of recurrence relations, we find that $N(\mu) \leq 1.28557^{\mu}$, where $N(\mu)$ again is the number of subproblems generated by branching when starting with an instance of measure μ.

Next, we consider the time and space used by the procedure NEQSAT-DP($I, \mathcal{T}, \mathcal{F}, \mathcal{W}$). For this, we could formulate a linear program similar to the one in Lemma 21. We will not do so, but identify the worst-case running time directly. Because we use Proposition 7 to obtain a tree decomposition, the largest treewidth is obtained when all vertices in I have degree three. Because the incidence graph is bipartite, this means that it contains an equal number of (degree three) vertices representing variables $v \in \mathcal{V}$ and (degree three) vertices representing clauses $c \in \mathcal{C}$. Moreover, all vertices representing clauses represent clauses with assigned variables since these use less measure than clauses without assigned variables. We conclude that the dynamic programming procedure requires the following amount of time and space:

$$3^{tw} \leq 3^{\frac{1}{6}(|\mathcal{C}|+|\mathcal{V}|)} \leq 3^{\frac{1}{6}\cdot 2\cdot \frac{\mu}{v(3)+y(3)}} \leq 1.28557^{\mu} \tag{6.7}$$

We conclude that the modified version of Algorithm 2 runs in $\mathcal{O}(1.28557^{\mu})$ time and space.

Identical to the proof of Theorem 19, we find that, after applying the kernel from Proposition 16 and the preprocessing of Proposition 17, the maximum measure μ of a k-NOT-ALL-EQUAL SATISFIABILITY instance with parameter k equals the value of z where:

$$z = k + \max \sum_{i=2}^{\infty} x(i) s_i \quad \text{with the restriction} \quad \sum_{i=2}^{\infty} \left(1 - \frac{1}{2^{i-1}}\right) s_i < k \tag{6.8}$$

This maximum measure equals $z < 2.152036k$. The claimed running time follows since $1.28557^z < 1.28557^{2.152036k} < 1.7171^k$. □

Corollary 25 *There exists an $\mathcal{O}^*(1.7171^k)$-time and space algorithm for k-*SET SPLITTING.

7 Additional Results

We conclude by giving some additional results that we also obtained by applying our new technique. These are a slightly faster algorithm for counting dominating sets obtained by introducing a different set of reduction rules in Sect. 7.1. This results in the currently fastest algorithm for #DOMINATING SET and an improved algorithm for MINIMUM WEIGHT DOMINATING SET for the case where the set of possible weight sums is polynomially bounded. Secondly, we consider DOMINATING SET and #DOMINATING SET restricted to some graph classes as studied by Gaspers et al. [41]. We will show how to improve these results using inclusion/exclusion-based branching in Sect. 7.2.

7.1 Counting Dominating Sets Slightly Faster

Next, we give another exponential space algorithm. We use this algorithm to obtain the currently fastest algorithms for #DOMINATING SET and MINIMUM WEIGHT DOMINATING SET for the case where the set of possible weight sums is polynomially bounded. The results obtained in this way are slightly faster than the results of using the equivalent of Theorem 22 for these problems. We present these result mainly for two reasons. First, because this approach results in a slightly faster algorithm, and this small improvement also results in slightly faster results in Sect. 7.2. Second, because the result shows an example of a branch-and-reduce algorithm using a ‘real’ set of reduction rules (similar to [33, 66]) instead of the annotation procedure.

Consider Algorithm 4.[1] This algorithm looks a lot more like earlier algorithms from the literature, e.g, see [33, 66]: it has a set of reduction rules similar to those

[1]An algorithm similar to Algorithm 4 has been published in an earlier draft of a part of this paper [69]. We note that the analysis published in [69] contains a mistake. We corrected this mistake and as a result, the preference order that can be found in the algorithm in [69] is no longer required. For the rest, both algorithms are very similar though they appear different. This is because Algorithm 4 is presented using red-blue dominating sets, while the Algorithm in [69] uses an equivalent presentation involving set covers. Furthermore, two reduction rules are omitted from Algorithm 4: one rule is superfluous (the base case in [69] is covered by the procedure CountRBDS2-DP(G, m)), and the other rule (connected components) is never used in the analysis and could thus be omitted.

Algorithm 4 An exponential-space algorithm for counting red-blue dominating sets of each size κ

Input: a red-blue graph $G = (\mathcal{R} \cup \mathcal{B}, E)$ and a function $m : \mathcal{R} \to \mathbb{N}$; initially $\forall r \in \mathcal{R} : m(r) = 1$

Output: a list with the number of red-blue dominating sets in G of size κ, for each value of κ

```
CountRBDS2($G, m$):
 1: if there exist two red vertices $r_1, r_2 \in \mathcal{R}$ with $N(r_1) = N(r_2)$ then
 2:    Modify $m$ such that $m(r_1) := m(r_1) + m(r_2)$
 3:    return CountRBDS2($G[(\mathcal{R} \cup \mathcal{B}) \setminus \{r_2\}], m$)
 4: else if there exists a blue vertex $b \in \mathcal{B}$ of degree one then
 5:    Let $L_{\text{take}}$ = CountRBDS2($G[(\mathcal{R} \cup \mathcal{B}) \setminus N[r]], m$) with $r$ the unique red neighbour of $b$
 6:    return $L_{\text{take}}$ after updating it using Proposition 26 for taking the red vertex $r$
 7: else if there exist two blue vertices $b_1, b_2 \in \mathcal{B}$ such that $N[b_1] \subseteq N[b_2]$ then
 8:    return CountRBDS2($G[(\mathcal{R} \cup \mathcal{B}) \setminus \{b_2\}, m$)
 9: else
10:    Let $r \in \mathcal{R}$ be a red vertex that is not an exceptional case in Overview 1 with $d(r)$ maximal
11:    Let $b \in \mathcal{B}$ be a blue vertex that is not an exceptional case in Overview 1 with $d(b)$ maximal
12:    if $d(r) \leq 3$ and $d(b) \leq 4$ then
13:       return CountRBDS2-DP($G, m$)
14:    else if $d(r) > d(b)$ or $d(b) \leq 4$ then
15:       Let $L_{\text{take}}$ = CountRBDS2($G[(\mathcal{R} \cup \mathcal{B}) \setminus N[r]], m$)
16:       Let $L_{\text{discard}}$ = CountRBDS2($G[(\mathcal{R} \cup \mathcal{B}) \setminus \{r\}], m$)
17:       Update $L_{\text{take}}$ using Proposition 26
18:       return $L_{\text{take}} + L_{\text{discard}}$
19:    else
20:       Let $L_{\text{optional}}$ = CountRBDS2($G[(\mathcal{R} \cup \mathcal{B}) \setminus \{b\}], m$)
21:       Let $L_{\text{forbidden}}$ = CountRBDS2($G[(\mathcal{R} \cup \mathcal{B}) \setminus N[b]], m$)
22:       return $L_{\text{optional}} - L_{\text{forbidden}}$
```

in [66] instead of the annotation procedure present in previous algorithms in this paper. Different from these previous algorithms, Algorithm 4 takes as input a multiplicity function $m : \mathcal{R} \to \mathbb{N}$ besides the red-blue graph G. Initially, this function takes the value 1 for each red vertex $r \in \mathcal{R}$. This multiplicity function is used to count the number of red vertices with identical blue neighbours so that the algorithm can remove all but one of these vertices while keeping track of their numbers using this function.

The reduction rule in lines 1–3 of Algorithm 4 removes these identical red vertices and updates the multiplicity function accordingly. This rule originates as a weakened form of the subsets rule from [33, 66] that still works when counting the number of solutions. If Algorithm 4, at a later stage, selects a red vertex to be in the red-blue dominating sets being counted, then it must correct the computed numbers for the fact that multiple red vertices are represented by this vertex. The algorithm uses the following proposition to do so; see lines 6 and 17.

Proposition 26 *Let x_κ be the number of red-blue dominating sets of size κ computed in a recursive call to Algorithm 4 where we just decided that a red vertex $r \in \mathcal{R}$ must be in every red-blue dominating set. That is, in a recursive call where we have decided to take at least one of the $m(r)$ copies of the red vertex r in the red-blue dominating set, but where we have not yet taken this into account in the values x_κ: we have removed only $N[r]$ from the incidence graph. Also, let x'_κ be the number of red-blue dominating sets of size κ when adjusted for the fact that we could have taken any number (but at least one) of the $m(r)$ copies of the red vertex r. Then:*

$$x'_\kappa = \sum_{i=1}^{m(r)} \binom{m}{i} x_{\kappa - i} \tag{7.1}$$

Proof We sum over the different number of copies i of the red vertex r that we can take. The binomial gives the number of ways to pick these i copies from the total of $m(r)$ copies. □

We note that, in the above formula, we assume that $x_i = 0$ for i outside the range $0 \le i \le n$.

Next, we consider the second reduction rule of Algorithm 4 at lines 4–6. Here, a blue vertex b exists that can be dominated by only one red vertex. However, due to the use of the multiplicity function, there may be multiple copies of this red vertex. We must take at least one of these copies in order to dominate b. We do so by recursively solving the problem where we take the red vertex in the red-blue dominating sets and using Proposition 26 to correct the computed values for the fact that we could also have taken a number of copies of this red vertex. We note that this reduction rule is very similar to the unique elements rule in [33, 66].

The last reduction rule (lines 7–8) considers blue vertices b_1, b_2 where the closed neighbourhood of one vertex is contained in the closed neighbourhood other: $N[b_1] \subseteq N[b_2]$. In this case, any red-blue dominating set that dominates b_1 also dominates b_2. Hence, we can remove b_2. We note that the multiplicity function does not play any role here, and that this reduction rule is highly similar to the subsumption rule in [66].

If no reduction rule is applicable, then Algorithm 4 branches or uses dynamic programming on tree decompositions obtained through applying Proposition 7, similar to Algorithm 3. A small difference to Algorithm 3 is that, because of the use of the multiplicity function m, the algorithm uses Proposition 26 to update the list containing the number of red-blue dominating sets of each size κ whenever it decides to take (at least one copy of) a red vertex in the solutions being counted. As a bigger difference, is that Algorithm 4 ignores certain vertices as candidates for branching: these are considered exceptional cases and are defined in Overview 1. These are blue vertices of degree five that have specific local configurations, and red vertices that have blue neighbours with such a specific local configuration. Also, Algorithm 4 never branches on vertices of degree at most three, and it never branches on blue vertices of

There are exceptional cases on which Algorithm 4 does not branch. These cases represent local configurations of vertices which would increase the running time of the algorithm when branched on, and that can be handled by the dynamic programming on tree decompositions quite effectively. The exceptional cases are:

1. Blue vertices of degree five that occur in many sets of small cardinality. More specifically, if we let a 5-tuple $(r_1, r_2, r_3, r_4, r_5)$ represent a blue vertex b of degree five with r_i red neighbours of degree i, then our special cases can be denoted as:

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

   Note that b can have at most one red neighbour of degree one due to the multiplicity function.
2. Red vertices of degree four or five that are a neighbour of a blue vertex corresponding to one of the exceptional cases defined above.

Overview 1: Exceptional cases for Algorithm 4

degree four: these are left for the dynamic programming phase of the algorithm. This means that if the maximum degree in G is four and there exist red vertices of degree four, then these are used for branching whether blue vertices of degree four exist or not. Similar to the result in Sect. 6.1, this precise selection of vertices to branch on based on their local configuration is used in the analysis: for some vertices branching is more effective, while for other vertices it is more efficient to process them while dynamic programming.

Before beginning with the running time analysis of Algorithm 4, we first need to state the following two propositions related to the dynamic programming performed by the algorithm.

Proposition 27 *Given a tree decomposition of a red-blue graph $G = (\mathcal{R} \cup \mathcal{B}, E)$ of width tw and a multiplicity function $m : \mathcal{R} \to \mathbb{N}$ counting the number of copies of each red vertex, the number of red-blue dominating sets of size κ, for each values $0 \leq \kappa \leq |\mathcal{R}|$, can be computed in $\mathcal{O}^*(2^{tw})$ time.*

Proof Simple modification of Proposition 4 using standard dynamic programming on tree decomposition techniques. Because computations are local, we can use Eq. (7.1) from Proposition 26 each time that a red vertex is processed (introduced or forgotten). □

Proposition 28 *When using Proposition 7 to obtain a tree decomposition of the red-blue graph G, we can ignore all vertices of degree one in G, effectively lowering the degrees of their neighbours, when applying the formula for the upper bound on the treewidth of the resulting tree decomposition. This can be done while increasing the width of the resulting tree decomposition by at most one.*

Proof If we remove all degree-one vertices from G and then compute a tree decomposition T using Proposition 7, then we can reintroduce these vertices in the following way. Let X_i be a bag containing the neighbour of a degree-one vertex v, and let i be the corresponding node of the tree decomposition T. We can add v by adding a new node j below i in T with $X_j = \{u, v\}$, where $(u, v) \in E$. This increases the treewidth by at most one. Furthermore, this operation can be repeated such that the width of T increases by at most 1 after reintroducing all degree-one vertices. □

We now start proving a bound on the running time of Algorithm 4. To this end, we instantiate the measure k used in the analyses with the following weights:

$$k := \sum_{e \in \mathcal{U}} v(d(e)) + \sum_{s \in \mathcal{S}} w(d(s))$$

i	1	2	3	4	5	6	7	>7
$v(i)$	0.000000	0.275603	0.551206	0.658484	0.719009	0.745509	0.749538	0.749538
$w(i)$	0.354816	0.625964	0.785112	0.923325	0.986711	1.000000	1.000000	1.000000

Again, let $\Delta v(i) = v(i) - v(i-1)$, $\Delta w(i) = w(i) - w(i-1)$ and observe that the measure satisfies the following constraints:

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. $2\Delta v(5) \leq v(2)$

These constraints have the same roles are their counterparts in the proof of Theorem 11.

We start the analysis of the running time of Algorithm 4 by bounding the number of subproblems generated by branching.

Lemma 29 *Let $N_h(k)$ be the number of subproblems of measure h generated by Algorithm 4 on an input of measure k. Then*:

$$N_h(k) < 1.26089^{k-h}$$

Proof Similar to previous analyses, we derive a set of recurrence relations of the following form:

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

with the appropriate values for $\Delta k_{\text{optional}}$ and $\Delta k_{\text{forbidden}}$ for every possible cases corresponding to branching on a blue vertex, and the appropriate values for $\Delta k_{\text{discard}}$ and Δk_{take} for every possible case corresponding to branching on a red vertex.

If we branch on a blue vertex, this leads to almost the same recurrence relations as in Theorem 11. Let b be a blue vertex with r_i neighbours of degree i, and let $\Delta k_{\text{optional}}$ and $\Delta k_{\text{forbidden}}$ be the decrease in the measure in the branch where we make it optional to cover the element represented by b, or forbidden to cover the element represented by b, respectively.

$$\Delta k_{\text{optional}} \geq v(d(b)) + \sum_{i=1}^{\infty} r_i \Delta w(i)$$

$$\Delta k_{\text{forbidden}} \geq v(d(b)) + \sum_{i=1}^{\infty} r_i w(i) + \Delta v(d(b)) \sum_{i=1}^{\infty} (i-1) r_i$$

The only difference with the same recurrence relations in Theorem 11 is that we now also consider neighbours of degree one.

If we branch on a red vertex r, then the changes are somewhat bigger. Let r be a red vertex with b_i blue neighbours of degree i, and let $\Delta k_{\text{discard}}$ and Δk_{take} be the decrease in the measure in the branch where we take some of the $m(r)$ copies of the red vertex r in the counted solutions, or where we discard them, respectively. Different to the formula for $\Delta k_{\text{discard}}$ given in Theorem 11, reduction rules now fire when r has blue neighbours of degree two. In the branch where the red vertices represented by r are discarded, their blue degree-two neighbours get degree one and are removed by the reduction rules. These blue degree-two neighbours cannot have any common neighbours besides r since that would have fired the reduction rule dealing with blue vertices b_1, b_2 with $N[b_1] \subseteq N[b_2]$ (lines 7–8) before branching. Therefore, the reduction rule for degree-one blue vertices (lines 4–6) removes at least one additional red vertex per neighbour of degree two, each having measure at least $w(1)$.

In this way, we obtain the following bounds on $\Delta k_{\text{discard}}$ and Δk_{take}:

$$\Delta k_{\text{discard}} \geq w(d(r)) + \sum_{i=2}^{\infty} b_i \Delta v(i) + b_2 w(1)$$

$$\Delta k_{\text{take}} \geq w(d(r)) + \sum_{i=2}^{\infty} b_i v(i) + \Delta w(d(r)) \sum_{i=2}^{\infty} (i-1) b_i$$

We solve this set of recurrence relations using the measure defined above this lemma and obtain an upper bound on the solution of $N_h(k) \leq 1.26089^{k-h}$. Notice that the exact definition of the branching rules and, in particular, the exceptional cases in Overview 1 play an important role in the analysis as they restrict the generated set of recurrence relations. □

Next, we prove a bound on the running time of a call to CountRBDS2-DP(G, m).

Lemma 30 *ExpCSC-DP(I, m) runs in time $\mathcal{O}(1.25334^k)$ when called on a set cover instance of measure k.*

Proof We will prove an upper bound on the treewidth of a red-blue graph of measure k on which the procedure CountRBDS2-DP(G, m) is called. Similar to Lemma 21 (and [31]), we do so by formulating a linear program in which all variables have the domain $[0, \infty)$.

We start by stating the first part of the linear program:

$$\max\ z = \frac{1}{6}n_3 + \frac{1}{3}n_4 + \frac{13}{30}n_5 \tag{7.2}$$

$$\text{such that:}\quad 1 = \sum_{i=1}^{5} w(i)x_i + \sum_{i=2}^{5} v(i)y_i \tag{7.3}$$

$$\sum_{i=1}^{5} i x_i = \sum_{i=2}^{5} i y_i \tag{7.4}$$

Again, x_i and y_i represent the number of red and blue vertices of degree i per unit of the measure k in a worst-case instance, respectively. And again, the above constraints guarantee that these variables correspond to exactly one unit of measure, and that the vertices of both colours are incident to the same total number of edges.

The objective function, however, is formulated in terms of the variables n_i. It is still based on Proposition 7 and represents the maximum treewidth z of a graph per unit of measure up to the term ϵn. The variables n_i represent the number of vertices of degree i in the input graph per unit of measure. Following Proposition 28, these degrees are taken after removing any blue vertices of degree one. All these blue vertex of degree one will contribute measure, but not increase the treewidth by more than one in total (which does not influence the exponential factor in the running time), so we can assume that they exist only if involved in an exceptional case.

Let C be the set of exceptional cases for blue vertices of degree five, as defined in Overview 1, and let c_i be the number of red neighbours of degree i of exceptional case $c \in C$. We introduce the variables p_c for the number of occurrences of each exceptional case $c \in C$ per unit of the measure k. These definitions directly give us the following additional constraints that can be added to the linear program:

$$n_3 = x_3 + y_3 \tag{7.5}$$

$$n_4 = x_4 + y_4 + \sum_{c\in C, c_1>0} p_c \tag{7.6}$$

$$n_5 = x_5 + \sum_{c\in C, c_1=0} p_c \tag{7.7}$$

$$y_5 = \sum_{c\in C} p_c \tag{7.8}$$

Notice that, here, we use that in a subproblem on which CountRBDS2-DP(G, m) is called, a vertex can have degree at most five. We also use that, in such a subproblem, vertices of degree five exist only if they are exceptional cases (Overview 1).

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 degrees of the red vertices that are neighbours of the blue vertices represented by the exceptional cases. We impose a lower bound on the number of red vertices of degree one, two, and three in the instance by introducing Constraint 7.9. Also, we impose an upper bound on the number of red vertices of cardinality four, five, and six by using that there can be at most c_i such red vertices per exceptional degree five blue vertex. This is done in Constraint 7.10.

$$x_i \geq \sum_{c \in \mathcal{C}} \frac{c_i}{i} p_c \quad \text{for } i \in \{1, 2, 3\} \tag{7.9}$$

$$x_i \leq \sum_{c \in \mathcal{C}} c_i p_c \quad \text{for } i \in \{4, 5\} \tag{7.10}$$

The solution to this linear program is $z = 0.325782$ with all variables equal to zero, except: $x_3 = n_3 = 0.781876$ and $n_4 = y_4 = 0.586407$. Using Proposition 27, we conclude that the dynamic programming on the tree decomposition can be performed in time $\mathcal{O}^*(2^{(z+\epsilon)k}) = \mathcal{O}(1.25334^k)$, when choosing ϵ sufficiently small. □

Combining Lemmas 29 and 30 gives the main result of this section.

Theorem 31 *There exists an algorithm that solves* #DOMINATING SET *in* $\mathcal{O}(1.5002^n)$ *time and space.*

Proof We again use the simple transformation from a graph H to a red-blue graph $G = (\mathcal{R} \cup \mathcal{B}, E)$ given in Proposition 2 to solve #DOMINATING SET on H by counting red-blue dominating sets in G. We use Algorithm 4 to count he number of red-blue dominating sets of size κ, for each value of κ. The result follows from the running time analysis of this approach.

Let $T(k)$ be the time used on a problem of measure k, and let H_k be the set of all possible measures that subproblems of a problem of measures k can have. Then, by Lemmas 29 and 30:

$$\begin{aligned} T(k) &\leq \sum_{h \in H_k} N_h(k) \cdot \mathcal{O}(1.25334^h) \\ &\leq \sum_{h \in H_k} 1.26089^{k-h} \cdot \mathcal{O}(1.25334^h) \\ &\leq \sum_{h \in H_k} \mathcal{O}(1.26089^k) \end{aligned}$$

Because we use only a finite number of weights, $|H_k|$ is polynomially bounded. Hence, Algorithm 4 runs in $\mathcal{O}(1.26089^k)$ time. This proves a running time of $\mathcal{O}(1.26089^{(v_{\max}+w_{\max})n}) = \mathcal{O}(1.26089^{(0.749538+1)n}) = \mathcal{O}(1.5002^n)$. □

Corollary 32 *There exists an* $\mathcal{O}(1.5002^n)$*-time-and-space algorithm solving* MINIMUM WEIGHT DOMINATING SET *for the case where the set of possible weight sums is polynomially bounded.*

Proof Identical to Corollary 12 using Theorem 31 instead of Theorem 11. □

7.2 Dominating Set on Some Graph Classes

We conclude our presentation of results by giving one last application of inclusion/exclusion-based branching. We show that inclusion/exclusion-based branching can be used to directly improve the algorithms for DOMINATING SET restricted to some graph classes by Gaspers et al. [41].

Gaspers et al. consider exact exponential-time algorithms for DOMINATING SET on some graph classes on which this problem remains $\mathcal{NP}$-complete [41]. 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 either there are 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 on a tree decomposition.

Below, we show that we need fewer vertices of high degree to obtain the same effect by using two branching rules instead of one. We can use this to improve the results by Gaspers et al. on four of these graph classes. We note that, on some graph classes, part of the improvement comes from using faster algorithms to solve the problem on tree decompositions.[2] We also note that our improvement for the fifth graph class (chordal graphs) is based on the faster algorithm in Theorem 31 and the faster algorithm on tree decompositions; not because we require fewer vertices of high degree. See Table 1 for an overview of results.

We begin by showing that having many vertices of high degree can be beneficial to the running time of an algorithm. First, consider the following result of Gaspers et al. [41]:

Proposition 33 [41] *Let $t \geq 1$ be a fixed integer, and let α be such that there exists an $\mathcal{O}(\alpha^n)$-time algorithm for* RED-BLUE DOMINATING SET. *If $\mathcal{G}_t$ is a class of graphs with, for all $G \in \mathcal{G}_t$, $|\{v \in V : d(v) \geq t-2\}| \geq t$, then there exists an $\mathcal{O}(\alpha^{2n-t})$-time algorithm to solve* DOMINATING SET *on graphs in $\mathcal{G}_t$.*

We now give and prove a variant of this proposition; the resulting lemma uses Algorithm 4 and its running-time analysis given in Sect. 6.1.

Lemma 34 *Let, for each integer $t \geq 1$, $\mathcal{G}_t$ be the class of graphs with $|\{v \in V : d(v) \geq t-2\}| \geq \frac{1}{2}t$. There exists an algorithm that counts the number of dominating sets of size κ, for each value $0 \leq \kappa \leq n$, in a graph $G \in \mathcal{G}_t$ in $\mathcal{O}(1.5002^{n-\frac{1}{2}t})$ time.*

Proof Given a graph G, we determine the largest integer $t \geq 1$ such that $G \in \mathcal{G}_t$. It is easy to see that this can be done in polynomial time. We transform G into the red-blue graph $G' = (\mathcal{R} \cup \mathcal{B}, E)$ such that the dominating sets in G correspond to the red-blue dominating sets in G' as described in Proposition 2. Let X be the set of vertices of degree at least $t-2$ in G from the definition of $\mathcal{G}_t$.

[2]Gaspers et al. [41] did not use Proposition 5 (see [68]) but slower algorithms on tree decompositions instead.

We will prove the lemma by giving an algorithm that counts the red-blue dominating sets of size κ in G', for each value $0 \le \kappa \le n$. Let α be the base of the exponent of the running time of Algorithm 4 using the measure given in Sect. 7.1 with maximum weights $v_{\max}$ and $w_{\max}$ for a blue vertex and a red vertex, respectively. The algorithm consists of a procedure that, given G', applies two branching rules generating $t+1$ red-blue graphs of measure at most $(v_{\max}+w_{\max})(n-\frac{1}{2}t)$ from which the result can be computed. We solve the instances generated by this procedure using Algorithm 4. Since the number of instances generated is at most linear in n, this gives an algorithm running in $\mathcal{O}^*(\alpha^{(v_{\max}+w_{\max})(n-\frac{1}{2}t)}) = \mathcal{O}(1.5002^{n-\frac{1}{2}t})$ time.

Let $b \in \mathcal{B}$ be a blue vertex in G' corresponding to a vertex $x \in X$ in G. Our procedure first applies inclusion/exclusion-based branching on b: in the optional branch, we remove b from G'; in the forbidden branch, we remove $N[b]$ from G'. Because $N[b]$ contains at least $t-1$ red vertices corresponding to b and its neighbours, the instance generated in the forbidden branch has measure at most $(n-1)v_{\max} + (n-t+1)w_{\max} < (v_{\max}+w_{\max})(n-\frac{1}{2}t)$ as $w_{\max} \ge v_{\max}$. In the optional branch, we branch on the next blue vertex corresponding to a vertex in X. We repeat this until all blue vertices corresponding to vertices in X are used. This gives a series of $\frac{1}{2}t$ instances of measure at most $(v_{\max}+w_{\max})(n-\frac{1}{2}t)$ and one instance of measure at most $v_{\max}(n-\frac{1}{2}t) + w_{\max}n$ in which all blue vertices corresponding to vertices in X are exhausted.

In this last instance, we branch on red vertices in G' corresponding to vertices in X in G. These vertices still have at least $\frac{1}{2}t-1$ neighbours in G', since only $\frac{1}{2}t$ blue vertices have been removed in the optional branches. Let $r \in \mathcal{R}$ be a red vertex corresponding to a vertex in X. When branching on r and taking the vertex in the solutions being counted, again a subproblem of measure at most $(v_{\max}+w_{\max}) \times (n-\frac{1}{2}t)$ is generated as $N[r]$ is removed. In the branch where r is discarded, we continue by branching on the next red vertex corresponding to a vertex in X. In this way, we again generate a series of $\frac{1}{2}t$ instances of measure at most $(v_{\max}+w_{\max}) \times (n-\frac{1}{2}t)$ and one instance in which all vertices to branch on are exhausted.

In the remaining instance, $\frac{1}{2}t$ blue vertices are removed and $\frac{1}{2}t$ red vertices are removed. This results in an instance that also has measure at most $(v_{\max}+w_{\max}) \times (n-\frac{1}{2}t)$ proving the lemma. $\square$

We now continue by giving the results on the different graph classes using Lemma 34. We start with *c-dense graphs*.

Definition 35 (c-Dense Graph) A graph $G=(V,E)$ is said to be c-dense if $|E| \ge cn^2$ where c is a constant with $0 < c < \frac{1}{2}$.

Gaspers et al. have given an $\mathcal{O}(1.5063^{(\frac{1}{2}+\frac{1}{2}\sqrt{1-2c})n})$-time algorithm for DOMINATING SET on c-dense graphs [41]. We give faster algorithms for DOMINATING SET and #DOMINATING SET on c-dense graphs below. A graphical comparison of both results can be found in Fig. 1.

Corollary 36 *There exists an $\mathcal{O}(1.5002^{(\frac{1}{4}+\frac{1}{4}\sqrt{9-16c})n})$-time algorithm for* #DOMINATING SET *on c-dense graphs.*

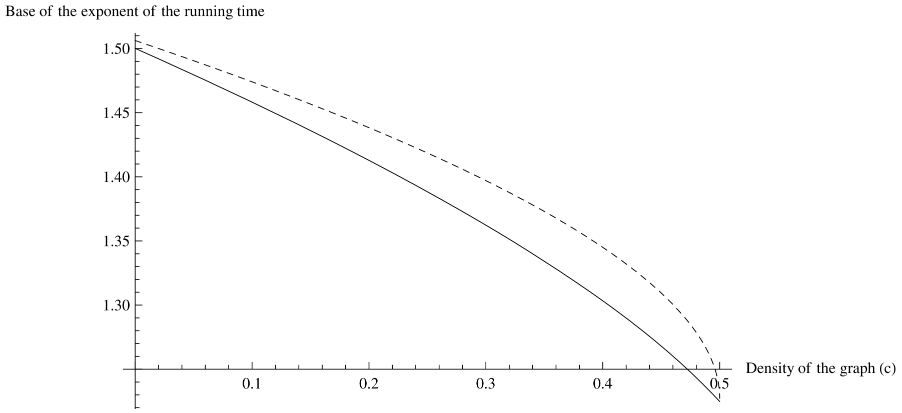

Fig. 1 Comparison of bounds on the running time on c-dense graphs. The *solid line* represents the base of the exponent of the upper bound on the running time of our algorithm. The *dashed line* represents the base of the exponent of the upper bounds from [41]

Proof By a counting argument in [41], any graph with sufficiently many edges has a set of high-degree vertices that allow application of Lemma 34 with parameter t. For c-dense graphs this occurs 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-$\mathcal{O}$ notation, we obtain a running time of $\mathcal{O}(1.5002^{(\frac{1}{4}+\frac{1}{4}\sqrt{9-16c})n})$. □

Corollary 36 gives the currently fastest algorithm for DOMINATING SET on c-dense graphs.

We proceed by giving faster algorithms on circle graphs, 4-chordal graphs, and weakly chordal graphs. Chordal graphs will be considered hereafter.

Definition 37 (Circle Graph) A *circle graph* is an intersection graph of chords in a circle: every vertex represents a chord, and vertices are adjacent if their corresponding chords intersect.

A *chordless cycle* in a graph G is a sequence of vertices $(v_1, v_2, \ldots, v_l, v_1)$ such that $G[\bigcup_{i=1}^{l}\{v_i\}]$ is an induced cycle in G, that is, the only edges of $G[\bigcup_{i=1}^{l}\{v_i\}]$ are the edges that form the cycle.

Definition 38 (Chordal Graph) A graph is *chordal* if it has no chordless cycle of length more than three.

Definition 39 (4-Chordal Graph) A graph is *4-chordal* if it has no chordless cycle of length more than four.

Definition 40 (Weakly Chordal Graph) A graph G is weakly chordal if both G and its complement are 4-chordal.

The following lemma is based on [41] and gives our running times on circle graphs, 4-chordal graphs, and weakly chordal graphs. We note that a graph class $\mathcal{G}$ is a hereditary class if all induced subgraphs of any graph $G \in \mathcal{G}$ are in $\mathcal{G}$ also.

Lemma 41 *Let $\mathcal{G}$ be a hereditary class of graphs such that all $G \in \mathcal{G}$ have the property that $tw(G) \leq c\Delta(G)$ and that a tree decomposition of G of width at most $c\Delta(G)$ can be computed in polynomial time. For any $t' \geq 1$, there exists an algorithm solving* #Dominating Set *in $\mathcal{O}(\max\{1.5002^{(1-\frac{1}{2}t')n}, 3^{(c+\frac{1}{2})t'n}\})$ for graphs $G \in \mathcal{G}$.*

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 34 with $t = t'n$ giving a running time of $\mathcal{O}(1.5002^{(1-\frac{1}{2}t')n})$.

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

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

Note that the first inequality is based on the fact that we can add all vertices in X to all bags of a tree decomposition of $G[V \setminus X]$: this increases its width by at most $|X|$.

Now, we can apply the $\mathcal{O}^*(3^k)$-time algorithm of Proposition 5. This gives the running time of $\mathcal{O}(3^{(c+\frac{1}{2})t'n})$. □

The running times follow from using the following values for c on the different graph classes.

Proposition 42 [41] *The following graph classes are hereditary graph classes with $tw(G) \leq c\Delta(G)$ for all $G \in \mathcal{G}$ using the indicated values of c:*

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

The corresponding tree decompositions can be computed in polynomial time.

Corollary 43 *There exist algorithms solving* #Dominating Set *on weakly chordal graphs in $\mathcal{O}(1.4590^n)$ time, on 4-chordal graphs in $\mathcal{O}(1.4700^n)$ time, and on circle graphs in $\mathcal{O}(1.4764^n)$ time.*

Proof Combine Lemma 41 and Proposition 42, and use the value of t' for which both running times in Lemma 41 are balanced. □

Our improvement for chordal graphs does not rely on the fact that we need fewer vertices of high degree, but is based on the result in Theorem 31 and the improvements on tree decompositions in [68].

It is well known that a chordal graph can be represented as a clique tree [44]. A *clique tree* T of a chordal graph G is a tree such that there exists a bijection between the nodes of the tree T and the maximal cliques in G and such that, for each vertex $v \in V$, all nodes in T corresponding to a clique that contains v form a connected subtree. A clique tree of a chordal graph is an optimal tree decomposition of G whose width equals the size of the largest clique in G minus one.

Proposition 44 ([41] + [68]; for details see [65]) *Given a clique tree a chordal graph $G = (V, E)$ of treewidth tw, the number of dominating sets of size κ, for each value $0 \leq \kappa \leq n$, can be computed in $\mathcal{O}^*(2^{tw})$ time.*

Corollary 45 *There exists an algorithm that solves* #DOMINATING SET *on chordal graphs in $\mathcal{O}(1.3687^n)$ time.*

Proof Compute a clique tree T of the chordal graph G; this can be done in polynomial time [64]. Find a largest clique C in G by considering the cliques corresponding to each node of T.

If $|C| \geq 0.452704n$, then there exists at least $0.452704n$ vertices of degree at least $0.452704n - 1$. In this case, we can apply Lemma 34 with $t = 0.452704n$; this solves given the instance in $\mathcal{O}(1.5002^{n-\frac{1}{2}0.452704n}) = \mathcal{O}(1.3687^n)$ time. Otherwise, $|C| < 0.452704n$. In this case, T is a tree decomposition of width at most $0.452704n$. Now, we solve the instance in $\mathcal{O}(2^{0.452704n}) = \mathcal{O}(1.3687^n)$ time using Proposition 44. □

8 Conclusion

In this paper, we have shown that the principle of inclusion/exclusion can be used as a branching rule in a branch-and-reduce algorithm. We combined the use of such an inclusion/exclusion-based branching rule with different series of reduction rules and standard (non-inclusion/exclusion-based) branching rules. This resulted in non-trivial branch-and-reduce algorithms that we analysed using measure and conquer. Besides this general approach, we also extended the inclusion/exclusion-based branching rule to the setting where not all requirements of a problem need to be satisfied. This resulted, amongst others, in the currently fastest polynomial-space and/or exponential-space algorithms for a large number of domination problems in graphs and the currently fastest polynomial-space and exponential-space algorithms for the well-studied parameterised problem k-SET SPLITTING.

Our approach has further applications. For example, Paulusma and van Rooij use inclusion/exclusion-based branching and measure and conquer to give faster algorithms for 2-DISJOINT CONNECTED SUBGRAPHS on a number of graph classes [59]. Also, we never considered running times for algorithms solving problems such as RED-BLUE DOMINATING SET when used to solve this problem itself instead of other problems such as DOMINATING SET. In [65], it is shown that Algorithm 4 can be used to solve this problem in $\mathcal{O}(1.2252^n)$ time and space. We omitted this result because it will probably be easy to improve this result using tools from the recent algorithm for DOMINATING SET by Iwata [47].

There are more techniques to prove good upper bounds on the running times of branch-and-reduce algorithms than measure and conquer. Examples are techniques based on average degrees in a graph (e.g., see [19]). Inclusion/exclusion-based branching algorithms can also be analysed by such means. The general idea of interpreting an inclusion/exclusion algorithm as branch-and-reduce algorithm is a nice approach to get better upper bounds on the running time of an inclusion/exclusion algorithm than the usual running times of $\mathcal{O}^*(2^n)$ corresponding to the 2^n terms of the inclusion/exclusion formula.

We note that any inclusion/exclusion algorithm can be turned into a branch-and-reduce algorithm without reduction rules. In this way, one can easily add reduction rules similar to those used in this paper to this algorithm. The resulting algorithm will possibly not have a faster worst-case running time, but will probably generate less subproblems on a given instance possibly improving its running time in practice.

Acknowledgements We thank Hans L. Bodlaender for his guidance and enthusiasm during this research.

References

1. Amini, O., Fomin, F.V., Saurabh, S.: Implicit branching and parameterized partial cover problems. J. Comput. Syst. Sci. **77**(6), 1159–1171 (2011)
2. Amini, O., Fomin, F.V., Saurabh, S.: Counting subgraphs via homomorphisms. SIAM J. Discrete Math. **26**(2), 695–717 (2012)
3. Andersson, G., Engebretsen, L.: Better approximation algorithms for SET SPLITTING and NOT-ALL-EQUAL SAT. Inf. Process. Lett. **65**(6), 305–311 (1998)
4. Bax, E.T.: Inclusion and exclusion algorithm for the Hamiltonian path problem. Inf. Process. Lett. **47**(4), 203–207 (1993)
5. Binkele-Raible, D., Fernau, H.: Enumerate & measure: improving parameter budget management. In: Raman, V., Saurabh, S. (eds.) 5th International Symposium on Parameterized and Exact Computation, IPEC 2010. Lecture Notes in Computer Science, vol. 6478, pp. 38–49. Springer, Berlin (2010)
6. Björklund, A.: Determinant sums for undirected Hamiltonicity. In: 51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, pp. 173–182. IEEE Computer Society, New York (2010)
7. Björklund, A.: Exact covers via determinants. In: Marion, J.-Y., Schwentick, T. (eds.) 27th International Symposium on Theoretical Aspects of Computer Science, STACS 2010. Leibniz International Proceedings in Informatics, vol. 3, pp. 95–106. Schloss Dagstuhl, Leibniz-Zentrum fuer Informatik (2010)
8. Björklund, A., Husfeldt, T.: Exact algorithms for exact satisfiability and number of perfect matchings. Algorithmica **52**(2), 226–249 (2008)
9. Björklund, A., Husfeldt, T., Kaski, P., Koivisto, M.: Fourier meets Möbius: fast subset convolution. In: Johnson, D.S., Feige, U. (eds.) 39th Annual ACM Symposium on Theory of Computing, STOC 2007, pp. 67–74. ACM Press, New York (2007)
10. Björklund, A., Husfeldt, T., Kaski, P., Koivisto, M.: Counting paths and packings in halves. In: Fiat, A., Sanders, P. (eds.) 17th Annual European Symposium on Algorithms, ESA 2009. Lecture Notes in Computer Science, vol. 5757, pp. 578–586. Springer, Berlin (2009)
11. Björklund, A., Husfeldt, T., Kaski, P., Koivisto, M.: Trimmed Moebius inversion and graphs of bounded degree. Theory Comput. Syst. **47**(3), 637–654 (2010)
12. Björklund, A., Husfeldt, T., Kaski, P., Koivisto, M.: Covering and packing in linear space. Inf. Process. Lett. **111**(21–22), 1033–1036 (2011)
13. Björklund, A., Husfeldt, T., Koivisto, M.: Set partitioning via inclusion-exclusion. SIAM J. Comput. **39**(2), 546–563 (2009)
14. Bodlaender, H.L.: A partial k-arboretum of graphs with bounded treewidth. Theor. Comput. Sci. **209**(1–2), 1–45 (1998)

15. Bodlaender, H.L., Koster, A.M.C.A.: Combinatorial optimization on graphs of bounded treewidth. Comput. J. **51**(3), 255–269 (2008)
16. Bourgeois, N., Croce, F.D., Escoffier, B., Paschos, V.T.: Algorithms for dominating clique problems. Theor. Comput. Sci. **459**, 77–88 (2012)
17. Bourgeois, N., Escoffier, B., Paschos, V.T.: Fast algorithms for min independent dominating set. In: Patt-Shamir, B., Ekim, T. (eds.) 17th International Colloquium Structural Information and Communication Complexity, SIROCCO 2010. Lecture Notes in Computer Science, vol. 6058, pp. 247–261. Springer, Berlin (2010)
18. Bourgeois, N., Escoffier, B., Paschos, V.T., van Rooij, J.M.M.: Maximum independent set in graphs of average degree at most three in $O(1.08537^n)$. In: Kratochvíl, J., Li, A., Fiala, J., Kolman, P. (eds.) 7th Annual Conference on Theory and Applications of Models of Computation, TAMC 2010. Lecture Notes in Computer Science, vol. 6108, pp. 373–384. Springer, Berlin (2010)
19. Bourgeois, N., Escoffier, B., Paschos, V.T., van Rooij, J.M.M.: Fast algorithms for max independent set. Algorithmica **62**(1–2), 382–415 (2012)
20. Chen, J., Lu, S.: Improved parameterized set splitting algorithms: a probabilistic approach. Algorithmica **54**(4), 472–489 (2009)
21. Cygan, M., Dell, H., Lokshtanov, D., Marx, D., Nederlof, J., Okamoto, Y., Paturi, R., Saurabh, S., Wahlström, M.: On problems as hard as CNF-SAT. In: IEEE Conference on Computational Complexity, pp. 74–84. IEEE, New York (2012)
22. Dehne, F.K.H.A., Fellows, M.R., Fernau, H., Prieto, E., Rosamond, F.A.: Nonblocker: parameterized algorithmics for minimum dominating set. In: Wiedermann, J., Tel, G., Pokorný, J., Bieliková, M., Stuller, J. (eds.) 32nd Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2006. Lecture Notes in Computer Science, vol. 3831, pp. 237–245. Springer, Berlin (2006)
23. Dehne, F.K.H.A., Fellows, M.R., Rosamond, F.A.: An FPT algorithm for set splitting. In: Bodlaender, H.L. (ed.) 29th International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2003. Lecture Notes in Computer Science, vol. 2880, pp. 180–191. Springer, Berlin (2003)
24. Dehne, F.K.H.A., Fellows, M.R., Rosamond, F.A., Shaw, P.: Greedy localization, iterative compression, modeled crown reductions: new FPT techniques, an improved algorithm for set splitting, and a novel $2k$ kernelization for vertex cover. In: Downey, R.G., Fellows, M.R., Dehne, F.K.H.A. (eds.) 1st International Workshop on Parameterized and Exact Computation, IWPEC 2004. Lecture Notes in Computer Science, vol. 3162, pp. 271–280. Springer, Berlin (2004)
25. Downey, R.G., Fellows, M.R.: Fixed-parameter tractability and completeness. Congr. Numer. **87**, 161–178 (1992)
26. Eppstein, D.: Quasiconvex analysis of multivariate recurrence equations for backtracking algorithms. ACM Trans. Algorithms **2**(4), 492–509 (2006)
27. Erdös, P.: On a combinatorial problem, I. Nord. Mat. Tidskrift **11**, 5–10 (1963)
28. Erdös, P.: On a combinatorial problem, II. Acta Math. Hung. **15**(3), 445–447 (1964)
29. Fernau, H., Kneis, J., Kratsch, D., Langer, A., Liedloff, M., Raible, D., Rossmanith, P.: An exact algorithm for the maximum leaf spanning tree problem. Theor. Comput. Sci. **412**(45), 6290–6302 (2011)
30. Fomin, F.V., Gaspers, S., Pyatkin, A.V., Razgon, I.: On the minimum feedback vertex set problem: Exact and enumeration algorithms. Algorithmica **52**(2), 293–307 (2008)
31. Fomin, F.V., Gaspers, S., Saurabh, S., Stepanov, A.A.: On two techniques of combining branching and treewidth. Algorithmica **54**(2), 181–207 (2009)
32. Fomin, F.V., Grandoni, F., Kratsch, D.: Solving connected dominating set faster than 2^n. Algorithmica **52**(2), 153–166 (2008)
33. Fomin, F.V., Grandoni, F., Kratsch, D.: A measure & conquer approach for the analysis of exact algorithms. J. ACM **56**(5) (2009)
34. Fomin, F.V., Grandoni, F., Pyatkin, A.V., Stepanov, A.A.: Combinatorial bounds via measure and conquer: bounding minimal dominating sets and applications. ACM Trans. Algorithms **5**(1) (2008)
35. Fomin, F.V., Grandoni, F., Pyatkin, A.V., Stepanov, A.A.: Combinatorial bounds via measure and conquer: bounding minimal dominating sets and applications. ACM Trans. Algorithms **5**(1) (2008)
36. Fomin, F.V., Kratsch, D.: Exact Exponential Algorithms. Texts in Theoretical Computer Science. Springer, Berlin (2010)
37. Fomin, F.V., Kratsch, D., Woeginger, G.J.: Exact (exponential) algorithms for the dominating set problem. In: Hromkovic, J., Nagl, M., Westfechtel, B. (eds.) 30th International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2004. Lecture Notes in Computer Science, vol. 3353, pp. 24–256. Springer, Berlin (2004)

38. Fomin, F.V., Lokshtanov, D., Raman, V., Saurabh, S.: Subexponential algorithms for partial cover problems. Inf. Process. Lett. **111**(16), 814–818 (2011)
39. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, New York (1979)
40. Gaspers, S., Kratsch, D., Liedloff, M.: On independent sets and bicliques in graphs. Algorithmica **62**(3–4), 637–658 (2012)
41. Gaspers, S., Kratsch, D., Liedloff, M., Todinca, I.: Exponential time algorithms for the minimum dominating set problem on some graph classes. ACM Trans. Algorithms **6**(1) (2009)
42. Gaspers, S., Liedloff, M.: A branch-and-reduce algorithm for finding a minimum independent dominating set. Discrete Math. Theor. Comput. Sci. **14**(1), 29–42 (2012)
43. Gaspers, S., Sorkin, G.B.: A universally fastest algorithm for max 2-Sat, max 2-CSP, and everything in between. J. Comput. Syst. Sci. **78**(1), 305–335 (2012)
44. Gavril, F.: The intersection graphs of subtrees in trees are exactly the chordal graphs. J. Comb. Theory, Ser. B **16**(1), 47–56 (1974)
45. Grandoni, F.: Exact algorithms for hard graph problems. PhD thesis, Department of Computer Science, Systems and Production, Universitá degli Studi di Roma “Tor Vergata”, Rome, Italy (2004)
46. Grandoni, F.: A note on the complexity of minimum dominating set. J. Discrete Algorithms **4**(2), 209–214 (2006)
47. Iwata, Y.: A faster algorithm for dominating set analyzed by the potential method. In: Marx, D., Rossmanith, P. (eds.) IPEC. Lecture Notes in Computer Science, vol. 7112, pp. 41–54. Springer, Berlin (2011)
48. Karp, R.M.: Dynamic programming meets the principle of inclusion-exclusion. Oper. Res. Lett. **1**(2), 49–51 (1982)
49. Kneis, J., Mölle, D., Richter, S., Rossmanith, P.: A bound on the pathwidth of sparse graphs with applications to exact algorithms. SIAM J. Discrete Math. **23**(1), 407–427 (2009)
50. Kneis, J., Mölle, D., Rossmanith, P.: Partial vs. complete domination: t-dominating set. In: van Leeuwen, J., Italiano, G.F., van der Hoek, W., Meinel, C., Sack, H., Plasil, F. (eds.) 33rd Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2007. Lecture Notes in Computer Science, vol. 4362, pp. 367–376. Springer, Berlin (2007)
51. Kohn, S., Gottlieb, A., Kohn, M.: A generating function approach to the traveling salesman problem. In: Proceedings of the 1977 Annual Conference of the ACM, pp. 294–300. ACM Press, New York (1977)
52. Koutis, I., Williams, R.: Limits and applications of group algebras for parameterized problems. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S.E., Thomas, W. (eds.) 36th International Colloquium on Automata, Languages and Programming (1), ICALP 2009. Lecture Notes in Computer Science, vol. 5555, pp. 653–664. Springer, Berlin (2009)
53. Liedloff, M.: Algorithmes exacts et exponentiels pour les problèmes NP-difficiles: domination, variantes et généralisation. PhD thesis, Laboratoire d’Informatique Théorique et Appliquée, Université Paul Verlaine, Metz, France (2007)
54. Lokshtanov, D., Saurabh, S.: Even faster algorithm for set splitting! In: Chen, J., Fomin, F.V. (eds.) 4th International Workshop on Parameterized and Exact Computation, IWPEC 2009. Lecture Notes in Computer Science, vol. 5917, pp. 288–299. Springer, Berlin (2009)
55. Lokshtanov, D., Sloper, C.: Fixed parameter set splitting, linear kernel and improved running time. In: Broersma, H., Johnson, M., Szeider, S. (eds.) 1st Algorithms and Complexity in Durham Workshop, ACiD 2005. Texts in Algorithmics, vol. 4, pp. 105–113. King’s College, London (2005)
56. Lovász, L.: Coverings and colorings of hypergraphs. Congr. Numer. **8**, 3–12 (1973)
57. Nederlof, J.: Space and time efficient structural improvements of dynamic programming algorithms. PhD thesis, Department of Informatics, University of Bergen, Bergen, Norway (2011)
58. Nederlof, J.: Fast polynomial-space algorithms using inclusion-exclusion. Algorithmica, 1–17 (2012)
59. Paulusma, D., van Rooij, J.M.M.: On partitioning a graph into two connected subgraphs. Theor. Comput. Sci. **412**(48), 6761–6769 (2011)
60. Radhakrishnan, J., Srinivasan, A.: Improved bounds and algorithms for hypergraph 2-coloring. Random Struct. Algorithms **16**(1), 4–32 (2000)
61. Riege, T., Rothe, J.: An exact 2.9416^n algorithm for the three domatic number problem. In: Jedrzejowicz, J., Szepietowski, A. (eds.) 30th International Symposium on Mathematical Foundations of Computer Science, MFCS 2005. Lecture Notes in Computer Science, vol. 3618, pp. 733–744. Springer, Berlin (2005)
62. Riege, T., Rothe, J., Spakowski, H., Yamamoto, M.: An improved exact algorithm for the domatic number problem. Inf. Process. Lett. **101**(3), 101–106 (2007)

63. Schiermeyer, I.: Efficiency in exponential time for domination-type problems. Discrete Appl. Math. **156**(17), 3291–3297 (2008)
64. Tarjan, R.E., Yannakakis, M.: Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. SIAM J. Comput. **13**(3), 566–579 (1984)
65. van Rooij, J.M.M.: Exact exponential-time algorithms for domination problems in graphs. PhD thesis, Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands (2011)
66. van Rooij, J.M.M., Bodlaender, H.L.: Exact algorithms for dominating set. Discrete Appl. Math. **159**(17), 2147–2164 (2011)
67. van Rooij, J.M.M., Bodlaender, H.L.: Exact algorithms for edge domination. Algorithmica **64**(4), 535–563 (2012)
68. van Rooij, J.M.M., Bodlaender, H.L., Rossmanith, P.: Dynamic programming on tree decompositions using generalised fast subset convolution. In: Fiat, A., Sanders, P. (eds.) 17th Annual European Symposium on Algorithms, ESA 2009. Lecture Notes in Computer Science, vol. 5757, pp. 566–577. Springer, Berlin (2009)
69. van Rooij, J.M.M., Nederlof, J., van Dijk, T.C.: Inclusion/exclusion meets measure and conquer. In: Fiat, A., Sanders, P. (eds.) 17th Annual European Symposium on Algorithms, ESA 2009. Lecture Notes in Computer Science, vol. 5757, pp. 554–565. Springer, Berlin (2009)
70. Zhang, J., Ye, Y., Han, Q.: Improved approximations for max set splitting and max NAE SAT. Discrete Appl. Math. **142**(1–3), 133–149 (2004)
71. Zwick, U.: Outward rotations: a tool for rounding solutions of semidefinite programming relaxations, with applications to MAX CUT and other problems. In: 31th Annual ACM Symposium on Theory of Computing, STOC 1999, pp. 679–687. ACM Press, New York (1999)