Utrecht University
Department of Information and Computer Science

Master of Science Thesis
Thesis number: INF/SCR-07-84

Inclusion exclusion for hard problems

by

Jesper Nederlof

First supervisor: Dr. H. L. Bodlaender
Second supervisor: Dr. G. Tel

Preface

One of the most important ways of getting more insight in $\mathcal{NP}$-complete problems is the comparison of worst-case running times of efficient algorithms solving them. Why can some problems be solved more efficiently than others?

In 2006, a surprisingly simple and efficient algorithm for one of the most famous $\mathcal{NP}$-complete problems was published: Björklund and Husfeldt, and independently Koivisto [BH06b, Koi06] gave an algorithm for the graph coloring problem with a worst-case running time bounded by $\mathcal{O}^*(2^n)$, where n is the number of nodes of the given graph. One year later, they showed that a strong generalization can be solved within the same time bound too [BHK06].

The technique they used, named inclusion exclusion (IE), is the subject of this master's thesis. For which other problems can the technique be used, how could one use additional techniques for obtaining algorithms with an even better worst-case running time, and could the technique be useful in any practical situation? In this thesis we will discuss these questions.

Moreover, we will discuss the concept of an IE-branch, which can be seen as the antipole of a normal branch, and provide some examples. Our main results are an $o(1.5^n)$ time algorithm for counting the number of dominating sets of size k and an IE-branch & reduce algorithm for graph coloring.

Jesper Nederlof
jespernederlof@hotmail.com
August 2008

Contents

Ik wil graag de volgende mensen bedanken voor hun bijdrage aan deze scriptie: **Hans Bodlaender** voor zijn flexibele begeleiding en waardevolle inzicht als ik met een idee langs kwam. **Ruben van der Zwaan** voor de talloze uitwisselingen van ideeën en suggesties, die vaak hielpen als ik even vast zat. **Thomas van Dijk** en **Johan van Rooij** voor de fijne samenwerking, maar ook voor de vele nuttige gesprekken over mijn scriptie hierbuiten, en hun enthousiasme dat mij extra gemotiveerd heeft. Hans, Thomas en **Gerard Tel** voor het nakijken van mijn scriptie, het vinden van fouten en geven van suggesties.

Voor de rest wil ik nog bedanken: **De mensen van ALGO-Lab**, vanwege inhoudelijke gesprekken danwel gezelligheid. **Mijn vrienden**, die voor afleiding zorgden als ik dat nodig had. En vooral: **mijn ouders en zus**, omdat ze er altijd voor me zijn.

Chapter 1

Exact exponential algorithms

Since their introduction by Cook in 1971, it is an open question how to tackle $\mathcal{NP}$-complete problems nicely. An important way of obtaining more insight in these problems is the study of exact exponential algorithms, which focuses on minimizing the worst-case running time. By means of comparison of such running times, we hope to clarify which problems are harder to solve than other ones, and which techniques do work better on which problems. Because $\mathcal{P}$ is probably not equal to $\mathcal{NP}$, the running times of our algorithms will be super-polynomial. Furthermore, if $T(n)$ is the running time of an algorithm we say:

$$T(n) \text{ is } \begin{cases} \text{sub-exponential} & \text{if } \exists c : T(n) \in \Theta^*(c^{o(n)}) \\ \text{exponential} & \text{if } \exists c : T(n) \in \Theta^*(c^n) \\ \text{super-exponential} & \text{otherwise} \end{cases}$$

using the notation $\Theta^*(e(n))$ for $\Theta(e(n) * p(n))$, where $p(n)$ is polynomial and $e(n)$ is superpolynomial in n. Furthermore, recall that $o(n) = \mathcal{O}(n) \setminus \Theta(n)$.

Although some special cases of $\mathcal{NP}$-complete problems can be solved in sub-exponential time, for example many problems on planar graphs, most of them are not known to be solvable that fast. Moreover, for some $\mathcal{NP}$-complete problems, only super-exponential algorithms are known. The Quadratic Assignment Problem is such a problem, which is considered as one of the 'hardest' $\mathcal{NP}$-complete problems, for more information see [Com03] or [Woe03]. Another one of these problems, the multicover problem, will be discussed briefly in Section 3.6. The remaining problems, problems for which the fastest known algorithms have exponential running times, are the main subject of this thesis.

Note that we study $T(n)$ where n is a classical measure of complexity, such as the number of nodes or edges of a graph or the number of given integers. The study of $T(n)$ where n is a less classical measure, such as the size of the subset to be found or the treewidth, is called parameterized complexity. Although this study has the same goal as ours, it will not be discussed in this thesis.

1.1 The notion of a problem

Let us begin with some standard notation, extended with some we need for our purposes.

Definition 1.1. *A problem* $\textsc{Prob}$ *is defined by a language* $L \subseteq \{0,1\}^*$*. Mostly,* $\textsc{Prob}$ *will refer to its* ***decision variant****, which is: given any parameter* k *and a measure function* $f : L \to \mathbb{N}$*, check whether there exists a word* $w \in L$ *such that* $f(w) \leq k$*. Moreover, for a problem we say its*

- ***optimization variant*** *is to determine the minimum* k *such that there exists a* $w \in L$ *with* $f(w) = k$*.*
- ***construction variant*** *is to construct a word* $w \in L$ *such that* $f(w) \leq k$*, for a given* k*.*
- ***counting variant****, denoted with* $\#\textsc{Prob}$*, is to compute* $|L|$*.*
- ***parameterized counting variant****, denoted with* $\#\textsc{k-Prob}$*, is to compute* $|\{w \in L \mid f(w) = k\}|$*.*

In this thesis, we will also use $\#\textsc{Prob}$ *and* $\#\textsc{k-Prob}$ *for the actual corresponding numbers. Similarly, we use* $\textsc{Prob}$ *for a word of* L*.*

Let us analyse the relationships between these variants. First we mention that the optimization variant can be solved by iteratively solving decision variants. Using binary search over the range of f, the worst-case number of calls will be logarithmic in the maximum of $f(w)$. Hence, the optimization variant can efficiently be reduced to the decision variant. If we look at the construction variant, the general reducibility to the decision variant is less clear. However, usually one could easily extend a decision algorithm to a construction algorithm bounded by the same time within a polynomial factor. Therefore, such extensions are left as exercise for the reader in this thesis. Because of these considerations, it is quite common to focus on decision variants. For instance, $\mathcal{NP}$-completeness is only defined on this variant.

For the latter two variants of Definition 1.1, the **counting problems**, a clear relation is that the decision variant can be solved with the parameterized counting variant. Moreover, the counting variant can be reduced to the parameterized counting variant. Because this thesis is strongly focused on counting problems, we will elaborate more on these in the following section.

1.2 Counting problems

Counting problems arise in a variety of fields, one of its most natural ones being estimating probabilities. For many of these applications, we aim to find formulas which can efficiently be computed. One of the most famous of these formulas is the Matrix-Tree Theorem, discussed in Section 2.4. Unfortunately, as we will explain now, for many counting problems such a formula is not likely to exist. We will use some simplified definitions which will do the job for our needs, and for a more in-depth discussion we refer to a standard book about computation complexity, for example 'Computational Complexity: A Modern Approach' [AB07].

First, we need the following:

Definition 1.2. *The complexity class $\#\mathcal{P}$ consists of all (parametrized) counting variants of decision problems in $\mathcal{NP}$. A reduction is said to be a* ***counting reduction*** *if it preserves the number of solutions. A problem is $\#\mathcal{P}$-complete if it is in $\#\mathcal{P}$ and every other $\#\mathcal{P}$ problem can be reduced to it by a polynomial-time counting reduction.*

For $\mathcal{NP}$-completeness, recall that the famous Cook-Levin theorem, which states that SATISFIABILITY is $\mathcal{NP}$-complete, is proven by showing a reduction from any problem in $\mathcal{NP}$ to SATISFIABILITY. Analogously, #SATISFIABILITY can be proven to be $\#\mathcal{P}$-complete, because the reduction already is a counting reduction. For more information about this, we refer to Chapter 9 of [AB07]. Unfortunately, for $\#\mathcal{P}$-complete problems, the following holds:

Theorem 1.3. *The existence of a polynomial algorithm for a $\#\mathcal{P}$-complete problem implies $\mathcal{P} = \mathcal{NP}$.*

Proof. Solving a $\#\mathcal{P}$-complete problem in polynomial time implies that all problems in $\#\mathcal{P}$ can be solved in polynomial time. For any problem PROB $\in \mathcal{NP}$, its variant #K-PROB is in $\#\mathcal{P}$. Because PROB can reduced to #K-PROB, we know $\mathcal{P} = \mathcal{NP}$. □

Obviously, one may not hope for finding a polynomial algorithm for a problem in $\#\mathcal{P}$ which is associated with a $\mathcal{NP}$-complete decision problem. However, some problems which are associated with a decision problem in $\mathcal{P}$, are also $\#\mathcal{P}$-complete. We will discuss the first known and most famous one, #PERFECT MATCHING: Given a graph $G = (V, E)$, a PERFECT MATCHING is a subset $M \subseteq E$ such that each node of V is in exactly one edge of M. This is well-known to be in $\mathcal{P}$. However, Valiant proved the following in [Val79]:

Theorem 1.4. #PERFECT MATCHING *is $\#\mathcal{P}$-complete, even when restricted to bipartite graphs.*

Actually Valiant proved a similar problem $\#\mathcal{P}$-complete, which is computing the **permanent**. We will only give a definition of the permanent, and show how

#PERFECT MATCHING can be reduced to it. Given a matrix $A = [a_{ij}]_{1\leq i,j\leq n}$, its permanent is defined as follows:

$$\text{perm}(A) = \sum_{\pi\in\Pi_n} \prod_{i=1}^{n} a_{i,\pi(i)} \tag{1.1}$$

where Π_n are all permutations of $\{1,\dots,n\}$. Now, suppose we have a bipartite graph B with equal-sized partitions $y_1,\dots,y_n$ and $z_1,\dots,z_n$, and edge set E. We construct the matrix $A = [a_{ij}]_{1\leq i,j\leq n}$:

$$a_{ij} = \begin{cases} 1 & \text{if } (y_i, z_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Because the product $\prod_{i=1}^{n} a_{i,\pi(i)}$ is 1 if π is a valid perfect matching, and else 0, it follows that #PERFECT MATCHING of B is equal to perm(A). For the proof of the $\#\mathcal{P}$-completeness of computing perm(A), we refer to [Val79].

1.3 Techniques

In this section we briefly give an overview of some important techniques for designing exact decision algorithms, that will be used throughout this thesis. For each technique we will discuss the extension to counting algorithms.

1.3.1 Exhaustive search

A very basic technique is **exhaustive branching**, which iteratively takes a decision variable, and goes in recursion with each possible assignment of this variable. If we would interpret this process as building a tree, all leaves simply correspond to words of the language $\{0,1\}^*$. Because of this, we can achieve the same by **exhaustive enumeration**, i.e. enumerate all leaves, and check if they actually are a solution. Note that in this technique, automatically the number of solutions is counted.

1.3.2 Dynamic programming on subsets

Suppose we want to compute a function $f(V)$ which is defined in terms $f(X)$, where $X \subset V$. Such a function can efficiently be computed using dynamic programming, by iteratively computing $f(X)$ for all subsets of increasing size: start with $f(\emptyset)$, then compute $f(X)$ for all subset of size 1, and proceed until $f(V)$ is computed.

As an example we give the classical Held-Karp algorithm for the TRAVELING SALESMAN PROBLEM (TSP) from [HK62]. Given a graph with nodes V and a distance function $V \times V \to \mathbb{N}$, the TSP is to find a cycle that contains all nodes exactly once and minimizes its total distance. Arbitrarily choose a vertex s. Define $O(R,t)$, for any $t \in V$, $t \neq s$ and $R \subseteq V \setminus \{s,t\}$, as the minimum total distance among all simple paths from s to t that visit vertices $R \cup \{s,t\}$ and no other. Note that $O(V \setminus \{s,t\}, t) + d(t,s)$ is the distance of the best cycle that has node t before node s, and we want to compute the minimum of $O(V \setminus \{s,t\}, t) + d(t,s)$ over all $t \in V \setminus \{s\}$. Now we can obtain an algorithm that uses $\mathcal{O}^*(2^n)$ time and space[1], using dynamic programming in combination with the following recurrence:

$$O(R,t) = \begin{cases} d(s,t) & \text{if } R = \emptyset \\ \min_{v \in R} d(v,t) + O(R \setminus \{v\}, v) & \text{otherwise} \end{cases} \tag{1.2}$$

Note that this technique can be extended for computing counting variants too.

1.3.3 Branch & reduce

As its name already suggests, a branch & reduce-algorithm is obtained by extending an exhaustive branching algorithm with **reduction rules**. A reduction rule converts an instance of a problem to an easier one, if it is applicable.

For illustration, we discuss and analyse a very simple algorithm for Independent Set. Given a graph $G = (V,E)$ and an integer k, the decision variant of Independent Set is to find a subset $I \subseteq V$ with $|I| \geq k$ such that for each $u,v \in I : (u,v) \notin E$. We use the following reduction rule: if there exists a vertex v with 0 neighbors, we remove it from the graph and lower k with one. This is valid because each independent set which does not contain v, can always be converted to an equal-sized one which does contain v. If such a vertex does not exist, we choose a node w and use the following branching:

- **discard** w, i.e. we do not include it in I, proceed with the graph obtained by removing w.
- **take** w, i.e. we do include it in I, proceed with the graph obtained by removing w and all its neighbors.

Observe that w has at least one neighbor. Because of this, the running time $T(n)$ of the above algorithm is upper-bounded as follows, using $n = |V|$:

$$T(n) \leq T(n-1) + T(n-2)$$

which implies our algorithm has a running time bounded by $\mathcal{O}(1.62^n)$. Note that the values of $T(1)$ and $T(2)$ do not matter for this time bound.

This technique can not always be used for computing counting variants: sometimes, reduction rules are used which make a choice based on knowledge that there always exists at least one (optimal) solution which is not thrown away. Obviously, any algorithm containing such reasoning does not count all solutions.

[1]Similar to Θ^*, the $\mathcal{O}^*$ notation suppresses polynomial factors.

1.3.4 Split and List

The following technique is due to Williams [Wil05], who named it the **split and list** approach. In a nutshell, we **split** a set in 3 subsets and **list** the power sets of these subsets. On this structure we do our computation, which can be done faster than before if we use matrix multiplication.

We will illustrate the technique more precisely with an example, obtaining an algorithm for $\#2\text{-SAT}$ which is also due to Williams. First we need the following terminology:

Definition 1.5. *Given a set of n boolean variables B, a **literal** is a variable b or its negation $\overline{b}$, with $b \in B$. A **clause** is a set of literals, and a **Conjuctive Normal Form (CNF) expression** is a set of clauses. An **assignment** $X \subseteq B$ is said to **satisfy** a clause if the clause contains a literal b such that $b \in X$ or it contains a literal $\overline{b}$ with $b \notin X$. X **satisfies** a CNF expression if it satisfies all clauses. The problem $\#2\text{-SAT}$ is, given a CNF formula where each clause consists of at most 2 literals, to compute the number of assignments that satisfy this formula.*

We mention that, altough $2\text{-SAT} \in \mathcal{P}$, $\#2\text{-SAT}$ is $\#\mathcal{P}$-complete. We will obtain an exponential algorithm, which is as follows: Given any instance of $\#2\text{-SAT}$, arbitrarily partition the set of variables B in 3 sets B_0, B_1 and B_2. Construct the following graph $G = (V, E)$: for each subset of B_0, B_1 or B_2 we create a node in V. If $u \subseteq B_i$ and $v \subseteq B_j$, then $(u, v) \in E$ if and only if

- $i \neq j$
- The assignment $u \cup v$ satisfies all clauses which only contain variables of B_i and B_j

Let $\#\textsc{Triangles}$ be the number of triangles in G, i.e. the number of cycles of 3 edges. Now the number of satisfying assignments is equal to $\#\textsc{Triangles}$, and we use the following lemma to compute $\#\textsc{Triangles}$ faster then the trivial $|V|^3$.

Lemma 1.6 ([Wil05]). *If there is an algorithm for multiplying $n \times n$-sized matrices, with running time $\mathcal{O}(n^\omega)$, $\#\textsc{Triangles}$ of a graph $G = (V, E)$ can be computed in $\mathcal{O}(n^\omega)$ time too.*

Proof. If A is the adjacency matrix of G, and $A' = A^3$ with $A' = [a_{uv}]_{u,v \in V}$, then the entry a_{uv} is 1 if and only if there is a path of exactly length 3 from node u to node v. Because of this, a_{vv} is twice the number of triangles containing v. We conclude with:

$$\#\textsc{Triangles} = \frac{\sum_{v \in V} a_{vv}}{6}$$

□

Now, note that G has $3 * 2^{n/3}$ nodes and at most $3 * 2^{2n/3}$ edges. Using the fastest known algorithm for matrix multiplication (the Coppersmith-Winograd algorithm), we have $\omega = 2.376$. Hence, we can compute $\#\textsc{Triangles}$ in $\mathcal{O}^*(2^{\omega n/3}) = \mathcal{O}^*(1.7314^n)$ time. This also gives the value of $\#2\text{-SAT}$.

In the same year of its publication, the running time of this algorithm was improved by Dahllof et al. in [DJW05] and Furer et al. in [FK05], using branch & reduce. However, we will see some applications of split and list which can not be improved easily in similar manner. Furthermore, in this technique, automatically all solutions are counted.

1.4 Outline thesis

This remainder of this thesis is organized as follows: In Chapter 2 we introduce inclusion exclusion and IE-branching. Furthermore, we give a proof of a famous theorem to count the number of spanning trees in polynomial time, using inclusion-exclusion. We also give a simple combinatorial interpretation of the theorem, which suggests some extensions.

In Chapter 3, we provide some applications of inclusion exclusion. We extend Karp's scheduling algorithm such that it is much more efficient for most instances. Furthermore, we give the first non-trivial algorithm for the multicovering problem (which for example contains multicoloring), and propose the first $\mathcal{O}^*(2^n)$ algorithm for Cluster editing.

In Chapter 4 we give an adjusted version of abstract tubes, and introduce the important concept of **IE-branch & reduce**, and discuss how it can be combined with recent approaches of Björklund et al. to compute the simplified problem fast, in the context of cover formulations. As an example we give a branch & reduce algorithm for graph coloring.

In Chapter 5, we do a case study, applying IE-branching to the well-studied dominating set problem. Although we compute the counting variant, our algorithm also solves the decision variant faster then the most efficient decision algorithm.

Chapter 2

Inclusion exclusion

In this chapter we give an introduction to the principle of inclusion exclusion. In Section 2.1, we will give 2 examples and provide some important terminology. After this, we will present 2 significantly different interpretations of the principle.

The first one is the algebraic interpretation in Section 2.2. We give a simplified version which suffices for our purposes. A more in-depth discussion on this subject might be useful, but unfortunately this is beyond the scope of this thesis. The study will be briefly continued in Section 4.1.

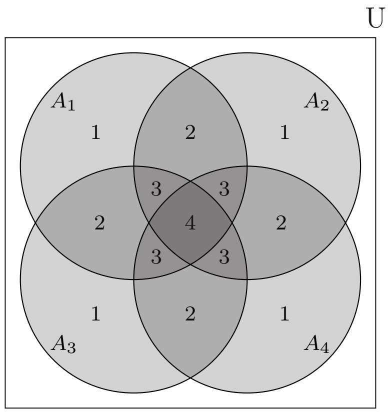
The second one is the combinatorial interpretation in Section 2.3. Here we introduce the important notion of an IE-branch, which is intuitive and appears to be combined easily with the technique of Section 1.3.3.

In Section 2.4 we will discuss one of the few known non-trivial polynomial time counting algorithms. This is interesting because it could inspire for other polynomial time algorithms. We will first give a proof of a famous theorem, using inclusion exclusion. After that, we will give a corresponding combinatorial algorithm on graphs.

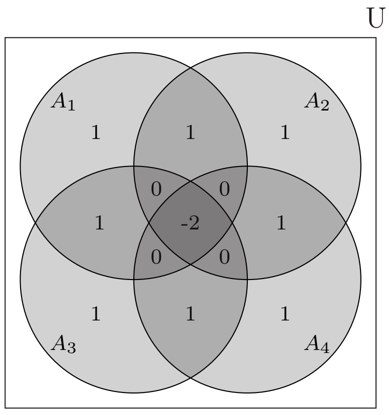
We will conclude this chapter with a brief discussion of the role of inclusion exclusion in the research area of designing algorithms.

2.1 Introduction

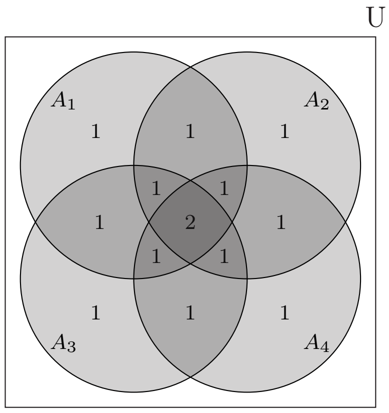
In a nutshell, inclusion exclusion is a way of computing a quantity through computing its complement with respect to some universe of known size. In many applications, computing the size of this complement appears to be a lot easier. Let us start with a nice visual example:



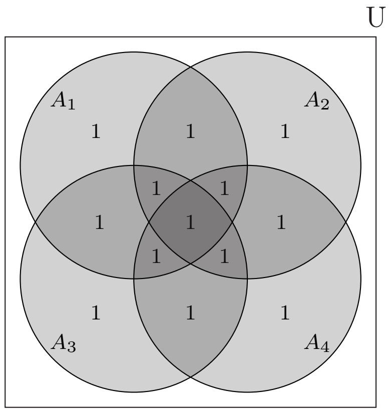
(a) $\sum_i |A_i|$



(b) - $\sum_{i<j} |A_i \cap A_j|$



(c) + $\sum_{i<j<k} |A_i \cap A_j \cap A_k|$



(d) - $\sum_{i<j<k<l} |A_i \cap A_j \cap A_k \cap A_l|$

Figure 2.1: An example of an inclusion-exclusion formula

2.1.1 Five sets in a Venn-diagram

Suppose we are given 5 subsets $A_1, \ldots, A_5$ of a set U, illustrated in Venn-diagram Figure 2.1. For notational ease, assume $A_i = \emptyset$ for $i < 1$ and $i > 5$. We want to compute $|\bigcup_i A_i|$, without using any union operator.

First, in (a), we sum over all subsets, counting elements of frequency i exactly i times. To compensate, we subtract the sizes of the intersections between each pair of sets in (b), subtracting elements of frequency i exactly $\frac{i*(i-1)}{2}$ times. After adding all intersections of 3 sets and subtracting the elements in all 4 sets, every element in one of the sets is counted exactly once. Concluding, we obtain the following identity:

$$|\bigcup_i A_i| = \sum_i |A_i| - \sum_{i<j} |A_i \cap A_j| + \sum_{i<j<k} |A_i \cap A_j \cap A_k| - \sum_{i<j<k<l} |A_i \cap A_j \cap A_k \cap A_l|$$

2.1.2 IE-terminology

The approach of the example of above, can used in many different settings. Moreover, as we shall see, for a given problem, there does not exist one distinct way of applying the technique. Therefore, we will extensively use the following terminology:

Definition 2.1. *In the context of inclusion exclusion for computing $\#\textsc{Prob}$ defined by a language L, we use the following notions and definitions:*

- *U is a superset of L, called the **universe**.*
- *A **requirement** A is a subset of U, whose elements are said to **meet** A.*
- *$\mathcal{A} = \{A_i\}_{i \in R}$ denotes the set of requirements, indexed with the **requirement space** R. An element $u \in U$ is said to meet $\mathcal{A}$ if it meets all of its requirements.*
- *The **complement** of a requirement $\overline{A} = U \setminus A$*
- *$\bigcap_{i \in \emptyset} A_i = U$.*
- *The **simplified problem** is to compute $|\bigcap_{i \in X} A_i|$, for a given $X \subseteq R$.*

2.1.3 Counting derangements

A classical exercise for inclusion exclusion is counting derangements. Given a requirement space R, a derangement is a permutation $\pi : R \to R$ such that for all $i \in R : \pi(i) \neq i$. Define the universe U to be all permutations of R, and the requirements A_i are all permutations π such that $\pi(i) = i$. Note that $|A_i| = (n-1)!$

Like the previous example, we compute $\sum_i |A_i| = \sum_i (n-1)!$, counting permutations which map i values into themselves i times. To compensate, we subtract $\sum_{i<j}(n-2)!$, achieving that permutations mapping no more than two elements into themselves are counted exactly once. If we keep going on, we get the following equality, of which the correctness will be proven later for a more general case.

$$|\bigcup_{i \in R} A_i| = \sum_{\substack{X \subseteq R \\ X \neq \emptyset}} (-1)^{|X|-1}(n - |X|)! \tag{2.1}$$

Observe that $|U| - |\bigcup_{i \in R} A_i|$ is the number of permutations without fixed elements, i.e. the number of derangements. Hence, if we denote $d(n)$ for the number of derangements of a n-sized set R, Equation 2.1, can be rewritten into

$$d(n) = \sum_{X \subseteq R} (-1)^{|X|}(n - |X|)!$$

Note that it is not efficient to compute this formula directly, since we have to sum over all 2^n subsets. Fortunately all subsets of the same size are **symmetric**, that is, they contribute exactly the same to the summation. Because of this the contribution of all i-sized subsets is $(-1)^i\binom{n}{i}(n-1)!$, which suggests the following much more efficiently computable formula:

$$d(n) = \sum_{i=0}^{n}(-1)^i\binom{n}{i}(n-1)!$$

2.2 Algebraic interpretation

In this section we give an algebraic interpretation of inclusion exclusion. More specificaly, we prove a generalization of Equation 2.1, using a basic concept from topology.

Definition 2.2. *A* ***sumspace*** *on a requirement space R is a family of non-empty subsets. A* ***simple sumspace*** *$\mathcal{S}$ is sumspace which is closed under inclusion, that is, if $X \in \mathcal{S}$ and $\emptyset \subset X' \subseteq X$, then $X' \in \mathcal{S}$.*

Note that a simple sumspace $\mathcal{S}$ is uniquely defined by its maximal sets $\mathcal{M}$, i.e. the sets which are not a subset of other sets. We say $\mathcal{S}$ is induced by $\mathcal{M}$. For the more interested reader, we mention that the notion of simple sumspace is called 'abstract simplicial complex' in mathematics.

Definition 2.3. *Given a sumspace $\mathcal{S}$ the* ***Euler characteristic*** *$\chi(\mathcal{S})$ is defined as:*

$$\chi(\mathcal{S}) = \sum_{X\in\mathcal{S}}(-1)^{|X|-1}$$

In this section, we only need the following property of the Euler characteristic:

Lemma 2.4. *Let $\mathcal{S}$ be the simple sumspace induced by a set S, then:*

$$\chi(\mathcal{S}) = \begin{cases} 0 & \text{if } S=\emptyset \\ 1 & \text{otherwise} \end{cases}$$

Proof. First, we rewrite the left-hand side:

$$\chi(\mathcal{S}) = \sum_{i=1}^{|S|}\binom{|S|}{i}(-1)^{i-1} = 1 - \sum_{i=0}^{|S|}\binom{|S|}{i}(-1)^i$$

Now substituting $S=\emptyset$ immediately boils down to 0, while using $S\neq\emptyset$ gives 1, because $\sum_{i=0}^{|S|}\binom{|S|}{i}(-1)^i = 0$ by the binomial theorem. □

The latter equality in the proof can be also be proven with the following bijection between even and odd subsets of a non-empty set S: Choose an element of v, and given an $X \subseteq S$, change v's membership of X.

With Lemma 2.4, we are ready to prove the inclusion exclusion identity:

Theorem 2.5.

$$|\bigcup_{i=1}^{n} A_i| = \sum_{\substack{X \subseteq R \\ X \neq \emptyset}} (-1)^{|X|-1} |\bigcap_{i \in X} A_i|$$

Proof. For each $u \in U$, we use $R[u] = \{i \in R \mid u \in A_i\}$ for all requirements which are met by u. Furthermore, we define $\mathcal{S}[u] = \{I \in \mathcal{S} \mid I \subseteq R[u]\}$, i.e. all subsets of requirements in R which are met by u . Now, we rewrite the right-hand side of the equation:

$$|\bigcup_{i=1}^{n} A_i| = \sum_{u \in U} \chi(\mathcal{S}[u])$$

and the theorem follows from Lemma 2.4. $\square$

Actually, the following variant of Theorem 2.5 is more usual:

Corollary 2.6.

$$|\bigcap_{i=1}^{n} \overline{A_i}| = \sum_{X \subseteq R} (-1)^{|X|} |\bigcap_{i \in X} A_i|$$

Proof. We use the equation of Theorem 2.5, multiply both sides with -1, and after that, add $|U|$. We obtain the equation from the corollary. $\square$

2.3 Combinatorial interpretation

In this section we will introduce the concept of Inclusion Exclusion-branching (IE-branching). Though the technique is quite fundamental, there are barely any publications using it. We use a formalization which is due to Bax [Bax96].

Definition 2.7. *Given a universe U and two sets of requirements $\mathcal{A} = \{A_i\}_{i \in R}$ and $\mathcal{B} = \{\ \overline{A_j}\ \}_{j \in S}$, where $A_i \subseteq$, for each i, we define:*

$$N(R, S) = |(\bigcap_{i \in R,} A_i) \cap (\bigcap_{j \in S,} \overline{A_j})|$$

In other words: $N(R, S)$ is the number of elements of U that satisfy both $\mathcal{A}$ and $\mathcal{B}$. Note that, due to Definition 2.1, $N(\emptyset, \emptyset) = |U|$. The definition is an extension of the notion $N(S)$ of Karp in [Kar82]. However, the use of the extra parameter R allows the following recurrence:

Theorem 2.8. *For all elements $v \in R$:*

$$N(R, S) = N(R \setminus \{v\}, S) - N(R \setminus \{v\}, S \cup \{v\})$$

Proof. $N(R \setminus \{v\}, S)$ is the number of elements which satisfy $\mathcal{A} \setminus A_v$ and $\mathcal{B}$. Subtracting number $N(R \setminus \{v\}, S \cup \{v\})$ of such elements that also satisfy $\overline{A_v}$, leaves us with the number of elements that meet both $\mathcal{A}$ and $\mathcal{B}$. $\square$

Any application of this theorem is said to be an **IE-branch**. In the context of counting algorithms, a classical branch is on something which could be taken, and if we simplify things a bit, it uses the following equation:

$$Optional = Required + Forbidden$$

Similarly, an IE-Branch is on something which is required and hence uses the following equation:

$$Required = Optional - Forbidden$$

Similarly to exhaustive branching, any application of Corollary 2.5 would be exhaustive IE-branching: Suppose if we want to compute $N(R, \emptyset)$. If we keep rewriting this using Theorem 2.8, until $R = \emptyset$, we arrive at Corollary 2.6.

Finding an efficient exhaustive branching algorithm is not always trivial. As will appear, this is certainly also the case with exhaustive IE-branching. We mention that it is possible to interleave both branching strategies and for an example we refer to Chapter 5.

2.4 Kirchhoff's Matrix Tree Theorem

In 1847, Gustav Kirchhoff published a theorem which relates the number of rooted spanning trees of a directed graph to the determinant of a variant of its adjacency matrix. First, we will prove this theorem by showing that the determinant of this matrix is equivalent to an inclusion exclusion formula. After the counting of the derangements, this is the second inclusion exlusion formula which can be computed efficiently. After this, we will also give a more straight forward proof which uses Gauss elimination.

2.4.1 A proof with inclusion exclusion

We assume a digraph $D = (V, A)$, and a **root** $r \in V$ are given. We abbreviate $V' = V \setminus \{r\}$. A rooted spanning tree of D with root r is an arc set $X \subseteq A$ such that each $v \in V'$ has one path to r. In this subsection we will prove the following:

Theorem 2.9. *Write $d^+(v)$ for the out degree of v. The* ***reduced Laplacian matrix*** *L of D is defined as $L = [l_{vw}]_{v,w \in V'}$, with:*

$$l_{vw} = \begin{cases} d^+(v) & \text{if } v = w \\ -1 & \text{if } v \neq w \text{ and } (v,w) \in A \\ 0 & \text{else} \end{cases}$$

Now, the number of spanning trees rooted at r of D is equal to the determinant $|L|$.

We will rewrite $|L|$ into a valid inclusion exclusion formulation. Our starting point is the following definition of the determinant:

$$|L| = \sum_{\pi \in \Pi_{V'}} sgn(\pi) \prod_{v \in V'} L_{v\pi(v)} \tag{2.2}$$

where $\Pi_{V'}$ are all permutations of V', and $sgn(\pi)$ is 1 if π is an even permutation, that is, it is obtained by an even number of interchangements, and -1 else.

We distinguish the permutations of $\Pi_{V'}$ from each other based on their **fixed part** F, which are all elements mapped into themselves. If we write $\overline{F}$ for the non-fixed part $V' \setminus F$ and $\Delta_{\overline{F}}$ for all derangements of $\overline{F}$, we obtain the following:

$$\sum_{F \subseteq V'} \left(\sum_{\delta \in \Delta_{\overline{F}}} sgn(\delta) \prod_{v \in \overline{F}} L_{v\delta(v)} \right) \prod_{v \in F} L_{vv} \tag{2.3}$$

Now, we obtain a combinatorial insight of this equation for a given fixed part:

Lemma 2.10. *A **cycle partition** is a set of cycles of D such that each node is in exactly one cycle. A cycle partition is called **even (odd)** if it consists of an even (odd) number of cycles. Given any non-fixed part $\overline{F}$,*

$$\sum_{\delta \in \Delta_{\overline{F}}} sgn(\delta) \prod_{v \in \overline{F}} L_{v\delta(v)} \tag{2.4}$$

is the number of even cycle partitions minus the number of odd cycle partitions of $\overline{F}$.

Proof. First notice that if $\prod_{v \in \overline{F}} L_{v\delta(v)} \neq 0$, the arc set $\{ \ (v, \delta(v)) \mid v \in \overline{F} \ \}$ is a cycle partition of $\overline{F}$. Therefore, we only have to look at derangements corresponding to cycle partitions.

A cycle σ with vertices $\sigma_1, \ldots, \sigma_k$ can be obtained by iteratively interchange σ_i with σ_{i+1} for $1 \leq i < k$. Hence it will inverse the sign of $sgn(\delta)$ if and only if it is even. However, such a cycle also contains k edges, so the sign will be multiplied k times by -1, because L'_{ij} is negative. Because of this every cycle, even or odd, will inverse the sign of its cycle partition. Now the lemma follows from the fact that $(-1)^k$ is 1 if k is even, and is -1 if k is odd. □

Now we will use this lemma to rewrite Equation 2.3 into an inclusion exclusion expression, and show that the latter indeed counts the number of spanning trees. To this end, we first need the following definition:

Definition 2.11. *A **parent assignment** is a subset $P \subseteq A$ such that for each $v \in V'$ there is one $w \in V$ such that $(v, w) \in P$, and there is no $w \in V$ such that $(r, w) \in P$.*

Lemma 2.12. *Let Σ be the set of cycles consisting of at least 2 arcs, $\Sigma \subset 2^A$. Moreover, given a set of cycles $\mathcal{X} \subseteq \Sigma$, define $c(\mathcal{X})$ as the number of parent assigments P such that for all cycles $\sigma \in \mathcal{X}$, $\sigma \subseteq P$. Now, Equation 2.3 is equal to*

$$\sum_{\mathcal{X} \subseteq \Sigma} (-1)^{|\mathcal{X}|} c(\mathcal{X}) \tag{2.5}$$

Proof. In a parent assignment all nodes can be in at most one cycle, so $c(\mathcal{X}) = 0$ if $\mathcal{X}$ consists of 2 cycles share that a node. Therefore, we assume that $\mathcal{X}$ consists of disjoint cycles. Thus, $\mathcal{X}$ is a cycle partition of exactly one node set $\overline{F}$. By Lemma 2.10, we know that the contribution of $\mathcal{X}$ to Equation 2.3 is $\prod_{v \in F} L_{vv}$ if $\mathcal{X}$ consists of an even number of cycles and otherwise, its contribution is $-\prod_{v \in F} L_{vv}$. Looking at Equation 2.5, the contribution of $\mathcal{X}$ is exactly the same, because for all $v \in F$ every outgoing edge can be chosen, which results in $\prod_{v \in F} d^+(v)$ possibilities. □

Now we are finally ready to prove the Matrix Tree Theorem:

Proof of Theorem 2.9. Define U as the set of all parent assignments. For each cycle σ, A_σ are all $P \in U$ such that $\sigma \subseteq P$. Now we have that $c(\mathcal{X}) = |\bigcap_{\sigma \in \mathcal{X}} A_\sigma|$ and the number of spanning trees is equal to $|\bigcap_{\sigma \in \mathcal{X}} \overline{A_\sigma}|$. Hence Equation 2.5 is equal to the number of spanning trees of D, by the inclusion exclusion identity. Due to Lemma 2.12 this is equal to the determinant $|L|$. □

We mention that the number of unrooted spanning trees of an undirected graph can also be computed the same way, since every node can be seen as root of a tree.

2.4.2 Gauss elimination on the reduced Laplacian

Keeping in mind that Gauss elimination has the determinant of the matrix as its invariant, the following question arises: Is there a combinatorial interpretation of Gauss elimination on the reduced Laplacian? In this section we sketch a proof of a generalization of Theorem 2.9, using a more combinatorial interpretation:

Theorem 2.13. *Given a directed multigraph D with root r, nodeset $V = \{v_1, \ldots, v_n\} \cup \{r\}$ and edge multiplicity function $m : (V' \times V) \to \mathbb{Q}$[1], we define its reduced laplacian $L = [l_{ij}]_{v_i, v_j \in V'}$ as:*

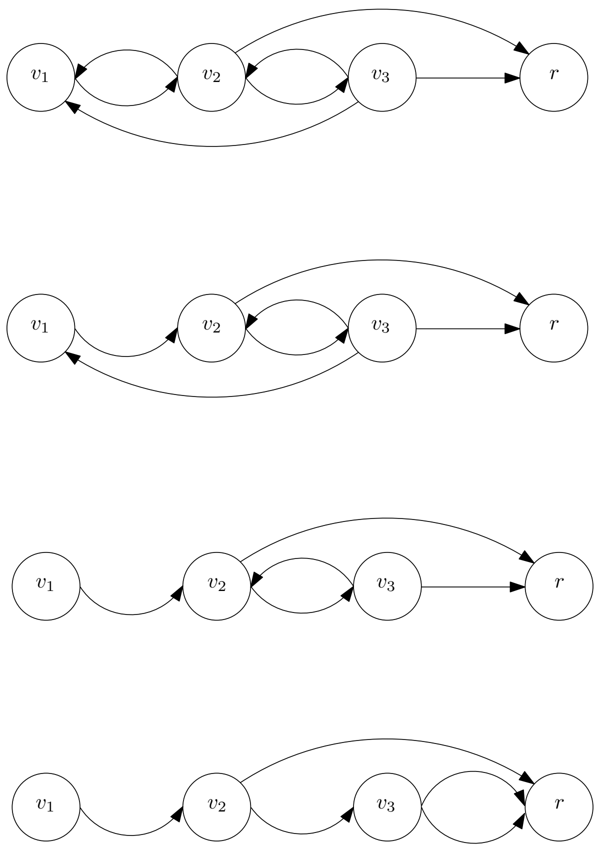
$$l_{ij} = \begin{cases} \sum_{u \in V} m(v_i, u) & \text{if } v_i = v_j \\ -m(v_i, v_j) & \text{else} \end{cases}$$

*Define the **multiplicity** of a spanning tree is the product of the multiplicity of its arcs. Now, the sum of the multiplicity of all spanning trees of D is equal to $|L|$.*

First suppose that L is an upper triangle matrix. That is, all elements below the diagonal are zero. Then, G is a directed acyclic multigraph. Observe that every parent assignment is a spanning tree since it does not contain any cycles. Now the number of parent assignments, with respect to m, is $\prod_{i \in V'} L_{ii}$, which is equal to $|L|$.

If L is not upper triangle, Gauss elimination performs an **elementary row operation**: Take a row and add it a constant times to another row. Note that Gauss elimination has the determinant as variant and makes a matrix upper triangle with elemenary row operations. Hence, for proving Theorem 2.13, it only remains to proof the following lemma:

[1]$\mathbb{Q}$ *denotes all numbers which be written as a ratio of 2 integers*



$$\begin{array}{c|ccc} & v_1 & v_2 & v_3 \\ \hline v_1 & 1 & -1 & 0 \\ v_2 & -1 & 3 & -1 \\ v_3 & -1 & -1 & 3 \end{array}$$

$$\begin{array}{c|ccc} & v_1 & v_2 & v_3 \\ \hline v_1 & 1 & -1 & 0 \\ v_2 & 0 & 2 & -1 \\ v_3 & -1 & -1 & 3 \end{array}$$

$$\begin{array}{c|ccc} & v_1 & v_2 & v_3 \\ \hline v_1 & 1 & -1 & 0 \\ v_2 & 0 & 2 & -1 \\ v_3 & 0 & -2 & 3 \end{array}$$

$$\begin{array}{c|ccc} & v_1 & v_2 & v_3 \\ \hline v_1 & 1 & -1 & 0 \\ v_2 & 0 & 2 & -1 \\ v_3 & 0 & 0 & 2 \end{array}$$

Figure 2.2: An example of gauss elimination on the reduced laplacian obtaining the number of spanning trees rooted at r.

Lemma 2.14. *Any elementary row operation, adding a row i a constant c times to another row j, on L preserves on the number of spanning trees of the associated directed multigraph.*

Proof. In the multigraph, such an operation corresponds to **redirecting arcs** (v_i, v_j): We remove $c * L_{ii}$ arcs (v_j, v_i) and adjust other values to compensate. If we would have chosen one of the removed arcs in our spanning tree, all arcs (v_i, v_j) cannot be chosen anymore. Therefore, we compensate by lowering the degree of v_j with $c * (v_i, v_j)$. Moreover, using one of the removed arcs, there have been $c * (v_j, v_k)$ possibilities to assign v_k as (indirect) parent of v_i. Therefore $m(v_i, v_k)$ is raised with $c * (v_j, v_k)$. □

Proof of Theorem 2.13. As mentioned above, the theorem holds if L is upper triangle. Otherwise, we can use elementary row operations to make it upper triangle. Now the theorem follows from Lemma 2.14. □

2.5 Context

The first explicit occurrences of inclusion exclusion are dated in the beginning of the 17th century. In 1708, Pierre Rémond de Montmort (1678-1719) used the principle to solve *"le problème des recontres"*, which basically boils down to the counting of derangements from Section 2.1.3. In 1718 Abraham Demoivre published the first textbook about probability theory, *"Doctrine of chances"*, which contained a more set-theoretic approach of the technique. Sometimes, the principle is also attributed to James Joseph Sylvester (1814-1897).

The first inclusion exclusion formulation of a $\mathcal{NP}$-hard problem is probably due to Whitney [Whi32] and will be discussed in Section 3.4. The next publication which is worth mentioning is a 3-page note of Karp [Kar82], in which he applies inclusion exclusion to a small selection of well-known $\mathcal{NP}$-complete problems. Curiously, this note did not get many attention, and the technique was still a little bit unknown in the field of exact exponential algorithms. In 1996 [Bax96] published a way to break down inclusion exclusion, and gives some experimental results of applications. In 2006, inclusion exclusion received new attention: Börklund and Husfeldt, and independently Koivisto [BH06a, Koi06] gave an $\mathcal{O}^*(2^n)$ algorithm for graph coloring and some other problems. After this, they decide to collaborate and achieved more similar interesting results. Takacs provides more historical information about the principle and some classical applications [Tak67].

We mention that inclusion exclusion is also used in many other areas of research. Most of these areas are concerned with probability theory, such as network reliability and data mining. For network reliability applications we refer to [Doh03], and inclusion exclusion for data mining we refer to [CG05].

Inclusion exclusion is discussed in many (under)graduate textbooks about discrete mathematics: For example [Tuc94] and [Gri94] devote an entire chapter to inclusion exclusion, and [And04] gives a briefer overview.

Chapter 3

IE-Formulations

In this chapter we will give a number of applications of inclusion exclusion. With each application, we provide a brief discussion of its efficiency. In this way, we illustrate how one should use inclusion exclusion. Moreover, we will give some optimizations if they are not too complicated. For more involved optimizations we refer to Chapter 4.

We will extensively use the equation

$$|\bigcap_{r\in R} \overline{A_i}| = \sum_{X\subseteq R} (-1)^{|X|} |\bigcap_{i\in X} A_i|$$

of Corollary 2.6. Because of this, we call such an application an **(IE)-formulation** and shorthand it with the following notation:

#Prob

Instance	Input of #Prob
U	The universe U
$\bigcap_{r\in R} \overline{A_r}$	#Prob
A_r	The requirements A_r

In Section 3.1 we give a straightforward formulation for #Satisfiability, with a new optimization. After this, we will review the formulations of Karp for Hamiltonian cycle and TSP in Section 3.2. In Section 3.3 we discuss a formulation for a scheduling problem, which is again due to Karp, and give a new optimization. Section 3.4 will be about an old formulation of Whitney for Graph Coloring. In Section 3.5 we will review the formulation of Björklund et al. for the generic Set Cover. In the next section we will study an even more generic problem, Multi Covering and give a new efficient algorithm. In Section 3.7 we will show how the Cluster Editing problem can be solved with our framework, obtaining the fastest known algorithm.

3.1 CNF Satisfiability

For the definition of a CNF formula, we refer to Definition 1.5. #SATISFIABILITY is to count the number of satisfying assignments of a given CNF formula. The following IE-formulation is a straightforward one:

#SATISFIABILITY

Instance	A CNF formula
U	Variable assigments
$\bigcap_{C \in R} \overline{A_C}$	Satisfying variable assignments
A_C	Variable assignments such that C is not satisfied, for each clause C

Given a set of clauses $\mathcal{C}$, a variable is said to be **free** if it does not occur in any 2 clauses of $\mathcal{C}$. A variable v is said to be **conflicting** if both literals v and $\overline{v}$ occur in any clause of $\mathcal{C}$. $\mathcal{C}$ is called conflicting if it contains a conflicting variable. Now the simplified problem of the above formulation is as follows: given a set of clauses $\mathcal{C}$, compute $v(\mathcal{C})$, the number of variable assignments such that all clauses of $\mathcal{C}$ are not satisfied. This can be computed in polynomial time, using:

$$v(\mathcal{C}) = \begin{cases} 0 & \text{if } \mathcal{C} \text{ is conflicting} \\ 2^{F(\mathcal{C})} & \text{else} \end{cases}$$

where $F(\mathcal{C})$ is the number of free variables induced by $\mathcal{C}$.

The sumspace $\mathcal{S}$ could be reduced by only considering non-conflicting clause sets. Such sets can be enumerated, using an independent set[1] enumeration algorithm: a non-conflicting clause set corresponds to an independent set in the graph $G = (V, E)$, where V are all clauses of the CNF formula and vertices $u, v \in V$ share an edge if $\{u, v\}$ is conflicting. Although this is a very dense graph, $\mathcal{S}$ would still be exponential in the number of clauses, while a straightforward exhaustive branching algorithm would be exponential in the number of variables, which is more efficient for most instances.

3.2 Hamiltonian cycles and the travelling salesman problem

A walk in a graph $G = (V, E)$ is a sequence of vertices $W = (w_1, \dots, w_k)$ such that $(w_i, w_{i+1}) \in E$, for each $1 \leq i < k$. W is said to be a **cycle** if $(w_k, w_1) \in E$. The **length** of W is k. An HAMILTONIAN CYCLE is a cycle of length n that contains each vertex exactly once. We use the following IE-formulation for solving its counting variant, which is originally due to Karp [Kar82]:

[1]For the definition of an independent set, we refer to Section 1.3

#HAMILTONIAN CYCLES

Instance	A graph $G = (V, E)$
U	All cycles of length n, containing an arbitrarily chosen node s
$\bigcap_{v \in R} \overline{A_v}$	All cycles that contain each vertex exactly once
A_v	All cycles containing s of length n such that v is avoided

The simplified problem is to compute the number of cycles of length n that contain s, avoiding a certain node set X. This can be computed in $\mathcal{O}(|V|^2 * |E|)$ time with dynamic programming. We mention that this can be improved slightly: Let B be the adjacency matrix of the graph obtained by removing vertices X, and their adjacent edges. Then, if $B^n = [b_{vw}]_{v,w \in V}$, then $|\bigcap_{v \in X} A_i| = b_{ss}$. B^n can be computed with $lg(n)$ matrix multiplications, hence the time bound of $\mathcal{O}(lg(n) * |V|^{\omega})$.

In 1996, Bax presented a branch and reduce algorithm, corresponding with the above formulation, but does not improve the worst-case running time. For information on the reduction rules and some experimental results, we refer to [Bax96].

The previous formulation can easily be extended to solve TSP, defined in Subsection 1.3.2:

#TRAVELLING SALESMAN

Instance	A node set V a distance function $d : V \times V \rightarrow \mathbb{N}$ and a decision parameter k
U	All loops of length n of distance $\leq k$
$\bigcap_{v \in R} \overline{A_v}$	All Hamiltonian cycles of G of total distance $\leq k$
A_v	All cycles of length n such that v is avoided, for each $v \in V$

The simplified problem of this formulation can be computed in $\mathcal{O}(|V|^3 * C)$ with dynamic programming too, where C is maximum value of d. The actual recurrence is left as exercise for the reader. If we compare this formulation with the dynamic programming algorithm of Subsection 1.3.2, the most important difference is that the IE-formulation uses **polynomial space**, in contrary to dynamic programming. The only very small advantage of dynamic programming is that it does not have a factor C in its running time.

Actually, before paper of Karp [Kar82], Kohn et at. [KGK77] already proposed a similar algorithm for TSP, using generating functions, which is somewhat similar with inclusion exclusion. Very recently [BHKK08b] presented an $\mathcal{O}^*((2-\epsilon)^n)$ algorithm on graphs with bounded degree, for a small ϵ. This was achieved using techniques of Chapter 4.

3.3 A scheduling problem

Definition 3.1. *An **interval** $[a \ldots b]$ is the set of all integers between a and b, including a and b.*

Consider the following scheduling problem: Given are n jobs $J = \{J_1, \dots, J_n\}$, each with an interval $W_i = [r_i \dots d_i]$ and production time p_i. We assume $p_i > 0$. A **schedule** S is a set of tuples (J_i, s) such that $[s \dots s + p_i] \subseteq W_i$, and for each pair (J_i, s_1) and (J_j, s_2): $[s_1 \dots s_1 + p_i] \cap [s_2 \dots s_2 + p_j] = \emptyset$. Moreover, a schedule S is called **valid** if for each $J_i \in J$ there exists a s and $(J_i, s) \in S$. For counting the number of valid schedules, we use the following IE-formulation of Karp from [Kar82]:

#SCHEDULING WITH RELEASE TIMES AND DEADLINES

Instance	Jobs $J = \{J_1, \dots, J_n\}$, each with an interval $W_i = [r_i \dots d_i]$ and processing times p_i
U	All schedules
$\bigcap_{i \in R} \overline{A_i}$	All valid schedules
A_i	Schedules that do not contain J_i

The simplified problem can be computed in $\mathcal{O}(n * \max_i d_i)$ time, using dynamic programming. We omit the recurrence.

Now we will discuss an optimization of this algorithm for the decision variant which could be useful in practice. We use the technique of **adjusting the universe**, which will be discussed more explicitly in Section 4.2. The improvement is based on the following concept:

Definition 3.2. *Given $J = \{J_1, \dots, J_n\}$, with intervals $W_i = [r_i \dots d_i]$ and production times p_i, we construct the* ***scheduling graph*** *as follows: first, we add the jobs J_s with $r_s = d_s = -1$, $p_s = 0$ and J_t with $r_t = d_t = \max_i d_t + 1$, $p_s = 0$. Now we make a node for each pair (J_i, x) such that $x \in [r_i \dots d_i - p_i]$. Furthermore, we construct an arc between two nodes with pairs (J_i, x_1) and (J_j, x_2) when none of the following conditions hold:*

1. ***Overlap:*** $x_1 + p_i \geq x_2$
2. ***Impossible job:*** *There exists a job J_k such that $r_k > x_1$ and $d_k < x_2 + p_j$*
3. ***Same jobs:*** $i = j$
4. ***Earlier possible time:*** *There exists a node with pair (J_j, x_3) such that $x_1 + p_i < x_3$ and $x_2 < x_3$*
5. ***Job in between:*** *There exists a node with pair (J_k, x_3), $i \neq k$, such that $s_1 + p_i < x_3$ and $x_3 + p_k < x_2$*

We say a schedule S ***contains an arc*** *$((J_1, s_1), (J_2, s_2))$ if $(J_1, s_1), (J_2, s_2) \subseteq S$ and, there does not exist a $(J_3, s_3) \in S$ such that $s_1 \leq s_3 \leq s_2$.*

The following lemma illustrates the use of the scheduling graph:

Lemma 3.3. *For each valid schedule S that contains an arc which is not in the scheduling graph, there exists a valid schedule that does.*

Proof. Note that a set containing an arc satisfying condition 1 is not a schedule, while a schedule containing an arc satisfying condition 2 implies invalidness. Therefore, they can safely be ignored. Moreover, if S contains an arc (u, v) that satisfies condition 3 or 4, and $(v, w) \in S$ is the other arc that contains v, they can safely be replaced with the arc (u, w) preserving the validness. Similarly, if (u, v) satisfies condition 5, it can be replaced with $(u, (J_k, x_3))$ and $((J_k, x_3), v)$. □

Hence we can restrict ourselves to finding a path in the scheduling graph such that for each job J_i, the path contains at least one (J_i, s), for some s. While we still branch on these requirements, we could for example use the following reduction rule: if the deletion of all nodes labeled with one job J_i form a separator, make its requirement optional.

We note that the proposed branching strategy can safely be interleaved with other kind of branching strategies. Moreover more reduction rules can be formulated. the disadvantage is that making a requirement optional usually does not make our problem that much easier. However, potentially this could be an efficient formulation. We mention that this algorithm could also be used if we extend the problem with a cost function on arcs, and the sum of the costs of all arcs has to be minimized.

3.4 Graph coloring

The following IE-formulation was already published in 1931 by Whitney in [Whi32]:

#Graph coloring

Instance	A graph $G = (V, E)$, an integer k
U	All functions $\phi : V \to \{1, \ldots, k\}$
$\bigcap_{e \in R} \overline{A_e}$	For each edge $(y, z) \in E$, $\phi(y) \neq \phi(z)$
A_e	$\phi(y) = \phi(z)$, if the edge $v = (y, z)$

Given an edge set X, define $cc(X)$ as the number of connected components of the graph induced by the edge set X. Now, the simplified problem is as follows: Given any edge set $X \subseteq E$, compute $k^{cc(X)}$. Obviously, the formulation is not very efficient. Using IE-branching in combination with reduction rules, would be a little bit better: if we branch on meeting a requirement A_e, where $e = (x, y)$, the nodes x and y can be merged.

Let us mention the similarity with the more classical branching strategy, which is as follows: Take a pair which does not share an edge. Either this pair has the same number assigned (contract), or they do not (create an edge).

3.5 Set cover and partition problems

The results of this section are due to Björklund et al. [BHK06]. Let us consider the #k-Set Cover problem: Given a set R and a family of sets $\mathcal{F}$, compute the number of tuples $(F_1, \dots, F_k) \in \mathcal{F}^k$ such that $\bigcup_{i=1}^k F_i = R$. We use the following IE-Formulation:

#k-Set Cover

Instance	A set R, a family of sets $\mathcal{F}$
U	All k-tuples $(F_1, \dots, F_k) \in \mathcal{F}^k$
$\bigcap_{r \in R} \overline{A_r}$	All elements of U such that each element of R is in at least one set F_i
A_r	Element r is not contained in any set

Notice that the simplified problem is to compute $|\{F \in \mathcal{F} \mid F \cap X = \emptyset\}|^k$, for some given $X \subseteq R$. If this can be solved in polynomial time, this formulation gives an algorithm with a running time bounded by $\mathcal{O}^*(2^n)$, with polynomial space. If the simplified problem is $\#\mathcal{P}$-complete, achieving the same time bound is not obvious. However, if we make the assumption that given any subset $X \subseteq R$, membership of $\mathcal{F}$ can be decided in polynomial time, we can use dynamic programming to achieve the same time bound with exponential space. Let $R = \{r_1, \dots, r_n\}$. Define $g_j(X)$ as follows:

$$g_j(X) = \left| \left\{ F \in \mathcal{F} \;\middle|\; F \subseteq X, F \cap \{r_{j+1}, \dots, r_n\} = X \cap \{r_{j+1}, \dots, r_n\} \right\} \right|$$

or in words, $g_j(X)$ is the number of sets F which must contain all $r_i \in X$, for $i > j$, and possibly contain $r_i \in X$, for $i \leq j$. Using this notation, the simplified problem is to compute $g_n(R \setminus X)$, which can be done simultaneously for all X in $\mathcal{O}^*(2^n)$ using dynamic programming in combination with the following recurrence:

$$g_j(X) = \begin{cases} b(X) & \text{if } j = 0 \\ g_{j-1}(X \setminus \{r_j\}) + g_{j-1}(X) & \text{if } r_j \in X \\ g_{j-1}(X) & \text{if } r_j \notin X \end{cases} \tag{3.1}$$

Where $b(X)$ is the indicator function of $\mathcal{F}$, that is, $b(X) = 1$ if $X \in \mathcal{F}$ and $b(X) = 0$ if $X \notin \mathcal{F}$.

This formulation can easily be extended for additional constraints on solutions: The #k-Set Partition problem asks to compute the number of words of k-Set Cover which contain each element of R exactly once. This can be solved by adjusting the universe, using the following observation: A k-Set Partition is a k-Set Cover if and only if the sum of the sizes of the sets are exactly n. So we can use the following formulation:

#k-Set Partition

Instance	A set R, a family of sets $\mathcal{F}$
U	All k-tuples $(F_1, \dots, F_k) \in \mathcal{F}^k$ such that $\sum_{i=1}^k |F_i| = n$
$\bigcap_{r \in R} \overline{A_r}$	All elements of U such that each element of R is in at least one set F_i
A_r	Element r is not contained in any set

The simplified problem can be solved as follows: Let $g_n(s, X)$ be the number of elements $F \in \mathcal{F}$ such that $F \subseteq X$ and $|F| = s$. We compute $g_n(s, X)$, for each $X \subseteq R$, using Equation 3.1 extended with the parameter s, and with the addition of the base case $g_0(s, X) = 0$, if $|X| \neq s$. Now denote $t(k, s, X)$ for the number of k-tuples $(F_1, \dots, F_k) \in \mathcal{F}^k$ such that $\sum_{i=1}^{k} |F_i| = s$ and $F_i \cap X = \emptyset$, for each $1 \leq i \leq k$. The simplified problem is to compute $t(k, n, V \setminus X)$ for each $X \subseteq R$ and can be computed using the following recurrence:

$$t(k, s, X) = \begin{cases} g_n(s, X) & \text{if } k = 1 \\ \sum_{i=0}^{s} g_n(i, X) * t(k-1, s-i, X) & \text{else} \end{cases}$$

Further pushing the envelope, suppose a cost function $c : 2^R \to \mathbb{N}$ is given, and we want to find a partition $(F_1, \dots, F_k) \in \mathcal{F}^k$ such that $\sum_{i=1}^{k} F_i$ is minimized. By further adjusting the universe, this problem can be solved too. We omit the details.

We conclude this section by mentioning that in [BHKK07], Björklund et al. gave an $\mathcal{O}^*(2^n)$ algorithm for the following even more generic problem: Given 2 polynomial time computable function $f, g : 2^R \to \mathbb{N}$, compute their **subset convolution** $(f * g)$ for all $R' \subseteq R$, where $(f * g)$ is defined as follows:

$$(f * g)(R) = \sum_{X \subseteq R} f(X) * g(R \setminus X)$$

For more similar algorithms and their applications we refer to [BHKK07].

3.6 Multi set cover and set partition problems

To state the problem that we will solve in this section, we need to introduce the following:

Definition 3.4. *A **multiplicity function** of R is a function $m : R \to \mathbb{N}$. We use the following notation:*

- *Given 2 multiplicity functions x and m of R, we write $x \preceq m$ if for all $r \in R$, $x(r) \leq m(r)$*
- *The **length** $|x|$, is defined as $\sum_{v \in V} x(v)$.*
- *The **elements** $R[x]$ of x, are all $r \in R$ for which $x(r) > 0$.*
- *$m - x$ is the component wise subtraction (as in vector subtraction).*

Now, the problem #k-Multi Set Cover is defined as follows: Given a set R with multiplicity function m and a family of sets $\mathcal{F}$, #k-Multi Set Cover is to compute the number of tuples $(F_1, \dots, F_k) \in \mathcal{F}^k$ such that for each $r \in R$, r is in contained in at least $m(v)$ sets.

This problem could be solved using a SET COVER formulation: Our requirement space R' consists of $m(r)$ copies of each $r \in R$, and hence $\mathcal{F}'$ consists of $\prod_{r \in F} m(r)$ copies, for each $F \in \mathcal{F}$. Unfortunately this gives an algorithm using $\mathcal{O}^*(2^{|m|})$ time, which is not very efficient. Somewhat inspired by the efficient formula for counting derangements of Subsection 2.1.3, we will give a more efficiently computable formula by exploiting symmetries. Denote $s(x)$ for the simplified of the Set Cover formulation above, corresponding to a subset of R' consisting of $x(r)$ copies of r, for each $r \in R$. Note that this is equal to $s(x) = \sum_{F \in \mathcal{F}} \prod_{r \in F} m(x)$. Now we can obtain the following:

Theorem 3.5.

$$\#k\text{-MULTI SET COVER} = \sum_{x \preceq m} (-1)^{|x|} \left(\prod_{r \in R[x]} \binom{m(v)}{x(v)} \right) s(m-x)^k$$

Proof. For each multiplicity function $x \preceq m$, there are $\prod_{r \in R(m)} \binom{m(v)}{x(v)}$ subsets of R' where each vertex r has $m(r) - x(r)$ copies. All subsets of R' have an associated multiplicity x for which $x \preceq m$. Moreover, there are exactly $\prod_{r \in F} m(x)$ copies of F in $\mathcal{F}'$. Therefore, the contribution of a subset with multiplicity function x to the #SET COVER formulation is exactly $s(m-x)$. Hence the equation is equivalent to the covering formulation with R' and $\mathcal{F}'$ □

As an example, this theorem can be used for solving the multicoloring problem, which is a special case of #k-MULTI SET COVER where $\mathcal{F}$ are all independent sets[1] of a given graph $G = (V, E)$, and the node set has to be covered, hence $R = V$. The simplified problem, computing $s(x)$, can be computed simultaneously for all $x \preceq m$ within the same time bound, using dynamic programming with the following recurrence:

$$s(x) = \begin{cases} 1 & \text{if } x(v) = 0 \text{ for each } v \in V \\ s(x_{\{v\} \to 0}) + x(v) * s(x_{N[v] \to 0}) & \text{otherwise} \end{cases}$$

where $x_{B \to 0}$ is the multiplicity function obtained by setting $x(v) = 0$ for each $v \in B$.

We mention that, similarly to the concept of an IE-branch, one could also use **multi IE-branching**: suppose an element has to be covered c times, then imagine there are c-copies. We can branch in the following way, where $Forbidden^i$ means that i copies are forbidden.

$$Required^c = \sum_{i=0}^{c} (-1)^i \binom{c}{i} Forbidden^i$$

[1] For the definition of an independent set, see Section 1.3

We conclude, by mentioning that in fact, one could formulate equations which strongly generalize the classical inclusion exclusion. We refer to Narushima [Nar82] for inclusion exclusion on partially ordered sets.

3.7 Cluster editing

If $G = (V, E)$ is a graph, a **clique** is a subset $X \subseteq V$ such that for each $u, v \in X$, $(u, v) \in E$. An **edge edit** is removing an edge, or adding one. The problem c-Cluster Editing is defined as follows: can we obtain a graph where each connected component is a clique, using c edge edits?

Definition 3.6. *Given a subset $X \subseteq V$, an edge is said to be **intern** if both adjacent vertices are in X. An edge is said to be **outward** if exactly one adjacent node is in X. Denote the number of intern and outward edges as respectively $i(x)$ and $o(X)$. The **edit costs** of X, $e(X)$, is defined as follows:*

$$e(X) = \frac{o(X)}{2} + \Big(\frac{|X| * (|X| - 1)}{2} - i(X)\Big)$$

Now, each c-Cluster Editing corresponds to a partition $(S_1, \ldots, S_k)$, for some k, such that $\sum_{i=1}^{k} e(S_i) = c$: We construct a clique with nodes S_i for each i. The costs of adding a missing intern edge is counted in the costs of the containing set S_i, while removing a redundant outward edge is counted half in both endpoints.

Solving the simplified problem boils down to computing $\#(c, s)$-**Clique Editing**, which is the number of $X \subseteq V$ of size s, such that, $e(X) = c$. We obtain the following negative result:

Lemma 3.7. *$\#(c, s)$-Clique editing is $\#\mathcal{P}$-complete.*

Proof. A subset $X \subseteq V$ is a **clique** if it induces a complete subgraph. We use a reduction from the problem $\#d$-Regular Clique: Given a graph $G = (V, E)$ where each node has the same degree d, compute the number of cliques $X \subseteq V$. This problem was proven to be $\#\mathcal{P}$-complete by Vadhan in [Vad01]. Note that in a regular graph for each clique X, $o(X) = |X| * (d - (|X| - 1))$, and there are no missing intern edges. Moreover, if X is not a clique, $e(X) > |X| * (d - (|X| - 1))$. Hence for the number of cliques we have:

$$\#d\text{-Regular Clique} = \sum_{i=0}^{n} \#\Big(\frac{i * (d - (i - 1))}{2}, i\Big)\text{-Clique editing} \qquad \square$$

Note that, due to Section 3.5, we can already obtain an $\mathcal{O}^*(2^n)$ time algorithm with exponential space, using dynamic programming: we can decide in polynomial time whether a certain subset is indeed a clique editing of size s and edit costs c, for any s and c. However, what should we do if only polynomial space is available? A naive approach would be exhaustive branching for each instance of $\#(c, s)$-Clique

EDITING. However, we present a slightly faster algorithm, using the technique of Williams from Subsection 1.3.4:

Lemma 3.8. $\#(c, s)$-CLIQUE EDITING *can be computed in* $\mathcal{O}^*(1.7314^n)$ *time.*

Proof. Arbitrarily partition V in V_0, V_1 and V_2. Define $e_{ij}(X)$ as $e(X)$, ignoring edges adjacent to vertices not in $V_i \cup V_j$. For each $c_{01} + c_{12} + c_{20} = c$ and $s_0 + s_1 + s_2 = s$, create the following graph: Construct nodes for each subset of some V_i, for each i. Construct an edge between nodes $X_i \subseteq V_i$ and $X_j \subseteq V_j$, if $i \neq j$, and $e_{ij}(X_i \cup X_j) = c_{ij}$. Now each triangle corresponds to a (c, s)-Clique editing, and vice versa. Concluding, we sum over all possible values for c_{ij} and s_i (note that this is a polynomial amount), and compute the number of triangles in the corresponding graph using matrix multiplication. □

Because we have to use the above algorithm for each subset of V, we obtain an improvement of the running time from $\mathcal{O}^*(3^n)$ to $\sum_{i=1}^{n} \binom{n}{i} \mathcal{O}^*(1.7314^i) = \mathcal{O}^*(2.7314^n)$, for the polynomial space variant.

Chapter 4

Additional techniques

After finding an IE-formulation, one may wonder how it can be used for an efficient algorithm. Is it necessary to compute all terms of the summation, or can we find more efficiently computable formulas? In this chapter we will discuss techniques that provide these smaller algorithms. The first three sections will be about **reducing the sumspace**: is it necessary to sum over the entire powerset of the requirement space, or can some subsets be ignored? In Section 4.4, we give an example of how to handle hard simplified problems, once we have obtained a reduced sumspace. The last section is about an interesting way algebraically rewriting an inclusion exclusion formula.

4.1 Abstract tubes

Denote $R[u] = \{r \in R \mid u \in A_r\}$, for any requirement space R and element $u \in U$. Using this notation, recall from the proof of Theorem 2.5 we defined $\mathcal{S}[u]$ as $\{S \in \mathcal{S} \mid S \subseteq R[u]\}$ for any sumspace $\mathcal{S}$. Assuming $\mathcal{S}$ is the simple sumspace induced by R, we obtained the equality:

$$|\bigcup_{i=1}^{n} A_i| = \sum_{\substack{X \subseteq R \\ X \neq \emptyset}} (-1)^{|X|-1} |\bigcap_{i \in X} A_i| = \sum_{u \in U} \chi(\mathcal{S}[u]) \tag{4.1}$$

where χ denotes the Euler characteristic from Definition 2.3. Now, the following question arises: is there any smaller sumspace $\mathcal{S}$, such that Equation 4.1 still holds? In some cases this is obviously true. For example, zero-valued terms of $\mathcal{S}$ can always be left out.

Abstract tubes were introduced by Naimann and Wynn in [NW92]. In [NW97] a generalization was obtained along with examples such as computing the volume of a convex polyhedron. In this brief overview, we will discuss a simplified version of

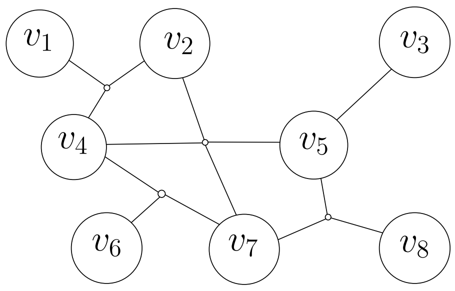


Figure 4.1: An example of a contractible sumspace visualized with a hypergraph where only the maximal hyperedges/sets are shown.

the abstract tubes of [NW97]. For more information and an application to network reliability we also refer to the book of Dohmen [Doh03].

Definition 4.1. *A **contraction** of a simple sumspace $\mathcal{S}$ on R is removing a set $S \in \mathcal{S}$ and simultaneously removing a set $S' \in \mathcal{S}$, such that $S = S' \cup \{r\}$, for some $r \in R$. A simple sumspace is said to be **contractible** if we can obtain the sumspace $\{\{r\}\}$, for some $r \in R$, only using contractions. An **abstract tube** is a pair $(\mathcal{A}, \mathcal{S})$ consisting of requirements $\mathcal{A} = \{A_r\}_{r \in R}$ and a simple sumspace $\mathcal{S}$ on R such that for each $u \in \bigcup_{r \in R} A_r$, the sumspace $\mathcal{S}[u]$ is contractible.*

Sometimes we will interchange the notion of a simple sumspace with a **hypergraph**, where the edge set of the hypergraph are the maximal sets of the simple sumspace, and vice versa. In this manner, contractibility is defined on hypergraphs too. In Figure 4.1 we give an example of such a contractible hypergraph. The reason of the definition of abstract tubes is the following theorem:

Theorem 4.2. *Suppose $(\{A_r\}_{r \in R}, \mathcal{S})$ is an abstract tube, then:*

$$\bigcup_{i=1}^{n} A_i = \sum_{X \in \mathcal{S}} (-1)^{|X|-1} |\bigcap_{i \in X} A_i|$$

Proof. As before, rewrite the right-hand side to $\sum_{u \in U} \chi(\mathcal{S}[u])$. Obviously, if $u \notin \bigcup_{i=1}^{n} A_i$, we have $\chi(\mathcal{S}[u]) = 0$. Hence, the only thing left to prove is that $\chi(\mathcal{S}[u]) = 1$ for each $u \in \bigcup_{i=1}^{n} A_i$. To this end, note that a contraction preserves the Euler characteristic $\chi(\mathcal{S}[u])$. Now the equality follows from the fact that $\chi(\{\{v\}\}) = 1$, for any v. $\square$

In fact, we could even use a weaker notion of contractibility. However, in this brief overview this notion will suffice.

4.1.1 Tree-shaped requirements

The following example is due to [NW97].

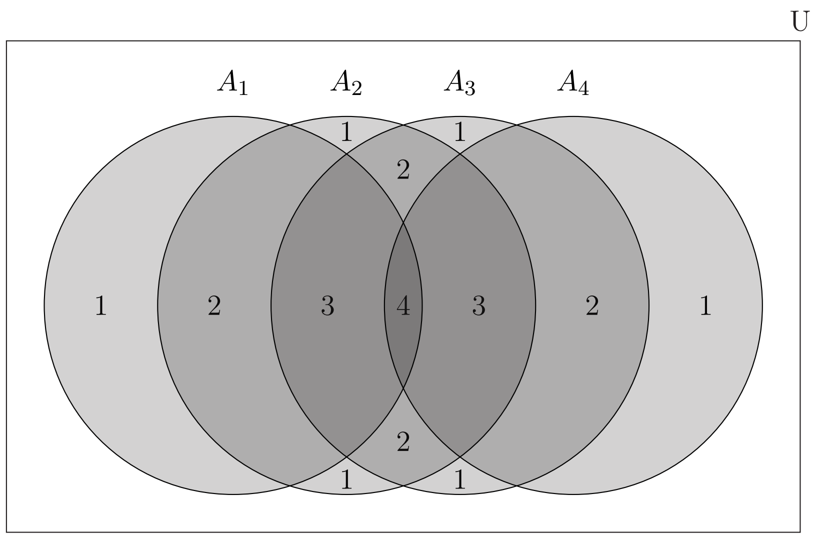


Figure 4.2: The origin of the name abstract tubes is due to this example. After computing $\sum_{i=1}^{4} |A_i|$, each part is counted as indicated in the diagram. Subtracting $\sum_{i=1}^{3} |A_i \cap A_{i+1}|$ gives us $|\bigcup_{i=1}^{4} A_i|$.

Lemma 4.3. *Suppose we are given a set of requirements $\mathcal{A} = (\{A_r\}_{r\in R})$, and a tree $T = (R, E)$ such that for each $x, y, z \in R$, where y is on the unique path between x and z in T, the following holds:*

$$A_x \cap A_z \subseteq A_y$$

Then, $(\mathcal{A}, T)$ is an abstract tube.

Proof. For each $u \in \bigcup_{i\in R} A_i$, $T[u]$ is a tree. Note that a tree is a single vertex, or we can obtain a smaller tree using a contraction. Hence $T[u]$ is contractible. $\square$

The name 'abstract tube' is due to the visual example of Figure 4.2. Let $\mathcal{S}$ be the simple sumspace induces by the edge set $\{(A_1, A_2), (A_2, A_3), (A_3, A_4)\}$. Note that $\mathcal{S}[u]$ is a path, for each $u \in \bigcup_{i=1}^{4} A_i$, hence contractible. Because of this, $(\{A_{1,4}\}, \mathcal{S})$ is an abstract tubes, implying that $\bigcup_{i=1}^{4} A_i = \sum_{i=1}^{4} |A_i| - \sum_{i=3}^{4} |A_i \cap A_{i+1}|$.

4.1.2 Whitney's broken circuits

As mentioned in [Doh03], we can prove an old theorem of Whitney from [Whi32], using the theory of abstract tubes. Actually, we will slightly improve the efficiency of the IE-formulation of Section 3.4 for computing #Graph Coloring. Suppose we are given a graph $G = (V, E)$, then recall from the IE-formulation: our universe $U = \{\phi : V \to \{1, \dots, k\}\}$, and the requirements $\mathcal{A} = \{A\}_{(v,w)\in E}$ with $A_{(v,w)} = \{\phi \in U \mid \phi(v) = \phi(w)\ \}$.

Definition 4.4 ([Doh03]). *Assume the edge set E is endowed with a linear ordering relation. A **broken circuit** of G is obtained from the edge set of a cycle of G by removing its maximum edge. The **broken circuit complex** of G, abbreviated to $\mathcal{BC}(G)$, is the simple sumspace consisting of all non-empty subsets of E that do not include any broken circuit of G as subset.*

Although the following theorem is due to [Doh03], we give simpler proof.

Theorem 4.5. *For any $k \in \mathbb{N}$, the following is an abstract tube:*

$$(\mathcal{A}, \mathcal{BC}(G))$$

Proof. We have to prove for any $u \in \bigcup_{e \in E} A_e$, that $\mathcal{BC}[u]$ is contractible. We use induction on the number of cycles of $R[u]$. In the base case, $R[u]$ has no cycles. Note that, due to the definition of $\mathcal{A}$, $R[u]$ also has no broken circuits. Because of this, $\mathcal{BC}[u]$ is equal to the sumspace induced by $R[u]$, which is trivially contractible.

Now suppose $R[u]$ consists of at least one cycle. Define m as the maximum edge of such a cycle, and define m' as some other edge on the same cycle. Note that for each $S \in \mathcal{BC}[u]$ such that $m' \in \mathcal{S}$ and $m \notin \mathcal{S}$, $S \cup \{m\}$ is also an element of $\mathcal{BC}[u]$. Hence, all sets containg m can be removed with contractions, consisting of simulateously removing S and $S \cup \{m\}$. Note that we can not create any broken circuit. Using the induction hypothesis, we may conclude $\mathcal{BC}[u]$ is also contractible in this case. $\square$

4.2 Adjusting the universe

Suppose we want to solve a decision problem. Using inclusion exclusion, we have to solve a counting problem to which the decision problem can efficiently be reduced. Obviously, there can be more than one of such a counting problem. Therefore, it is important to choose one that can be solved efficiently. The technique that will be discussed now was already used Section 3.3, although not mentioned explicitly.

We give an example for the decision variant of Graph Coloring. This study will be continued in the following two sections. First we need the following definition:

Definition 4.6. *Let $G = (V, E)$ be graph. A node u **dominates** v if $(u, v) \in E$ or $u = v$. Similarly, a node set X dominates X' if for each $v \in X'$, there exists a $u \in X$ such that u dominates v. A **maximal independent set** of G is an independent set of G that dominates V.*

k-Graph coloring can be reduced to a problem named k-IS-cover, which is defined as follows: given a graph $G = (V, E)$, can V be covered with k independent sets of G? Using the set cover formulation of Section 3.5, this problem can be solved in $\mathcal{O}^*(2^n)$. In this formulation the universe consists of all k-tuples of independent sets of G. Now observe the following: if a k-tuple covers V and contains an independent set which is not maximal, this independent set can always safely be extended to a maximal one. Because of this we are allowed to adjust the universe to tuples of maximal independent sets of G. With this new universe, we can predict zero-valued terms of the sumspace, using the following: if X does not dominate V, it can not contain a maximal independent set of G. Hence, we can reduce the sumspace to all dominating sets of G.

This technique was used by Björklund et al. in [BHKK08a] to obtain an algorithm with running time $\mathcal{O}^*((2-\epsilon)^n)$, for a small ϵ, for a graph coloring and some other problems on graphs of bounded degree. For graph coloring, they further adjusted the universe to all $\frac{1}{2}k$ tuples of maximal bipartite (i.e. 2-colorable) subgraphs. Note that, with this universe, an element of the sumspace can only be non-zero if it contains at least 2 neighbors of each node.

4.3 Inclusion-exclusion branch & reduce

In contrast with the two techniques of the previous sections, the method we discuss in this section reduces the sumspace in a more dynamic manner.

Similar to branch & reduce, this technique uses reduction rules on top of exhaustive IE-branching. Actually, we already saw some applications of this technique in Chapter 3. Usually, reduction rules are used with the purpose of proving a better upper bound of the running time of the algorithm. Unfortunately, for our applications such an upper bound can probably only be achieved using advanced techniques such as measure & conquer, which will be discussed in Chapter 5. In this case, such an analysis is beyond the scope of the thesis. Moreover, it is clear that some algorithms still use $\mathcal{O}^*(2^n)$ time on worst-case inputs.

Designing an IE-branch & reduce algorithm, is almost the same as designing a normal branch & reduce algorithm. Moreover, IE-branching also can easily be combined with other branching strategies. For an example we refer to Chapter 5.

For the other part of the algorithm, the reduction rules, the effect of IE-branching seems less clear. Instead of the old problem, we have to solve the simplified problem which is some counting variant. As mentioned in Section 1.3, some reduction rules for decision problems can not be used for their counting variants.

For illustration, we proceed by studying the IE-formulation of the previous section. Recall we use a maximal independent set cover formulation. Hence, using the notation #k-$\mathrm{MIS\text{-}C}$(V) for the number of ways to cover V with k maximal independent sets of k, we obtain:

$$\#k\text{-}\mathrm{MIS\text{-}C}(V) = \sum_{X \subseteq V} (-1)^{|X|} \#\mathrm{MIS}(V \setminus X, X)^k$$

where $\mathrm{MIS}(O,S)$ is the number of maximal independent sets that are a subset of O and dominate all nodes $O \cup S$, defined for $O \cap S = \emptyset$. Because the summation only depends on the simplified problem, the following question arises: can reduction rules for the simplified problem also be used for reducing the sumspace?

In Algorithm 1, we give an algorithm for the $\#\mathrm{MIS}(O, S)$ problem, only consisting of reduction rules which are due to Gaspers et al. [GKL08]. We use the following notation:

Definition 4.7. *Given a graph $G = (V, E)$, we denote $N(v)$ for the set of all neighbors of v and $N[v] = N(v) \cup \{v\}$. Similarly we denote $N_X(v) = N(v) \cap X$ and $N_X[v] = (N(v) \cup \{v\}) \cap X$. Moreover, $d(v) = |N(v)|$ and $d_X(V) = |N_X(v)|$.*

Algorithm 1 $\#\mathrm{MIS}(O, S)$

```
1: if there exists $u \in S$ s.t. $d_O(u) = 0$ then
2:     return 0
3: else if $\exists u \in S$ s.t. $N_O(u) = \{v\}$ then
4:     return $\#MIS(O \setminus N[v], S \setminus N(v))$
5: else if $\exists u \in S$ and $v \in N_O(u)$ s.t. $N_O[u] \subseteq N[v]$ then
6:     return $\#MIS(O, S \setminus \{u\})$
7: else if $\exists u, v \in S$ s.t. $N_O(u) = N_O(v)$ then
8:     return $\#MIS(O, S \setminus \{v\})$
9: else if $\exists u \in O \cup S$ and $v \in O$ s.t. $u \neq v$ and $N_O(u) = N_O(v)$ then
10:    return $\#MIS(O \setminus \{v\}, S)$
11: end if
```

We give a brief explanation of the reduction rules of Algorithm 1:

Line 1 Halt: u can not be dominated.

Line 3 Unique element: u can only be dominated by choosing its neighbor.

Line 5 Subsumption: If v is dominated, u will be dominated too.

Line 7 Identical elements: If v is dominated, u will be dominated too.

Line 9 Mirroring: If v is chosen, u will be chosen too.

Notice that for the corresponding decision variant, maximum independent set, many powerful reduction rules are applicable, for examples we refer to [FGK06].

Now, we use the reduction rules for an IE-Branch algorithm to solve $\#k\text{-MIS-C}(V)$, displayed in Algorithm 2. We compute $\#k\text{-MIS-C}(R, O, S)$, which is the number of k-tuples of maximal independent sets $(M_1, \ldots, M_k)$ such that for each i : $M_i \subseteq R \cup O$ and $R \subseteq \bigcup_{i=1}^{k} M_i$. Similar to the IE-branching terminology, we say R are the **required**, S are the **forbidden** and O are the **optional** nodes. We also use $A = R \cup O$ for all **active** nodes.

On Line 1 we have the Halt reduction rule, which does not change. For the Unique element rule on Line 3, we have that u can only be dominated by choosing its neighbor, hence each M_i has to contain v. The Subsumption rule on Line 9 and the

Algorithm 2 $\#k$-MIS-C(R, O, S)

```
Input: Integer $k$, required nodes $R$, optional nodes $O$ and forbidden nodes $S$
Output: The number of $k$-tuples of maximal independent sets $(M_1, \ldots, M_k)$ s.t.
    for each $i$ : $M_i \subseteq R \cup O$ and $R \subseteq \bigcup_{i=1}^{k} M_i$
 1: if there exists $u \in S$ s.t. $d_A(u) = 0$ then
 2:   return 0
 3: else if $R = \emptyset$ then
 4:   return $\#\text{MIS}(O, S)^k$
 5: else if $\exists u \in S$ s.t. $N_A(u) = \{v\}$ then
 6:   if $d_R(v) = 0$ then
 7:     return $\#k$-MIS-C$(R \setminus \{v\}, O \setminus N[v], S \setminus N(v))$
 8:   else
 9:     return 0
10:   end if
11: else if $\exists u \in S$ and $v \in N_A(u)$ s.t. $N_A[u] \subseteq N[v]$ then
12:   return $\#k$-MIS-C$(R, O, S \setminus \{u\})$
13: else if $\exists u, v \in S$ s.t. $u \neq v$ and $N_A(u) = N_A(v)$ then
14:   return $\#k$-MIS-C$(R, O, S \setminus \{v\})$
15: else if $\exists u \in S \cup A$ and $v \in A$ s.t. $u \neq v$ and $N(u) = N(v)$ then
16:   if $u \in S$ and $v \in R$ then
17:     return 0
18:   else if $u \in O$ and $v \in R$ then
19:     return $\#k$-MIS-C$(R, O \setminus \{u\}, S)$
20:   else
21:     return $\#k$-MIS-C$(R, O \setminus \{v\}, S)$
22:   end if
23: else
24:   Take some $u \in R$
25:   $n_{optional} = \#k$-MIS-C$(R \setminus \{u\}, O \cup \{u\}, S)$
26:   $n_{forbidden} = \#k$-MIS-C$(R \setminus \{u\}, O, S \cup \{u\})$
27:   return $n_{optional} - n_{forbidden}$
28: end if
```

Identical elements rule on Line 11 also do not change. The mirroring rule becomes a bit more involved: if u is forbidden (but has to be dominated) and v is required, we return 0 because v can not be in any. If u is optional and v is required, we can remove u without changing the number of indepedent sets. In all other cases, v is required and u is optional, both are required or both are optional, v can be removed.

The reduction rules of above are just given as example and these are not the only ones that are applicable. Furthermore, it might be interesting to look at which reduction rules can be used with $\frac{1}{2}k$-tuples of maximal bipartite subgraphs. This is left as further research.

4.4 Dynamic programming for the simplified problem

As discussed in Section 3.5, we can use dynamic programming to compute all simplified problems simultaneously in the context of set cover formulation. If we can decide in polynomial time whether $X \in \mathcal{F}$, for a given $X \subseteq R$, the inclusion exclusion formula can still be evaluated in $\mathcal{O}^*(2^n)$ time. Now the following question suggests itself: can this technique be combined with the techniques of the previous section? The answer is positive:

Lemma 4.8. *Given are a branching tree and a set of simplified problems* $\#\mathrm{MIS}(O, S)$*, for some* $O \subseteq V$*,* $S \subseteq V$ *and* $O \cap V = \emptyset$*, obtained by an execution of Algorithm 2. Now all simplified problems can be computed simultaneously in time bounded by the branching tree.*

Proof. We will only give a sketch of a proof. Extend Algorithm 1 with the following branching rule:

$$\#\mathrm{MIS}(O, S) = \#\mathrm{MIS}(O \setminus N[v], S \setminus N(v)) + \#\mathrm{MIS}(O \setminus \{v\}, S \cup \{v\})$$

i.e. count maximal independent sets that contain v (the take branch), and maximal independent sets do not contain v (the discard branch), and return the sum of both. In the take branch, all neighbors can't be chosen anymore. Use this extended algorithm in combination with memorization to compute all simplified problems.

Now we have, that the number of terms $\#\mathrm{MIS}(O, S)$ that will be evaluated is bounded by the number of terms $\#k\text{-}\mathrm{MIS\text{-}C}(R, O, S)$ that are evaluated because the take branch is equivalent to making $N[v]$ forbidden, while the discard branch is equivalent to making v forbidden. Because these will both occur too in the branch tree, the lemma follows. □

We mention that the algorithm of Björklund et al [BHKK08a] follows if we only use the reduction rule on Line 3 from Algorithm 1.

4.5 Rewriting

Recall from Section that given a graph $G = (V, E)$, a Perfect Matching is a subset $M \subseteq E$ such that each node of V is in exactly one edge of M. Notice that this is a special case of k-Set Partition: Each edge is a set containing its to adjacent nodes, and we have to partition the entire node set, hence using exactly $n/2$ edges. Using the considerations of Section 3.5 about k-Set Partition, we obtain:

$$\#\text{Perfect Matching} = \sum_{X \subseteq V} (-1)^{|V \setminus X|} e(V \setminus X)^{n/2} \tag{4.2}$$

where $e(V \setminus X)$ is the number of edges in the subgraph induced by node set $V \setminus X$. Björklund et al. published an interesting way of improving this algorithm in [BH06a]:

Lemma 4.9. *Define $a_0(i)$ ($a_1(i)$) as the number of even (odd) subsets of V containing exactly i edges. Now the following holds:*

$$\#\text{Perfect Matching} = \sum_{i=0}^{|E|} (a_0(i) - a_1(i)) i^{n/2}$$

Proof. We rewrite the right-hand side of Equation 4.2, by grouping all even and odd subsets which have the same number of edges in it. □

This lemma is already interesting on its own. For example it implies that computing $a_b(i)$ is $\#\mathcal{P}$-complete. However, it appears that $a_b(i)$ can be computed faster than the $\mathcal{O}^*(2^n)$ time bound, using the technique of Williams (see Subsection 1.3.4). We sketch the algorithm and refer to [BH06a] for more details: To compute $a_b(i)$, construct the graph H with nodes X_i for each subset of each $X_i \subseteq V_i$. Create edges such that each subgraph induced by $X \subset V$ containing exactly i edges corresponds with a triangle (X_0, X_1, X_2) in H, such that $X = X_0 \cup X_1 \cup X_2$. Now use matrix multiplication to compute the number of triangles. This results in an algorithm with a $\mathcal{O}^*(1.7314^n)$ time bound.

Lemma 4.10. *Given a graph G with an independent set I, #Perfect Matching can be computed in $\mathcal{O}^*(2^{n-|I|})$*

Proof. Adjust the universe of the above formulation, each node of the independent set chooses an adjacent edge. The other edges are chosen arbitrarily. Now, the requirement space can be reduced to $V \setminus I$. □

#Perfect Matching can also be computed faster than $\Omega * (2^n)$ using abstract tubes. We refer to [Doh99].

The rewriting step of Lemma 4.9 can obviously be used on any (improved) IE-formula. For example, the IE-formulation of Section 3.4 implies:

$$\#\text{Graph Coloring} = \sum_{i=0}^{|V|} (a_0(i) - a_1(i)) k^i$$

where $a_0(i)$ $(a_1(i))$ is the number of even (odd) subsets of E induced exactly i connected components. Note that computing $a_b(i)$ is necessarily $\#\mathcal{P}$-complete, due to the $\#\mathcal{P}$-completeness of $\#$COLORINGS.

Chapter 5

A case study: dominating sets

This chapter is joint work with Thomas C. van Dijk and Johan M.M. van Rooij, and will also appear as technical report. We use IE-branching on a well-studied problem and improve the fastest known algorithm. We combine a branching algorithm with dynamic programming on a path decomposition, following Fomin et al. [FS07], and extend it with IE-branching.

In this brief discussion we will simplify things a bit. This chapter is still work in progress. For a more detailed and complete discussion, we refer to the technical report that will appear shortly on http://www.cs.uu.nl/research/techreps/.

5.1 Dominating set

Recall the following terminology of Section 4.3:

Definition 5.1. *Given a graph $G = (V, E)$, we denote $N(v)$ for the set of all neighbors of v and $N[v] = N(v) \cup \{v\}$. Moreover, $d(v) = |N(v)|$.*

Given a graph $G = (V, E)$, $\#k$-DOMINATING SET is to compute the number of subsets $X \subseteq V$ such that $|X| = k$ and for each $v \in V$, $N[v] \cap X \neq \emptyset$.

A transformation from $\#k$-DOMINATING SET to $\#k$-SET COVER is straightforward: for every $v \in V$ in the dominating set instance, make an element in R and a set in $\mathcal{F}$ containing the elements corresponding to the vertices in $N[v]$. Its decision variant DOMINATING SET, is a well studied problem: in 2004, an $\mathcal{O}(1.93782^n)$ time was published by Fomin et al. [FKW04] and independently Randerath and Schiermeyer gave an $\mathcal{O}(1.8899^n)$ time algorithm [RS04]. A publication of Grandoni improved the upper bound to $\mathcal{O}(1.81^n)$ [Gra06]. However, in 2005, Fomin et al. [FGK04] used the technique of measure & conquer to obtain an $\mathcal{O}(1.5137^n)$ algorithm. This was later improved by van Rooij in [vR06, vRB08] to $\mathcal{O}(1.5086^n)$ and $\mathcal{O}(1.5063^n)$ respectively.

In this case study, we will give an extended version of the algorithm from Fomin et al. [FS07], which computes $\#k$-Dominating set. We improve their worst-case running time of $\mathcal{O}(1.5535^n)$ to $o(1.5^n)$, hence also obtaining the fastest algorithm for the decision variant. Our improvement is due to the use of IE-branching, and the addition of memorization.

5.2 Algorithm

Algorithm 3 k-SC($\mathcal{F}$,R)

```
Input: A set $R$, a family $\mathcal{F}$ of subsets of $R$, an integer $k$
Output: The number of subsets $\mathcal{S} \subseteq \mathcal{F}$ s.t. $|\mathcal{S}| = k$ and $\bigcup_{S \in \mathcal{S}} S = R$
 1: if $\exists u \in R$ : freq$(u) = 1$ then //unique elements
 2:     Let $S \ni u$
 3:     return $\sum_{i=1}^{k} k$-SC({ $S' \setminus S \mid S' \in \mathcal{F}$ } , $R \setminus S$ )
 4: else if $\exists \emptyset \subset \mathcal{C} \subset \mathcal{F}$ : $\{\mathcal{F}[u] \mid u \in S \in \mathcal{C}\} = \mathcal{C}$ then //connected components
 5:     Let $R[\mathcal{C}] = \bigcup_{S \in \mathcal{C}} S$
 6:     Let $n(k, \mathcal{C}) = k$-SC$(\mathcal{C}, R[\mathcal{C}])$
 7:     Let $n(k, \overline{\mathcal{C}}) = k$-SC$(\mathcal{F} \setminus \mathcal{C}, R[\mathcal{F} \setminus \mathcal{C}])$
 8:     return $\sum_{i=0}^{k} n(i, \mathcal{C}) \ * \ n(k-i, \overline{\mathcal{C}})$
 9: else if $\exists u, v \in R$ : $\mathcal{F}[u] \subseteq \mathcal{F}[v]$ then //subsumption
10:     return $k$-SC$(\{S \setminus \{v\} \mid S \in \mathcal{F}\}, R \setminus \{v\}))$
11: end if
12:
13: //branching or path decomposition
14: Let $e \in R$ be an element of maximum frequency
15: Let $S \in \mathcal{F}$ be a set of maximum size that is not an exceptional case[1]
16: Preference list P: $\mathsf{S}_4 < \mathsf{S}_5 < \mathsf{S}_6 < \mathsf{e}_5 < \mathsf{e}_6 < \mathsf{S}_7 < \mathsf{e}_7 < \mathsf{S}_{\geq 8} < \mathsf{e}_{\geq 8}$
17: if $S_{|S|}$ is in the list P and $S_{|S|} \not< e_{\text{freq}(e)}$ then //set branch
18:     $n_{take} = \sum_{i=1}^{k-1} i$-SC$(\{S' \setminus S \mid S' \in \mathcal{F}\}, R \setminus S)$
19:     $n_{discard} = k$-SC$(\mathcal{F} \setminus S$,$R)$
20:     return $n_{take} + n_{discard}$
21: else if $e_{\text{freq}(e)}$ is in the list P and $e_{\text{freq}(e)} \not< S_{|S|}$ then //element branch
22:     $n_{optional} = k$-SC({ $S' \setminus \{e\} \mid S' \in \mathcal{F}$ } , $R \setminus \{e\}$)
23:     $n_{forbidden} = k$-SC$(\mathcal{F} \setminus \mathcal{F}[e], R \setminus \{e\})$
24:     return $n_{optional} - n_{forbidden}$
25: else //path decomposition
26:     return SC-PW$(\mathcal{F}, R)$
27: end if
```

[1] There are 14 exceptional combinations of frequencies of the elements of a set of cardinality 5 we prefer to handle in the path decomposition phase.

Our algorithm is listed in Algorithm 3. We denote k-SC($\mathcal{F}$,R) for #k-Set Cover with sets $\mathcal{F}$ and nodes R to be covered. $\mathcal{F}[u]$ is the set of all elements of $\mathcal{F}$ that contain u, and $\mathrm{freq}(e) = |\mathcal{F}[u]|$.

Let us start with discussing the reduction rules: if there exists a **unique element**, that is, an element which is contained in only one set, this set has to be at least once chosen. If there is a part which can be solved independently, that is, there is a (non-trivial) **connected component**, solves this independently. Moreover, if there are two elements $u, v \in R$ such that if u is covered, v is covered, we can safely remove v.

If the reduction rules are not applicable, we find an element of maximum frequency and a set of maximum size, and choose on which we branch using the **preference list** P. If both are not in the list, we use algorithm `SC-PW`, which is discussed in Subsection 5.3.

5.3 Base case: path decomposition

If an efficient branch is not possible, that is, there do not exist any large sets or large frequency elements, Algorithm 3 calls the procedure `SC-PW`, which will be discussed in this section. Note that we solve the same problem as Algorithm 3, with the extra information that the element of maximum frequency and the set of maximum size are too small to be in the preference list P. To exploit this information, we use the following notion:

Definition 5.2. *A **path decomposition** of a graph $G = (V, E)$ is a sequence of sets of vertices $P = \langle P_1, \ldots, P_r \rangle$ such that:*

- $\cup_{i=1}^{r} P_i = V$
- *For each $(u, v) \in E$, there exists a X_i such that $\{u, v\} \subseteq P_i$*
- *if $v \in P_i$ and $v \in P_k$ then $v \in P_j$, for each $i \leq j \leq k$*

*P_i is said to **introduce** $v \notin P_{i-1}$ if $P_i = P_{i-1} \cup \{v\}$ and said to **forget** $v \in P_{i-1}$ if $P_i = P_{i-1} \setminus \{v\}$. If $|P_1| = 1$ and for $2 \leq i \leq r$, P_i either introduces or forgets a node v, then P is called a **nice path decomposition**. The **width** of P is $max_{1 \leq i \leq r} P_i - 1$ and the pathwidth $\mathbf{pw}(G)$ of G is the minimum over the width of all its path decompositions.*

It is well known that any path decomposition can be converted to a nice path decomposition of equal width. We implement `SC-PW` by dynamic programming on a path decomposition. However, we first have to convert the set-cover instance to a graph:

Definition 5.3. *Given sets $\mathcal{F}$ and a set R, the **incidence graph** $G_{\mathcal{F}}$ is the bipartite graph with **red vertices** $V_{Red} = \mathcal{F}$ and **blue vertices** $V_{Blue} = R$. Vertices $S \in V_{Red}$*

*and $u \in V_{Blue}$ are adjacent if and only if $u \in S$. A **Red-Blue Dominating Set** is a set $D \subseteq V_{Red}$ such that each node of V_{Blue} is adjacent to a vertex of D.*

Note that each Red-Blue Dominating Set of $G_{\mathcal{F}}$ corresponds directly to an equal-sized set cover of $\mathcal{S}$. Now, suppose we have a nice path decomposition $\langle P_1, \dots, P_r \rangle$ of $G_{\mathcal{F}}$. Denote Y_i for $\cup_{j=1}^{i} P_j \setminus P_i$, i.e. all nodes in the previous bags and not in the current one. Given subsets X_{Red} and X_{Blue} of the red and blue vertices of bag P_i, we define $A(X_{Red}, X_{Blue}, k, i)$ as the number of k-sized Red-Blue Dominating Sets S of the graph induced by $Y_i \cup X_{Red} \cup X_{Blue}$ such that $S \cap P_i = X_{Red}$. All possible terms of A can be computed with Algorithm 4.

Lemma 5.4. *If we are given $G_{\mathcal{S}}$ with a path decomposition P of width at most p, the number of Red-Blue Dominating Sets of given size k can be counted in $\mathcal{O}^*(2^p)$ time.*

Proof. Note that the following is equal to the number of Red-Blue Dominating Sets:

$$\sum_{X_{Red} \subseteq P_r \cap V_{Red}} A(X_{Red}, V_{Blue} \cap P_i, k, r)$$

If we convert P to nice path decomposition, all terms $A(X_{Red}, V_{Blue} \cap X_i, k, i)$ can be computed in $\mathcal{O}^*(2^p)$ using Algorithm 4 in combination with dynamic programming. Now the sum can be computed within the same time bound. ☐

5.4 Memorization

On top of Algorithm 3, we use **memorization**: Fix a parameter h'. Create a database with all instances that can be generated from the original problem, with complexity smaller or equal to h', for some measure of complexity. Now the time need to solve all of these problems simultaneously is polynomially bounded by their amount. To upper bound this quantity, we use the following lemma of Robson [Rob86]:

Lemma 5.5. *Let G be a graph of maximum degree $d \geq 3$. Then we can bound the number of connected induced subgraphs s_h of G on h vertices by:*

$$s_h = \mathcal{O}\left(\left(\frac{(d-1)^{d-1}}{(d-2)^{d-2}} \right)^h n^{\mathcal{O}(1)} \right)$$

5.5 Measure & conquer analysis

Theorem 5.6. *Algorithm 3 computes the number of dominating sets of given size in an n vertex graph G in $\mathcal{O}(1.4890^n)$ when it is applied to a set cover formulation of this problem.*

Algorithm 4 $A(X_{Red}, X_{Blue}, k, i)$

Input: Integers $1 \leq i \leq r$ and k, subsets X_{Red} and X_{Blue} of the red and blue nodes of X_i.

Output: The number of k-sized Red-Blue Dominating Sets S of the graph induced by $Y_i \cup X_{Red} \cup X_{Blue}$ such that $S \cap P_i = X_{Red}$

```
1: if $i = 1$ then
2:    //the only two non-zero base cases, using $|P_i| = 1$
3:    if ($k = 1$ and $|X_{Red}| = 1$) or ($k = 0$ and $|X_{Blue}| = 0$) then
4:       return 1
5:    end if
6: else if $X_i$ introduces $v \in V_{Red}$ then
7:    //if $v \in X_{Red}$,it is not allowed to dominate vertices not in $X_{Blue}$
8:    if $v \in X_{Red}$ and $N(v) \cap P_i \subseteq X_{Blue}$ then
9:       //take $v$ at least once, dominate a subset of its neighbors
10:      return $\sum_{j=1}^{k} \sum_{B \subseteq N(v) \cap X_i} A(X_{Red} \setminus \{v\}, X_{Blue} \setminus B, k-j, i-1)$
11:   else if $v \notin X_{Red}$ then
12:      //we do not take $v$
13:      return $A(X_{Red}, X_{Blue}, k, i-1)$
14:   end if
15: else if $X_i$ introduces $v \in V_{Blue}$ then
16:   //detects whether $v$ is actually dominated
17:   if $v \in X_{Blue}$ and $N(v) \cap X_{Red} \neq \emptyset$ then
18:      return $A(X_{Red}, X_{Blue} \setminus \{v\}, k, i-1)$
19:   else if $v \notin X_{Blue}$ and $N(v) \cap X_{Red} = \emptyset$ then
20:      return $A(X_{Red}, X_{Blue} \setminus \{v\}, k, i-1)$
21:   end if
22: else if $X_i$ forgets $v \in V_{red}$ then
23:   //$v$ can be taken or discarded
24:   return $A(X_{Red} \cup \{v\}, X_{Blue}, k, i-1) + A(X_{Red}, X_{Blue}, k, i-1)$
25: else if $X_i$ forgets $v \in V_{blue}$ then
26:   //$v$ still has to be dominated
27:   return $A(X_{Red}, X_{Blue} \cup \{v\}, k, i-1)$
28: end if
29: //if none of the above applies
30: return 0
```

Proof. We will prove that the algorithm counts the number of set covers of given size in a set cover instance with sets $\mathcal{F}$ and elements R in time $\mathcal{O}(1.2204^d)$, where d is the maximum of the number of sets and the number of elements. This results in a running time of $\mathcal{O}(1.22024^{2n}) < \mathcal{O}(1.4890^n)$ for counting the number of dominating sets.

We analyze the algorithm using measure and conquer from [FGK04], i.e. we introduce a non standard complexity measure $k(\mathcal{F}, R)$ on problem instances. To this end, we introduce weight functions $v, w : \mathbb{N} \to [0, 1]$ giving weight $v(i)$ to an element of frequency i and weight $w(i)$ to a set of cardinality i, respectively. This gives us the following complexity measure:

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

This measure is also used in some other publications [FGK04, vRB08]. From the fact that $k \leq d$ follows that a proven running time of $\mathcal{O}(\alpha^k)$ also proves a running of $\mathcal{O}(\alpha^d)$. For convenience reasons we let $\Delta v(i) = v(i) - v(i-1)$, $\Delta w(i) = w(i) - w(i-1)$ be the complexity reduction gained by reducing the frequency of an element or the cardinality of a set by one.

We first prove the following lemma analyzing the branching phase.

Lemma 5.7. *Let $N_h(k)$ be number of subproblems of measured complexity h generated by Algorithm 3 on an input of measured complexity k. Then: $N_h(k) < 1.22770^{k-h}$.*

Proof. Before analyzing the actual branching, we need some constraints on the weights. Firstly, we observe that elements of frequency one are removed by the reduction rules and sets of cardinality zero are ignored in the branching phase. Therefore, we set their measured complexity to zero: $v(0) = v(1) = w(0) = 0$. Secondly, we do not want the complexity of our instance to increase when decreasing the frequency of an element or the cardinality of a set and impose the restrictions $\Delta v(i), \Delta w(i) \geq 0$. Finally we impose the restrictions $\Delta v(i) \geq \Delta v(i+1)$ and $\Delta w(i) \geq \Delta w(i+1)$ for $i \geq 1$, and $2\Delta v(5) \geq v(2)$. The functions of these last constraints will become clear from the analysis.

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

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

Here we bound each extra reduction of the frequency an element because of the removal of a set by $\Delta v(\mathrm{freq}(e))$. Notice that we branch on an element of highest frequency which is at least five and we have the constraints $\Delta v(i) \geq \Delta v(i+1)$ and $2\Delta v(5) \geq v(2)$. The first one assures that each time we reduce the cardinality of a set more than one time, the $\Delta v(i)$ for $i < \mathrm{freq}(e)$ are large enough. And the second one compensates for the fact that an element can be completely removed if it occurs only in removed sets while $v(1) = 0$. Without this constraint we could have taken all the weight from this element already, leaving none for its final occurrence.

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

Using the constraints $\Delta w(i) \geq \Delta w(i+1)$, this leads to the following values for $\Delta k_{discard}$, Δk_{take}:

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

By the above considerations we have:

$$N_h(k) \leq N_h(k - \Delta k_1) + N_h(k - \Delta k_2)$$

with Δk_1 and Δk_2 being $\Delta k_{optional}$ and $\Delta k_{forbidden}$ for every possible element branch, and $\Delta k_{discard}$ and Δk_{take} for every possible set branch. The solution of this set of recurrence relations is a function of the form α^{k-h} where α is the largest positive real root of the corresponding set of equations:

$$1 = \alpha^{-\Delta k_1} + \alpha^{-\Delta k_2}$$

for all $|S| = \sum_{i=2}^{\infty} e_i$ and $\mathrm{freq}(e) = \sum_{i=1}^{\infty} s_i$ agreeing with the preference order of branching:

$$\mathsf{S}_4 < \mathsf{S}_5 < \mathsf{S}_6 < \mathsf{E}_5 < \mathsf{E}_6 < \mathsf{S}_7 < \mathsf{E}_7 < \mathsf{S}_{\geq 8} < \mathsf{E}_{\geq 8}$$

Furthermore, we have the following considerations when branching on an element: it can be in at most one set of size one, and the element can not be in any of the 14 exceptional cases. These 14 exceptional cases are all elements of frequency five that:

1. only occurs in sets of size at most three with at least one occurrence is a set of size at most two. (10 cases)

2. four other cases containing elements of frequency at most four denoted by the 4-tupples (e_1, e_2, e_3, e_4):

$$(1,3,0,1) \quad (1,2,1,1) \quad (0,4,0,1) \quad (0,3,1,1)$$

The weight functions will converge to 1 at some point p resulting in $w(p) = v(p) = w(p+1) = v(p+1) = 1$ and $\Delta w(p+1) = \Delta v(p+1)$. Therefore the recurrence relations corresponding to $|S| > p+1$ and $\mathrm{freq}(e) > p+1$ are dominated by those corresponding to $|S| > p$ and $\mathrm{freq}(e) = p$, respectively. This leads to a large, but finite, numerical optimisation problem (quasi-convex program [Epp04]) of choosing the optimal weights. We solve this by computer obtaining an upper bound on the number of subproblems of measured complexity h generated of 1.22770^{k-h} and the following set of weights:

i	1	2	3	4	5	6	≥ 7
$v(i)$		0.403961	0.773268	0.987570	0.996364	1	1
$w(i)$	0.410162	0.657131	0.805891	0.890186	0.974481	0.997053	1

This concludes the analysis of the branching phase. □

The next thing we need is a bound on the running time of SC-PW$(\mathcal{S}, R)$. We use the following lemma:

Lemma 5.8. *For any $\epsilon > 0$, there exists and integer n such that for every graph G with $n > n_\epsilon$ vertices,*

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

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

Lemma 5.9. *SC-PW$(\mathcal{S}, R)$ runs in time $\mathcal{O}(1.2111^k)$ when called by Algorithm 3 on a set cover instance of measured complexity k.*

Proof. By considerations from Section 5.3, we know that the running time of this part of the algorithm is $\mathcal{O}^*(2^p)$, where p is the pathwidth of the incidence graph of a generated subproblem. Therefore we will now prove an upper bound on the pathwidth of the incidence graph of a set cover instance of measured complexity k that remains after the branching phase. To this end we formulate a linear program in which all variables have the domain $[0, 1]$. In a simpler form this was also done in [FS07].

$$\begin{aligned}
\max \quad z &= \frac{1}{6}(x_3 + y_3) + \frac{1}{3}(x_4 + y_4) + \frac{13}{30}y_5 \qquad \text{such that:} && (5.1)\\
1 &= \sum_{i=1}^{4} w(i)x_i + \sum_{i=2}^{5} v(i)y_i && (5.2)\\
\sum_{i=1}^{4} ix_i &= \sum_{i=2}^{5} iy_i && (5.3)
\end{aligned}$$

Using Lemma 5.8 we compute the maximum pathwidth of a graph per unit of its measured complexity z: see Formula 5.1. Although the lemma only applies to graphs of size at least n_ϵ, this is correct because we consider all smaller graphs to be handled in constant time. Here x_k and y_k represent the number of sets of cardinality k and elements of frequency k per unit of measured complexity in a worst case instance, respectively. Notice that Constraint 5.2 guarantees that these x_k and y_k use exactly one unit of measured complexity. And, notice that a subproblem on which SC-PW($\mathcal{F}$,R) is called can have sets of cardinality at most four and elements of frequency at most five. Additionally, we know that both partitions of the bipartite incidence graph must have an equal number of edges, which we guarantee by Constraint 5.3. As a result a dynamic program on this path decomposition runs in time $\mathcal{O}^*(2^{zk})$.

Finally, we have to add additional constraints because elements of frequency five can only occur in the instance if these are one of the 14 exceptional cases and cardinality four sets can only exist if these contain these exceptional frequency five elements. Therefore we add variables p_i representing the number of elements of frequency 5 per unit of measured complexity of exceptional case i $(1 \leq i \leq 14)$. Naturally, their sum should equal y_5 which is forced by Equation 5.4. We now use these p_i by observing that whenever $p_i > 0$ for some i we have further restrictions on the instance. Namely, given an exceptional case of a frequency five element we know the cardinalities of the sets in which it occurs. So, we let $[s_i]_j$ be the number of occurrences of a exceptional case i frequency five element in a set of cardinality j. Now we can lower bound the number of sets of cardinalities one, two and three in the instance using Constraint 5.5. Furthermore we can upper bound the number of sets of cardinality four since there can be at most one of these per exceptional frequency five elements contained in it. This is done in Constraint 5.6

$$y_5 = \sum_{i=1}^{14} p_i \tag{5.4}$$

$$x_j \geq \sum_{i=1}^{14} [s_i]_j \frac{p_i}{j} \qquad \text{for } i \in \{1,2,3\} \tag{5.5}$$

$$x_4 \leq \sum_{i=1}^{14} [s_i]_4 \tag{5.6}$$

The solution to this linear program is $z = 0.27624$ with all variables equal to zero, except:

$$x_2 = 0.386443 \qquad x_3 = 0.085876 \qquad x_4 = 0.257629$$
$$y_4 = 0.193222 \qquad y_5 = 0.257629 \qquad p_{14} = 0.257629$$

This results in a running time of $\mathcal{O}(1.2111^k)$ and completes the proof of the lemma. $\square$

What remains to be done is to combine the results of Lemmas 5.7, 5.9 also using memorization. Let $T(k)$ be the total time spend to solve a problem of measured complexity k and let h' be the measured complexity of the largest precomputed instance. We divide the analysis in two parts using $T(k) = T_{>h'}(k) + T_{\leq h'}(k)$, where $T_{>h'}(k)$ and $T_{\leq h'}(k)$ denote the total time spend solving instances generated by branching of measured complexity greater than h' and smaller or equal to h', respectively. I.e., $T_{\leq h'}(k)$ is the total time used to generate the precomputed instances, and $T_{>h'}(k)$ is the total time spend in all calls to $\texttt{SC-PW}(\mathcal{S})$. Also, let H_k be the set of all possible measured complexities of subproblems generated by branching on a problem of measured complexity k. Notice that $|H_k|$ is polynomially bounded because there are only a finite number of distinct weights involved.

$$\begin{aligned} T(k) &= T_{>h'} + T_{\leq h'} \leq \sum_{h \in H_k, h > h'} 1.22770^{k-h} 1.2111^h + \sum_{h \in H_k, h \leq h'} \left(\frac{6^6}{5^5}\right)^{\frac{h}{w}} \\ &< |H| 1.22770^{k-h'} + |H| \left(\frac{6^6}{5^5}\right)^{\frac{h'}{w}} \end{aligned} \tag{5.7}$$

In the above equations we use Lemmas 5.7 and 5.9 to bound $T_{>h'}$. Denoting the smallest positive weight by w, the maximum number of vertices in an incidence graph of complexity h' is $\frac{h'}{w}$. We use this together with Lemma 5.5 to bound $T_{\leq h'}$.

What remains is to find the optimal value of h'. Hereto we let $h' = \alpha k$ and compute α such that both parts of Equation 5.7 are in balance, i.e.:

$$1.22770^{1-\alpha} = \left(\frac{6^6}{5^5}\right)^{\frac{\alpha}{w}}$$

Solving this equation we obtain $\alpha = 0.029742$ and $1.22770^{1-\alpha} = 1.22024$. Hence:

$$T(k) \quad < \quad 1.22770^{(1-\alpha)k} = \left(\frac{6^6}{5^5}\right)^{(\frac{\alpha}{w})k} < 1.22024^k n^{\mathcal{O}(1)}$$

This completes the proof. □

Notice that a lot of effort has been spent to properly balance this algorithm. The asymmetry between sets and elements due to the reduction rules requires a carefully balanced branching preference order. Moreover, branching on some elements of frequency five gives very bad branches, especially when many small sets are involved. These cases can, however, be treated quite efficiently in the path decomposition phase since any size one or two sets do not contribute to the pathwidth of the incidence graph while they do have an amount of measured complexity. Carefully selecting cases to be handled in either way resulted in the 14 exceptional cases mentioned in the description of the algorithm.

Conclusions and further research

We have seen that there are little non-trivial polynomial counting algorithms, indicating that, in general, counting problems are particularly difficult. Unfortunately, inclusion exclusion can only solve counting problems. This is the main reason why the practical use of inclusion-exclusion for decision problems is still questionable, because it is not clear how to be combined with heuristics and techniques such as branch-and-bound.

However, IE-branch & reduce will be much more efficient in practice than exhaustive IE-branching. Its main advantage is the possibility to branch on requirements instead of variables. Although we can adjust the universe to improve the running time, at this moment this does not outweigh the additional techniques that can be used in normal branches i.e. branch & bound combined with heuristics.

IE-branch & reduce can easily be used to extend a branch & reduce algorithm. Moreover, for a decision problem, it may even make sense to extend it a counting problem such that efficient IE-branching can be used. This is also illustrated with the dominating set algorithm.

The set cover problem is a particularly interesting application of inclusion exclusion. Many set cover problems can be solved with the set cover formulation. We can also use this for an IE-branch & reduce algorithm which is similar to one of the simplified problem, i.e. many reduction rules for the simplified problem can be used.

If the simplified problem is $\#\mathcal{P}$-complete we can often use dynamic programming to compute all simplified problems simultaneously. For a set cover formulation, we have seen that this can also be combined with a IE-branch & reduce algorithm.

The most appealing open question is: Can IE-branch & reduce be used to obtain an $o^*(2^n)$ algoritme voor graph coloring? The difference with other set-cover formulations, such as the one of dominating set, is the exponential amount of sets. For this exponential number, one should achieve recurrences for the running time that reduce the number of sets with a multiplicative factor.

Can IE-branch & reduce be used to improve other branching algorithms? There exist many branching algorithms for problems similar to dominating set, and one could extend such an algorithm with IE-branching, using the set cover formulation.

Can inclusion-exclusion be used to obtain better combinatorial upper bounds, for example the number of minimal dominating sets [FGPS05]. Note that if we use the equation

$$Required = Optional - Forbidden$$

we can obtain an upper bound for $Required$ by subtracting a lower bound of $Forbidden$ from an upper bound of $Optional$. A lower bound for $Required$ can be achieved similarly.

Can the number of elements that meet exactly k requirements be computed efficiently, given any IE-branch & reduce algorithm? In the textbook [Gri94], Grimaldi gave a simple formula if we use the trivial sumspace. However, it is not clear how this can be combined with the techniques of Chapter 4.

Bibliography

[AB07] S. Arora and B. Barak. Computational Complexity: A Modern Approach. *to appear: http://www.cs.princeton.edu/theory/complexity*, 2007. [cited at p. 3]

[And04] J.A. Anderson. *Discrete mathematics with combinatorics*. Pearson Education Upper Saddle River, NJ, 2004. [cited at p. 17]

[Bax96] E. Bax. Recurrence-Based Reductions for Inclusion and Exclusion Algorithms Applied to #P Problems. 1996. [cited at p. 12, 17, 20]

[BH06a] A. Björklund and T. Husfeldt. Exact algorithms for exact satisfiability and number of perfect matchings. *Lecture Notes in Computer Science*, 4051:548–559, 2006. [cited at p. 17, 36]

[BH06b] A. Björklund and T. Husfeldt. Inclusion–exclusion algorithms for counting set partitions. In *FOCS* [DBL06], pages 575–582. [cited at p. ii]

[BHK06] A. Björklund, T. Husfeldt, and M. Koivisto. Set partitioning via inclusion-exclusion. Special Issue for [DBL06], October 2006. [cited at p. ii, 22]

[BHKK07] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. Fourier meets möbius: fast subset convolution. *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 67–74, 2007. [cited at p. 24]

[BHKK08a] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. Trimmed Moebius Inversion and Graphs of Bounded Degree. *Arxiv preprint arXiv:0802.2834*, 2008. [cited at p. 32, 35]

[BHKK08b] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. The travelling salesman problem in bounded degree graphs. In Luca Aceto et al., editor, *ICALP (1)*, volume 5125 of *Lecture Notes in Computer Science*, pages 198–209. Springer, 2008. [cited at p. 20]

[CG05] T. Calders and B. Goethals. Quick inclusion-exclusion. *Proceedings ECML-PKDD 2005 Workshop Knowledge Discovery in Inductive Databases*, 3933, 2005. [cited at p. 17]

[Com03] C.W. Commander. *A survey of the quadratic assignment problem, with applications*. PhD thesis, University of Florida, 2003. [cited at p. 1]

[DBL06] *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2006), 21-24 October 2006, Berkeley, California, USA, Proceedings*. IEEE Computer Society, 2006. [cited at p. 50, 52]

[DJW05] V. Dahllöf, P. Jonsson, and M. Wahlström. Counting models for 2SAT and 3SAT formulae. *Theoretical Computer Science*, 332(1-3):265–291, 2005. [cited at p. 7]

[Doh99] K. Dohmen. An improvement of the inclusion-exclusion principle. *Arch. Math.*, 72, 1999. [cited at p. 36]

[Doh03] K. Dohmen. *Improved Bonferroni Inequalities Via Abstract Tubes: Inequalities and Identities of Inclusion-Exclusion Type*. Springer, 2003. [cited at p. 17, 29, 30, 31]

[Epp04] D. Eppstein. Quasiconvex analysis of backtracking algorithms. *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004*, pages 781–790, 2004. [cited at p. 45]

[FGK04] F.V. Fomin, F. Grandoni, and D. Kratsch. Measure and conquer: Domination — a case study. Proceedings of the 32nd International Colloquium on Automata, Languages and Programming, ICALP 2005, 3580:191–203, 2004. [cited at p. 38, 42, 43]

[FGK06] F.V. Fomin, F. Grandoni, and D. Kratsch. Measure and conquer: a simple O (2 0.288n) independent set algorithm. *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 18–25, 2006. [cited at p. 33]

[FGPS05] F.V. Fomin, F. Grandoni, A. Pyatkin, and A Stepanov. Bounding the Number of Minimal Dominating Sets: A Measure and Conquer Approach. *Algorithms and Computation*, pages 573–582, 2005. [cited at p. 49]

[FK05] M. Furer and S.P. Kasiviswanathan. Algorithms for counting 2-SAT solutions and colorings with applications. *Electronic Colloquium on Computational Complexity (ECCC)*, 33:1, 2005. [cited at p. 7]

[FKW04] F.V. Fomin, D. Kratsch, and G.J. Woeginger. Exact (exponential) algorithms for the dominating set problem. In *in Proceedings of the 30th Workshop on Graph Theoretic Concepts in Computer Science (WG 2004)*, pages 245–256. Springer, 2004. [cited at p. 38]

[FS07] F.V. Fomin and A.A Stepanov. Counting minimum weighted dominating sets. In Guohui Lin, editor, *COCOON*, volume 4598 of *Lecture Notes in Computer Science*, pages 165–175. Springer, 2007. [cited at p. 38, 39, 45]

[GKL08] S. Gaspers, D. Kratsch, and M. Liedloff. On independent sets and bicliques in graphs. *The 34th International Workshop on Graph-Theoretic Concepts in Computer Science WG 2008*, 2008. [cited at p. 33]

[Gra06] F. Grandoni. A note on the complexity of minimum dominating set. *Journal of Discrete Algorithms*, 4(2):209–214, 2006. [cited at p. 38]

[Gri94] R.P. Grimaldi. *Discrete and combinatorial mathematics*. Addison-Wesley Reading, Mass, 1994. [cited at p. 17, 49]

[HK62] M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962. [cited at p. 4]

[Kar82] R. M. Karp. Dynamic programming meets the principle of inclusion and exclusion. *Oper. Res. Lett.*, 1:49–51, 1982. [cited at p. 12, 17, 19, 20, 21]

[KGK77] S. Kohn, A. Gottlieb, and M. Kohn. A generating function approach to the traveling salesman problem. In *ACM '77: Proceedings of the 1977 annual conference*, pages 294–300, New York, NY, USA, 1977. ACM. [cited at p. 20]

[Koi06] M. Koivisto. An $\mathcal{O}^*(2^n)$ algorithm for graph coloring and other partitioning problems via inclusion–exclusion. In *FOCS* [DBL06], pages 583–590. [cited at p. ii, 17]

[Nar82] H. Narushima. Principle of inclusion-exclusion on partially ordered sets. *Discrete Math*, 42:243–250, 1982. [cited at p. 26]

[NW92] D.Q. Naiman and H.P. Wynn. Inclusion-Exclusion-Bonferroni Identities and Inequalities for Discrete Tube-Like Problems via Euler Characteristics. *The Annals of Statistics*, 20(1):43–76, 1992. [cited at p. 28]

[NW97] D.Q. Naiman and H.P. Wynn. Abstract Tubes, Improved Inclusion-Exclusion Identities and Inequalities and Importance Sampling. *The Annals of Statistics*, 25(5):1954–1983, 1997. [cited at p. 28, 29]

[Rob86] J.M. Robson. algorithms for maximum independent sets. *Journal of algorithms(Print)*, 7(3):425–440, 1986. [cited at p. 42]

[RS04] B. Randerath and I. Schiermeyer. Exact algorithms for minimum dominating set. Technical report, April 2004. [cited at p. 38]

[Tak67] L. Takacs. On the method of inclusion and exclusion. *Journal of the American Statistical Association*, 62(317):102–13, 1967. [cited at p. 17]

[Tuc94] A. Tucker. *Applied combinatorics*. John Wiley & Sons, Inc. New York, NY, USA, 1994. [cited at p. 17]

[Vad01] S. Vadhan. The complexity of counting in sparse, regular, and planar graphs. *SIAM J. Comput.*, 31(2):398–427, 2001. [cited at p. 26]

[Val79] L.G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201, 1979. [cited at p. 3, 4]

[vR06] J. M. M. van Rooij. Design by measure and conquer: An o(1.5086n) algorithm for minimum dominating set and similar problems. Master's thesis, Institute for Information and Computing Sciences, Utrecht University, 2006. [cited at p. 38]

[vRB08] J. M. M. van Rooij and H. L. Bodlaender. Design by measure and conquer: a faster exact algorithm for dominating set. Proc. 25th Symp. Theoretical Aspects of Computer Science, STACS 2008, pages 657–668., 2008. [cited at p. 38, 43]

[Whi32] H. Whitney. A logical expansion in mathematics. *Bull. Amer. Math. Soc*, 38:572–579, 1932. [cited at p. 17, 22, 30]

[Wil05] R. Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theoretical Computer Science*, 348(2-3):357–365, 2005. [cited at p. 6]

[Woe03] G.J. Woeginger. Exact algorithms for NP-hard problems: A survey. Combinatorial Optimization: "Eureka, you shrink", LNCS 2570, 2003. [cited at p. 1]