The Sparsification Lemma for CNF Satisfiability

Zeijlemaker, Sjanne

Award date:
2020

Eindhoven University of Technology

The Sparsification Lemma for CNF Satisfiability

Author:
Sjanne Zeijlemaker

Supervisors:
Dr. N. Bansal
Dr. B.M.P. Jansen
Dr. J. Nederlof

July 9, 2020

Abstract

Many NP-hard problems have a natural notion of sparseness. For example, a graph is said to be sparse if it has few edges and a propositional formula in conjunctive normal form (CNF) is sparse if it has few clauses. Sparseness influences the difficulty of a problem instance in a nontrivial way. While graphs with a constant number of edges admit a polynomial time algorithm for many graph problems, the same can be said about the complete graph. In this thesis, we study how sparseness and difficulty are related in the range between these two extremes.

For CNF satisfiability, one of the most fundamental NP-hard problems, this link has been studied extensively. The Sparsification Lemma states that every CNF formula can be written as a subexponential-size disjunction of CNF formulas such that the number of clauses is at most linear in the number of variables. This shows that dense formulas are not necessarily more difficult to satisfy, as they can be reduced to a sparse core. On the other hand, Lovász Local Lemma shows that CNF formulas cannot be made arbitrarily sparse, as this makes them trivially satisfiable. The Sparsification Lemma also plays a fundamental role in the theoretical study of NP-hard problems, as it implies that many do not admit a $2^{o(n)}$-time algorithm if CNF satisfiability cannot be solved in $2^{o(n)}$ time.

In this thesis, we consider three variants of the Sparsification Lemma. First, we propose a sparsification theorem based on the Sunflower Lemma, which is more flexible in choosing parts of the formula on which decisions need to be made, at the expense of a worse sparsification constant. We then tailor the Sparsification Lemma to the Hitting-Set problem, adding the maximum size x of the hitting set as a parameter. If x is small, this sparsification algorithm produces significantly fewer outputs and has a faster running time than the Sparsification Lemma. Finally, we design a randomised sparsification algorithm such that every satisfying assignment of the input satisfies the output formula with probability at least $2^{-\varepsilon n}$. We also derive conditions under which k-Cnf-Sat and Hitting-Set are always satisfiable using Lovász Local Lemma.

Contents

1 Introduction

It is generally assumed that no efficient algorithms exist for NP-hard problems. However, this does not mean that an instance of these problems is difficult in its entirety. For example, when looking for a vertex cover[1] in a graph, isolated vertices do not contribute to the inherent difficulty of the problem, as they may be discarded immediately. For this reason, algorithms for NP-hard problems often start with a *preprocessing* stage. The aim of preprocessing is to reduce a problem to its difficult core by efficiently solving the easy parts. The algorithm then solves this smaller core, rather than the entire instance.

Preprocessing plays a particularly central role in the theoretical study of algorithms. The complexity of computational problems can be expressed as a function of several parameters, which often have different orders of magnitude. The number of edges in a graph can for example be quadratic in the number of vertices, but the number of non-isolated vertices equals at most twice the number of edges. Since isolated vertices can easily be discarded for most graph problems, this effectively means that the number of vertices is at most linear in the number of edges. We see a similar disbalance in propositional formulas in conjunctive normal form (CNF). A CNF formula on n variables with clauses of width k may have order n^k clauses, whereas a formula with m clauses always has at most $\mathcal{O}(m)$ variables.

To express the relation between parameters of a problem, we introduce the notion of sparseness. If a problem has one potentially large parameter, we call a problem instance *sparse* when this parameter is relatively small compared to the other parameters. Many mathematical objects have a natural measure of sparseness; a graph is sparse if its edge to vertex ratio is small and a set system is sparse if it contains relatively few sets. The difficulty of a problem instance often depends on its sparseness. For example, many hard graph problems can be solved in polynomial time on a graph with a constant number of edges. Curiously, the same thing holds for a complete graph. This illustrates that sparser does not automatically mean easier.

Sparseness is especially interesting in the context of CNF formulas. CNF satisfiability (Cnf-Sat), the problem of deciding whether there exists a variable assignment such that a CNF formula evaluates to true, is one of the most fundamental NP-hard problems. A CNF formula is called sparse if the number of clauses is linear in the number of variables and the process of transforming a CNF into a (number of) sparse one(s) is referred to as *sparsification*. A well-known sparsification result for CNF formulas is the *Sparsification Lemma*, which roughly states that for every $\varepsilon > 0$, a CNF formula can be rewritten as a disjunction of $2^{\varepsilon n}$ sparse CNF formulas such that every variable appears in at most dn clauses for a constant d. This lemma sheds an interesting light on the relation between the sparseness and difficulty of CNF satisfiability, as it shows that every dense instance can be transformed into a sparse one which is equally difficult. The algorithm which lies at the heart of this lemma repeatedly finds flowers, large collections of clauses with a common intersection, and replaces them with one or more smaller clauses. An open problem, which will play a central role in this thesis, is whether the constant d is optimal.

The Sparsification Lemma is an important tool in complexity theory. While it is widely believed that P $\neq$ NP, it is not clear whether certain NP-hard problems might have a subexponential algorithm which is not polynomial, for example with running time $2^{\sqrt{n}}$. It is conjectured that such an algorithm does not exist for 3-Cnf-Sat. Since any NP-complete problem can be reduced to 3-Cnf-Sat, this has consequences for many other problems in NP. Using the Sparsification Lemma, it can be shown that problems such as Vertex-Cover and Hitting-Set do not admit a $2^{o(n)}$-time algorithm if we assume that 3-Cnf-Sat cannot be solved in $2^{o(n)}$ time.

Goals and motivation. In this thesis, we will study how sparseness influences the difficulty of a problem in several contexts. One main goal is to take steps towards improving the constant d of the Sparsification Lemma. The starting point for our research is the following.

The algorithm of the Sparsification Lemma processes flowers in a fixed order. While this rigidness makes it easier to analyse the algorithm in a structural way, it might be limiting the algorithm's performance. A more flexible algorithm could be key to improving the sparsification constant. One possible way to achieve this, is by replacing the flowers with a different object. Flowers are a generalisation of other, more structured collections of sets, such as sunflowers, relaxed sunflowers and quasi-sunflowers. If these types of flowers were used for sparsification, it could result in more flexible algorithms, potentially with a better sparsification

[1] A set of vertices which includes at least one endpoint of every edge.

constant. In this thesis, we take a first step in this direction by analysing a sparsification algorithm which works with sunflowers and studying a sparsification lemma for DNF formulas based on quasi-sunflowers.

The sparsification constant cannot be improved indefinitely. Very sparse formulas with clauses of equal size are known to be trivially satisfiable. Intuitively, a similar result should hold for formulas with clauses of different sizes. Using the probabilistic method, we will show that this is indeed the case.

To simplify its proof, the Sparsification Lemma was originally stated for the problem Hitting-Set. CNF satisfiability can been seen as a special case of this problem. Since the lemma was written with the aim to sparsify CNF formulas, it does not incorporate an essential parameter of the Hitting-Set problem: the maximum size of the hitting set. We propose a parametrised sparsification algorithm which uses this extra information to achieve a smaller number of outputs and a faster running time.

1.1 Our results

In this thesis, we introduce a three new variants of the Sparsification Lemma. The first replaces the flowers in the branching process with sunflowers. This version has a more flexible algorithm, at the expense of a worse sparsification constant. Using the Sunflower Lemma, we show that it attains the same running time and number of outputs as the original Sparsification Lemma.

Secondly, we propose a sparsification algorithm for parametrised instances of Hitting-Set. If an upper bound x on the size of the hitting set is given, the number of outputs of the Sparsification Lemma can be reduced to $2^{\varepsilon x}$. This improvement is achieved by cutting short any recursion path which cannot lead to a 'yes'-instance. As $x \leq n$, our lemma never performs worse than the original Sparsification Lemma and for small values of x it is much more efficient.

Finally, we take a probabilistic approach at sparsification. We analyse a randomised sparsification algorithm which branches on the heart or petals of a flower according to an appropriately chosen probability distribution. Given some $\varepsilon > 0$, the algorithm preserves satisfiability with probability $2^{-\varepsilon n}$ for all truth assignments.

The current bounds on CNF formulas that are trivially satisfiable only hold for formulas where every clause has width k. This can be generalised to arbitrary instances of k-Sat. We will show that a CNF formula is satisfiable if every variable appears in at most $f_w := \lfloor 2^{w-2}/(w^2 k) \rfloor$ clauses of width w. A similar bound is proved for (k, x)-Hitting-Set.

1.2 Related work

The Sparsification Lemma. The most notable sparsification result is a lemma by Impagliazzo, Paturi and Zane [16] known as the Sparsification Lemma. This lemma states that there exists an algorithm which, given $k \in \mathbb{N}$, $\varepsilon > 0$ and a k-CNF formula φ on n variables, computes a disjunction of at most $2^{\varepsilon n}$ sparse k-CNF formulas $\varphi_1, \ldots, \varphi_l$ with at most $\left(\frac{k \log 1/\varepsilon}{\varepsilon}\right)^{\mathcal{O}(k)}$ clauses such that any assignment x satisfies φ if and only if it satisfies φ_i for some i. This algorithm runs in $\mathcal{O}^*(l)$ time. The core observation behind the algorithm is the following.

Suppose that there exists a large collection of equal-sized clauses with nonempty common intersection. Such a collection is referred to as a flower and the common intersection as the heart. The part of a clause which is not contained in the heart is called a petal. An assignment which satisfies the flower must either satisfy the heart or every petal. Conversely, if either of these two is satisfied, then so is the flower. The algorithm described in the Sparsification Lemma repeatedly finds large flowers and forms two new CNF formulas by replacing the flower by its heart and by removing the heart from every clause of the flower. At least one of these two formulas is satisfiable if and only if the original CNF formula is satisfiable.

The Sparsification Lemma is often applied in proofs under a complexity assumption known as the Exponential Time Hypothesis. ETH was first formulated by Impagliazzo and Paturi [15] and implies that 3-Sat cannot be solved in $2^{o(n)}$ time. Assuming ETH, it can be shown that many NP-hard problems, such as Vertex-Cover and Hitting-Set, do no have a $2^{o(n+m)}$ algorithm. In general, a problem P with instances of size x cannot be solved in $2^{o(f(x))}$ time if there exists an polynomial-time reduction from 3-Sat to an instance of P of size $\mathcal{O}\left(f^{-1}(n+m)\right)$. A stronger statement known as the Strong Exponential Time Hypothesis conjectures that for all $\varepsilon > 0$ there exists a k such that k-Sat cannot be solved in $\mathcal{O}\left(2^{(1-\varepsilon)n}\right)$ time.

SETH has been studied extensively because of its implications for problems in P. Assuming SETH, problems such as Edit-Distance and Orthogonal-Vectors do not admit a subquadratic algorithm. While ETH is generally considered a plausible assumption, there is much uncertainty about SETH.

Sunflowers. Flowers are also studied in a broader context especially, in the field of extremal set theory. A fundamental result in this field is a lemma by Erdős and Rado [9] known as the Sunflower Lemma. A flower is called a sunflower if its petals are pairwise disjoint. The Sunflower Lemma states that any k-uniform set system with more than $k!(s-1)^k$ sets contains a sunflower with s petals. This bound was recently improved to $(Cs^3 \log k \log\log k)^k$ for some constant C by Alweiss, Lovett, Wu and Zhang [3]. It is a major open problem whether the base of the exponent can be reduced to a constant C depending on s only.

Sunflowers have a distinct behaviour with regard to random sets. A randomly sampled set S which includes every element of the flower with probability $1/2$ contains an entire petal with large probability. This is due to the fact that sunflower petals are disjoint. Flowers which mimic this behaviour are referred to as quasi-sunflowers. A flower is a γ-quasi-sunflower if S contains at least one petal with probability $1 - e^{-\gamma}$. Rossman [20] proved that every set system with m sets of size at most k contains a $\gamma(m)$-quasi-sunflower with $\gamma(m) := 1/5 \cdot (m/(k!))^{1/k}$. This Theorem is also known as the Quasi-sunflower Lemma.

Difficulty of sparse instances. The relation between sparseness and difficulty is sometimes counter-intuitive. It would be logical to assume that very dense instances are the hardest to solve, since they are simply larger. However, one particular situation for which this intuition fails is the most extreme case: for a complete graph, the densest graph possible, finding a vertex cover and independent set becomes trivial. The Sparsification Lemma shows that this idea is false in general, as every dense CNF formula can be reduced to a sparse core which is equally difficult. It appears that after a certain point, increasing the density no longer increases the hardness of a problem.

Nevertheless, a certain level of denseness is required to make a problem difficult. Using a probabilistic tool known as the Lovász Local Lemma, it has been shown that very sparse CNF formulas are essentially trivial. A classic textbook result of the lemma shows that CNF formulas with clauses of size exactly k are always satisfiable if every variable appears in at most $\lfloor 2^k/(ke) \rfloor$ clauses. This means that CNF formulas cannot be made arbitrarily sparse. A slightly better bound was achieved by Gebauer, Szabó and Tardos [11], who proved that CNF formulas with clauses of width k are always satisfiable if every variable appears in at most $\lfloor 2^{k+1}/((k+1)e) \rfloor$ clauses. This bound is known to be tight.

DNF sparsification. The term 'sparsification' may also refer to a different type of preprocessing used for DNF formulas. Contrary to the Sparsification Lemma, it takes a probabilistic approach to compute sparse formulas that give the same truth value with high probability under a random assignment. In this context, it is convenient to view propositional formulas as functions from $\{0,1\}^n$ to $\{0,1\}$. Given DNF formula f, we call two functions $f_l \leq f \leq f_u$ ε-sandwiching approximators if they differ from f with probability at most ε on a random input. Gopalan, Meka and Reingold [12] showed that every DNF formula has a pair of ε-sandwiching approximators of size $(w \log(1/\varepsilon))^{\mathcal{O}(w)}$. Due to their close relation to DNF, the same result holds for CNF formulas.

1.3 Outline of this thesis

The content of this thesis can be split into two parts. Chapter 2, 3 and 4 contain the theoretical background of the Sparsification Lemma, covering some preliminaries, applications and a detailed proof. The remaining chapters contain our results, accompanied by some relevant definitions and theorems from related work. In Chapter 5, conditions are presented under which k-CNF formulas are always satisfiable. In Chapter 6 we present our main result: a sparsification lemma which uses sunflowers. The Sparsification Lemma is then adapted to the parametrised Hitting-Set problem in Chapter 7. Finally, Chapter 8 covers a randomised sparsification lemma which is inspired by the probabilistic approach of DNF sparsification.

2 Preliminaries

In this chapter, we introduce some important concepts from complexity theory and present a list of computational problems that will be relevant throughout this thesis. For more details, we refer to for example [10] and [6].

2.1 An introduction to complexity theory

Complexity theory studies the difficulty of computational problems and classifies them accordingly. A problem is considered difficult if any algorithm that solves it requires a substantial amount of resources, such as time or memory. In this section, we specify how the running time of an algorithm is measured and briefly describe some complexity classes.

To compare the efficiency of different algorithms, we need a standardised way to measure their time complexity. The time it takes to run an algorithm depends on the input and the machine you run it on. To eliminate these dependencies, we define the running time as the number of computational steps performed by the algorithm and express it as a function of the input size. An algorithm might have different running times for different inputs of the same length. We are only interested in the worst-case behaviour for a given input size, since this gives a hard upper-bound for all instances. The precise number of steps required to execute an algorithm might depend on the input and is often hard to capture with a general function for all input sizes n. Therefore, we only consider the asymptotic behaviour of the algorithm as n grows large, which is described using the following asymptotic notation.

Let f be a function from $\mathbb{N}$ to $\mathbb{N}$, then

$$\begin{aligned}
\mathcal{O}(f) &= \{g : \mathbb{N} \to \mathbb{N} \mid \exists c, n_0 \in \mathbb{N}\, \forall n \geq n_0 : g(n) \leq c \cdot f(n)\} \\
\Omega(f) &= \{g : \mathbb{N} \to \mathbb{N} \mid \exists c, n_0 \in \mathbb{N}\, \forall n \geq n_0 : g(n) \geq c \cdot f(n)\} \\
\mathrm{o}(f) &= \{g : \mathbb{N} \to \mathbb{N} \mid \forall \varepsilon > 0 \exists n_0 \in \mathbb{N}\, \forall n \geq n_0 : g(n) \leq \varepsilon \cdot f(n)\} \\
\omega(f) &= \{g : \mathbb{N} \to \mathbb{N} \mid \forall \varepsilon > 0 \exists n_0 \in \mathbb{N}\, \forall n \geq n_0 : g(n) \geq \varepsilon \cdot f(n)\}.
\end{aligned}$$

We usually write $f(n) = \mathcal{O}(g(n))$ rather than $f \in \mathcal{O}(g)$.

In this thesis, we will consider *decision problems*, computational problems which can be answered with 'yes' or 'no'. The complexity class P contains all decision problems that can be solved in polynomial time, i.e. problems for which there exists an algorithm that runs in $\mathcal{O}(n^c)$ time on inputs of size n for some constant $c > 0$. If the order of the polynomial function is unknown or irrelevant, we sometimes write $\mathcal{O}(\mathrm{poly}(n))$.

A problem is in NP if a 'yes'-answer can be verified in polynomial time. This means that, given a certificate for a 'yes'-answer, we can verify in polynomial time that it is correct. For example, consider the problem Subset-Sum, where one is given a set of integers and asked to find a non-empty subset that sums up to zero. A certificate of correctness could be a subset that meets this requirement and we can verify in linear time whether this set indeed has sum zero by adding its elements. Note that P is a subset of NP, because we can verify the solution of a problem in P in polynomial time by simply computing it. It is not known whether P = NP, although it is widely believed that this is not the case.

Suppose we are given two problems, A and B. A *reduction* $\mathtt{R}$ from A to B is an algorithm that transforms every instance of problem A into an equivalent instance of problem B. By equivalent, we mean that x is a 'yes'-instance of problem A if and only if $\mathtt{R}(x)$ is a 'yes'-instance of problem B. We write $A \leq_P B$ if there exists a polynomial-time reduction from A to B. This implies that we can solve problem A in polynomial time if we have a polynomial algorithm for problem B.

A problem A is called NP-*hard* if $L \leq_P A$ for all problems $L \in$ NP. Note that A itself need not be in NP. If A is NP-hard and $A \in$ NP, it is NP-*complete*. If any NP-complete problem is in P, this would imply that P = NP. We discuss several NP-complete problems in the next section.

If a problem depends on some fixed parameter k, then we can describe the running time as a function of both the input size n and k using the notation

$$\mathcal{O}^*(f(k)) = \{g : \mathbb{N} \to \mathbb{N} \mid \exists c \in \mathbb{R}, n_0 \in \mathbb{N}\, \forall n \geq n_0 : g(n) \leq f(k) \cdot n^c\}.$$

An algorithm which runs in $\mathcal{O}^*(f(k))$ for some computable function $f : \mathbb{N} \to \mathbb{N}$ is said to be *fixed-parameter tractable*. FPT is the class of parametrised problems for which there exists such an an algorithm.

2.2 NP-complete problems

The subject of this thesis relates to many well-known NP-complete problems. In this section, we briefly introduce these problems and explain underlying concepts and notation from various areas of mathematics.

A *graph* is an ordered pair $G = (V, E)$ consisting of a set V of vertices and a set E of edges. An *edge* is an unordered pair (u, v) with $u, v \in V$. We refer to the vertex and edge set of graph G as $V(G)$ and $E(G)$ respectively, or as V and E if the intended graph is clear from the context. The cardinality of $V(G)$ and $E(G)$ are often denoted as n and m. We only consider *simple* graphs, i.e. graphs such that there exists at most one edge between each pair of vertices and every edge has two distinct endpoints. A graph is *directed* if the edges are ordered pairs.

Two vertices are *neighbours* if there exists an edge between them. The *neighbourhood* $N(v)$ of a vertex v is the set of its neighbours and its *degree* is defined as $\delta(v) := |N(v)|$. We call vertex v *adjacent* to vertex w if $(v, w) \in E$ and *incident* to edge e if $v \in e$. A set of vertices which are all pair-wise adjacent is called a *clique*. The *(maximum) degree* of a graph G is defined as $\Delta(G) := \max_{v \in V(G)} \delta(v)$.

A *subgraph* of $G = (V, E)$ is a graph $F = (V_F, E_F)$ such that $V_F \subseteq V$ and $E_F \subseteq E$. A subgraph is *induced* by $X \subseteq V$ if $F = (X, \{(v, w) \in E : v, w \in X\})$. We denote this graph by $G[X]$. For the sake of simplicity, we introduce the (abuse of) notation $G - v := G[V \setminus v]$.

A set of vertices is called an *independent set* if none of the vertices are adjacent.

Independent-Set	
Input:	A graph G and $k \in \mathbb{N}$.
Problem:	Does G have an independent set of size k?

If $X \subseteq V$ is independent set, then the set $V \setminus X$ is incident to all edges of the graph, since at least one endpoint of every edge is not in the independent set. A vertex set that is incident to all edges is called a *vertex cover*.

Vertex-Cover	
Input:	A graph G and $k \in \mathbb{N}$.
Problem:	Does G have a vertex cover of size k?

A *propositional formula* is a formula with variables that can either be true or false. The variables are connected with the operators OR (*disjunction*), AND (*conjunction*), and NOT (*negation*), denoted by $\vee, \wedge$ and $\neg$[2]. A variable x or its negation $\neg x$ is referred to as a *literal*. A propositional formula is called *satisfiable* if there exists an assignment of its variables such that the formula evaluates to true.

A propositional formula is said to be in *conjunctive normal form* (*CNF*) if it is a conjunction of distinct *clauses*, where a clause is a disjunction of literals. We denote the number of variables of a CNF formula by n and the number of clauses by m. The number of literals in a clause is called the *width* of a clause. The width of a CNF formula equals the maximum clause width. We say that a clause is *violated* if it evaluates to false and *satisfied* if it evaluates to true. Deciding satisfiability of a CNF formula is one of the most fundamental problems of complexity theory.

k-Cnf-Sat	
Input:	A propositional formula φ in conjunctive normal form of width k.
Problem:	Does there exist a truth assignment that satisfies φ?

Since we don't consider satisfiability of arbitrary formulas, we use the term k-Sat to refer to k-Cnf-Sat.

Let U be any set. We denote the collection of all subsets of U by 2^U. A *set system* on U is a collection $\mathcal{F} \subseteq 2^U$. Sets of size s are referred to as s-sets. A set $X \in U$ *hits* $\mathcal{F}$ if $F \cap X \neq \emptyset$ for all $F \in \mathcal{F}$.

(k, x)-Hitting-Set	
Input:	A universe U and a set system $\mathcal{F} \subseteq 2^U$ with sets of cardinality at most k.
Problem:	Does there exist a set $X \in U$ with $\lvert X \rvert \leq x$ that hits $\mathcal{F}$?

[2]Some definitions allow other operations such as IMPLIES and XOR. However, note that these can be rewritten as combinations of OR, AND, and NOT.

If k and x are clear from the context or irrelevant for our analysis, we simply refer to the problem as Hitting-Set. The parameters $|U|$ and $|\mathcal{F}|$ are denoted by n and m.

Finally, we describe a problem which is not NP-hard, but nevertheless important in the field of complexity theory, as we will see in Section 3.1. Let $\{0,1\}^d$ be the space of binary vectors of length d. Recall that two vectors $a, b \in \{0,1\}^d$ are *orthogonal* if their inner product $\langle a, b\rangle = \sum_{i=1}^{d} a_i b_i$ equals zero.

Orthogonal-Vectors	
Input:	Sets $A, B \subseteq \{0,1\}^d$ such that $\lvert A\rvert = \lvert B\rvert$.
Problem:	Do there exist vectors $a \in A$, $b \in B$ such that $\langle a, b\rangle = 0$?

2.3 Recursion and trees

The algorithms in this thesis make use of *recursion*; from within the algorithm new calls to the algorithm are made. In this section, we introduce some useful tools and terminology to analyse such algorithms.

A *path* is a sequences of unique vertices $v_1, \ldots, v_k$ such that (v_i, v_{i+1}) is an edge for all $i \in \{1, \ldots, k-1\}$. If there exists a unique path between every pair of distinct vertices, we call the graph a *tree*. A tree is called *rooted* if it has one special vertex labelled as the *root*. If a vertex v directly proceeds a vertex w in the unique path from the root to w, we call w a *child* of v and v the *parent* of w. A vertex with no children is called a *leaf*. If every vertex which is not a leaf has exactly two children, the tree is said to be *binary*.

The execution of a recursive algorithm can be represented by a tree where every instance of the algorithm is associated with a vertex. The root corresponds to the initial call of the algorithm and if an instance of the algorithm calls another this is represented by a parent-child relationship. Such a tree is referred to as a *recursion tree*. A path from root to leaf in this tree is called a *recursion path*

To analyse the running time of recursive algorithms, we will use binomial coefficients. The following lemma is a helpful tool to upper bound these.

Lemma 2.1. *Let $p \in (0, \frac{1}{2}]$, $N \in \mathbb{N}$ and $q \leq N$. Then*

$$\sum_{i=0}^{\lfloor Np \rfloor} \binom{N}{i} \leq 2^{Np \log\left(\frac{4}{p}\right)}$$

and

$$\binom{N}{q} \leq 2^{q \log\left(\frac{4N}{q}\right)}.$$

Proof. We prove the first inequality in two steps using the binary entropy function, which is defined as $H(p) = -p \log p - (1-p)\log(1-p)$. First, we show that

$$\sum_{i=0}^{\lfloor Np \rfloor} \binom{N}{i} \leq 2^{NH(p)} \tag{1}$$

and then that

$$H(p) \leq p \log\left(\frac{4}{p}\right). \tag{2}$$

The second inequality follows directly from these results.

Note that $\log p \leq \log(1-p) \leq 0$, because $p \leq \frac{1}{2}$ and log is an increasing function. Then for all $i \leq \lfloor Np \rfloor$, the expression $(\log(1-p) - \log p)(Np - i)$ is non-negative. This can be rewritten as

$$\begin{aligned}
&(\log(1-p) - \log p)(Np - i) \geq 0 && \Leftrightarrow \\
&i(\log p - \log(1-p)) + Np(\log(1-p) - \log p) \geq 0 && \Leftrightarrow \\
&i(\log p - \log(1-p)) + N\log(1-p) - Np\log p - N(1-p)\log(1-p) \geq 0 && \Leftrightarrow \\
&i\log p + (N-i)\log(1-p) + NH(p) \geq 0 && \Leftrightarrow \\
&i\log p + (N-i)\log(1-p) \geq -NH(p),
\end{aligned}$$

so $p^i(1-p)^{N-i} = 2^{i\log p+(N-i)\log(1-p)} \geq 2^{-NH(p)}$. It follows from Newton's binomial theorem that

$$1 = (p+1-p)^N = \sum_{i=0}^{N} \binom{N}{i} p^i(1-p)^{N-i} \geq 2^{-NH(p)} \sum_{i=0}^{\lfloor Np \rfloor} \binom{N}{i},$$

which concludes the proof of (1).

Define $h(p) := H(p) - p\log(4/p) = -(1-p)\log(1-p) - 2p$. Then $h(0) = 0$ and $h'(p) = -(1-p)\cdot 1/(1-p) + \log(1-p) - 2 = -3 + \log(1-p) \leq 0$. This means that $h(p) \leq 0$ for all $p \in (0, 1/2]$, proving (2).

Finally, we derive the second inequality from the first. If $q \leq N/2$, we may choose $p = q/N$ and the result follows. Since $\binom{N}{q} = \binom{N}{n-q}$ and $q\log(4N/q)$ is an increasing function for $q \in \{1, \ldots, N\}$, it also holds for $q > N/2$. □

3 The (Strong) Exponential Time Hypothesis

In this chapter we introduce some complexity hypotheses and their implications for which the Sparsification Lemma is highly useful. A key problem in complexity theory is 3-Cnf-Sat. Under the popular assumption that P $\neq$ NP, it does not have a polynomial-time algorithm. However, this does not exclude the possibility that there exists some subexponential algorithm, for example with running time $\mathcal{O}\left(2^{\sqrt{n}}\right)$. Although the existence of such algorithms has not been proved or refuted, it is conjectured that they do not exist. Since all NP-hard problems can be reduced to 3-Sat, this assumption lower bounds the running time of many other computational problems. To prove these implications, we need the Sparsification Lemma.

Consider for example the standard reduction from 3-Sat to Vertex-Cover, which we will describe in more detail below. It transforms a 3-CNF formula with n variables and m clauses into a graph with $N = 3m + 2n$ vertices. Since m may be cubic in n, a $2^{o(N)}$-time algorithm for Vertex-Cover would allow us to solve 3-Sat in $2^{o\left(n^3\right)}$ time. This does not contradict our assumption that 3-Sat cannot be solved in subexponential time, so we cannot rule out the existence of such an algorithm. However, if we first sparsify the 3-CNF formula, the number of clauses is at most $m = \mathcal{O}(n)$. Then a $2^{o(N)}$-time algorithm for Vertex-Cover cannot exist, as it would solve 3-Sat in $2^{o(n)}$ time.

So far, no subexponential algorithm for k-Sat has been found. The current best bound for $k = 3$ is $\mathcal{O}^*\left(1.3071^n\right)$ and the current fastest algorithm for k-Sat has running time $2^{n(1-\Omega(1/k))}$ [19, 14]. It has been shown that for every $k \geq 3$, k-Sat can be solved in $\mathcal{O}^*\left((2-\varepsilon_k)^n\right)$ time for some $\varepsilon_k > 0$, but we also know that $\varepsilon_k \to 0$. This suggests that following two things are hard:

(1) Finding a subexponential algorithm for 3-Sat.

(2) Finding an algorithm for general Cnf-Sat with running time $\mathcal{O}^*\left((2-\varepsilon)^n\right)$ for some $\varepsilon > 0$.

To express these difficulties, we need a stronger conjecture than P$\neq$NP. Let

$$\delta_k := \inf\{c : \text{there exists an } \mathcal{O}\left(2^{cn}\right) \text{ algorithm for } k\text{-Cnf-Sat}\}.$$

Conjecture 3.1 (Exponential Time Hypothesis, ETH)**.** $\delta_3 > 0$.

Conjecture 3.2 (Strong Exponential Time Hypothesis, SETH)**.** $\lim_{k\to\infty} \delta_k = 1$.

While both conjectures address the complexity of k-Sat, they operate at a different precision level: ETH conjectures that it takes exponential time to solve this problem and SETH fixes the exponent. Assuming ETH, it can be shown that many NP-hard problems do not have a $2^{o(n+m)}$ algorithm using the Sparsification Lemma. Theorem 3.3 illustrates the structure of such a proof with Vertex-Cover.

Theorem 3.3. *Let G be a graph on N vertices with M edges. Assuming ETH, there exists no algorithm that solves* Vertex-Cover *in $2^{o(N+M)}$ time.*

Proof. Let φ be a 3-CNF formula with n variables and m clauses. We can transform φ to an instance of Vertex-Cover with $N = 3m + 2n$ vertices and $M \leq 6m + 2n$ edges using the standard reduction to prove NP-completeness.

For every clause of φ, introduce a clique of size three where the vertices correspond to the literals in the clause. Create a pair of vertices connected by an edge for every variable and its negation. Connect these vertices to every clause vertex corresponding to the same literal. We will show that φ is satisfiable if and only if there exists a vertex cover of size $n + 2m$.

If φ has a satisfying assignment, every clause has at least one true literal. For every clause, choose such a literal and add the other two literals to the vertex cover. For every literal pair, add the literal which evaluates to true. It is easy to check that this gives a valid vertex cover of size $n + 2m$.

Suppose that a vertex cover of size $n + 2m$ exists. To cover every edge between a literal pair, n literal vertices must be included in the vertex cover. Similarly, two vertices of every 3-clique must be in the cover. This requires at exactly $n + 2m$ vertices, so the vertex cover contains no additional vertices. Set every literal whose corresponding literal vertex is in the vertex cover to true. This is a valid truth assignment for φ, because every variable is either true or its negation is true. Since only two vertices of every clause clique are

in the vertex cover, at least one edge per clique is covered by a literal vertex. This means that at least one literal of every clause is true.

Now suppose that there exists an algorithm that solves Vertex-Cover in $2^{o(N+M)}$ time and let $\varepsilon > 0$ be fixed. Apply the Sparsification Lemma with parameter $\varepsilon/2$. This reduces φ to $l \leq 2^{\frac{1}{2}\varepsilon n}$ CNF formulas with at most dn clauses for some constant d depending on ε only. Transform these formulas into instances of Vertex-Cover using the reduction above. The resulting graphs have $N = (3d+2)n$ vertices and $M \leq (6d+2)n$ edges. For any $\gamma > 0$, the given algorithm can solve Vertex-Cover in $2^{\gamma(N+M)} \leq 2^{\gamma(9d+4)n}$ time if n is large enough. This brings the total running time to $2^{\gamma(9d+4)n+\frac{1}{2}\varepsilon n}$, which equals $2^{\varepsilon n}$ if we choose $\gamma = \varepsilon/(2(9d+4))$. Since ε was chosen arbitrarily, we have found a $2^{\varepsilon n}$ algorithm for 3-Sat for every $\varepsilon > 0$, contradicting ETH. □

Using the proof structure of Theorem 3.3, the running time of many other problems in NP can be lower bounded. In general, if there exists a polynomial-time reduction from 3-Sat to a problem A which returns an instance of size $f(n+m)$, then there does not exist a $2^{f^{-1}(n+m)}$-time algorithm for A under ETH.

3.1 Orthogonal-Vectors and SETH

Both ETH and SETH are based on the inherent difficulty of k-Sat. A similar hypothesis has been formulated with respect to Orthogonal-Vectors, a problem which is not NP-hard. It can be brute-forced in $\mathcal{O}\left(dn^2\right)$ time by checking every pair of vectors. The best known algorithm is only slightly more efficient, solving instances of dimension $c \log n$ in $n^{2-\frac{1}{o(\log c)}}$ time, with $c = n^{o\left(\frac{1}{\log \log n}\right)}$ [1]. It is therefore conjectured, that one cannot do significantly better than the naive $\mathcal{O}\left(dn^2\right)$ algorithm.

Conjecture 3.4 (Orthogonal Vector Hypothesis, OVH)**.** *For every $\varepsilon > 0$, there exists a constant C such that instances of* Orthogonal-Vectors *with n vectors of dimension $d = C \log n$ cannot be solved in $\mathcal{O}\left(n^{2-\varepsilon}\right)$ time.*

Note the similarity of this conjecture with SETH, which claims that for every $\varepsilon > 0$ there exists a k such that k-Sat cannot be solved $\mathcal{O}\left(2^{(1-\varepsilon)n}\right)$ time. Both make a statement about the constant in the exponent of the running time, whereas ETH only conjectures that the running time of Sat must be exponential. A reduction by Williams [23] showed that OVH is a direct consequence of SETH. No reductions from Orthogonal-Vectors to Cnf-Sat are known, so OVH may still hold if SETH is refuted.

Theorem 3.5 (SETH implies OVH)**.** *Assuming SETH, there is no $\varepsilon > 0$ such that there exists an $\mathcal{O}\left(N^{2-\varepsilon}\right)$ algorithm which solves* Orthogonal-Vectors *with N vectors of dimension $d = C \log N$ for all $C > 0$.*

Proof. A k-Sat instance $\varphi = \wedge_{i=1}^{m} C_i$ on n variables can be reduced to an m-dimensional instance of Orthogonal-Vectors with sets of cardinality $N = 2^{\frac{n}{2}}$ with the following $\mathcal{O}\left(2^{\frac{n}{2}} m\right)$-time reduction.

Split the variables $x_1, \ldots, x_n$ into two sets $x_1, \ldots, x_{\frac{n}{2}}$, $x_{\frac{n}{2}}, \ldots, x_n$ of equal size, adding a dummy variable if n is odd. Let U be the set of all truth assignments of $x_1, \ldots, x_{\frac{n}{2}}$ and V the set of all truth assignments of $x_{\frac{n}{2}}, \ldots, x_n$. For a clause C and assignment $u \in U \cup V$, define the function $\mathrm{sat}(u, C) := 1$ if and only if C is satisfied under u. If u does not satisfy C, let $\mathrm{sat}(u, C) := 0$.

For all $u \in U$, $v \in V$, define the vectors

$$u' := (1 - \mathrm{sat}(u, C_1), \ldots, 1 - \mathrm{sat}(u, C_m)), \qquad v' := (1 - \mathrm{sat}(v, C_1), \ldots, 1 - \mathrm{sat}(v, C_m))$$

and let $A := \{u' \mid u \in U\}$, $B := \{v' \mid v \in V\}$. Note that A and B are subsets of $\{0,1\}^m$ with cardinality $2^{\frac{n}{2}}$. We claim that Orthogonal-Vector instance (A, B) is equivalent to φ.

For any pair $u' \in A$, $v' \in B$, we have $\langle u', v' \rangle = 0$ if and only if $\mathrm{sat}(u, C_i) = 1$ or $\mathrm{sat}(v, C_i) = 1$ for all $i = 1, \ldots, m$. By definition, this happens if and only if clause C_i is satisfied by u or v respectively. Then u', v' are orthogonal vectors if and only if (u, v) is a satisfying assignment for φ.

Now let $\varepsilon > 0$ be arbitrary and suppose that there exists an algorithm that can solve instances of Orthogonal-Vectors with N vectors of dimension $d = C \log N$ in $\mathcal{O}\left(N^{2-\varepsilon}\right)$ time. If we assume SETH to be true, there exists a $k \in \mathbb{N}$ such that k-Cnf-Sat has no $\mathcal{O}\left(2^{(1-\varepsilon)n}\right)$-time algorithm. Let $\gamma > 0$ be a constant to be determined later and let φ be a k-CNF formula with m clauses and n variables. First, apply

the Sparsification Lemma with parameter γ and then apply the above reduction to every output instance of the Sparsification Lemma. This results in $2^{\gamma n}$ instances of Orthogonal-Vectors with $2^{\frac{n}{2}}$ vectors of dimension Dn.

$$k\text{-Sat}_{n \text{ variables}, m \text{ clauses}} \xrightarrow{\text{Sparsification Lemma}} 2^{\gamma n} \cdot k\text{-Sat}_{n \text{ variables}, Dn \text{ clauses}} \xrightarrow{\text{Reduction 3.5}} 2^{\gamma n} \cdot \text{Orthogonal-Vectors}_{2^{\frac{n}{2}} \text{ vectors in } \{0,1\}^{Dn}}$$

As $Dn = 2D \log N$, we may apply the given algorithm to solve each instance in $\mathcal{O}\left(N^{2-\varepsilon}\right) = \mathcal{O}\left(2^{1-\frac{\varepsilon}{2}}\right)$ time. This means we have solved k-Sat in

$$\mathcal{O}\left(2^{\gamma n} + 2^{\gamma n} \cdot 2^{\frac{n}{2}} Dn + 2^{\gamma n} \cdot 2^{(1-\frac{\varepsilon}{2})n}\right) = \mathcal{O}\left(2^{(1-\frac{\varepsilon}{2}+\gamma)n}\right).$$

If we choose $\gamma < \varepsilon/2$, we now have an algorithm which solves k-Sat in $\mathcal{O}\left(2^{(1-\delta)n}\right)$ time for $\delta = \varepsilon/2 - \gamma > 0$, contradicting SETH. □

Orthogonal-Vectors is a special case of a more general problem.

k-Orthogonal-Vectors

Input: Sets $A_1, A_2, \ldots, A_k \subseteq \{0,1\}^d$ such that $|A_1| = |A_2| = \cdots = |A_k|$.

Problem: Do there exist k vectors $a_1 \in A_1, \ldots, a_k \in A_k$ such that $\langle a_i, a_j \rangle = 0$ for all $1 \leq i < j \leq k$?

By partitioning the variables into k parts, the proof of Theorem 3.5 can be generalised to show that no $\mathcal{O}\left(N^{k-\varepsilon}\right)$-time algorithm exists for k-Orthogonal-Vectors if we assume SETH to be true.

By applying the reduction from Theorem 3.5, we can use algorithms for Orthogonal-Vectors to solve instances of k-Sat. An interesting question is, whether a small improvement of the current best algorithm for Orthogonal-Vectors or an improvement of the sparsification constant could lead to a an algorithm for k-Sat which is faster than the current best known $2^{n(1-\Omega(1/k))}$. The former turns out to be true. If we improve the exponent of the fastest Orthogonal-Vectors algorithm to $2 - \omega(\log\log c/\log c)$, then k-Sat can be solved in $2^{n(1-\omega(1/k))}$ time. However, an improvement of the Sparsification Lemma alone does not immediately result in a faster algorithm.

Suppose that the sparsification constant can be improved to $d(\varepsilon, k) = (1/\varepsilon)^{\mathcal{O}(k)}$. In combination with the fastest algorithm for Orthogonal-Vectors, this gives us a $2^{\left(1+\varepsilon-\omega\left(\frac{1}{k \log 1/\varepsilon}\right)\right)n}$-time algorithm for k-Sat, which we want to be at most $2^{(1-\omega(1/k))n}$. To make sure that ε does not dominate the exponent, we have to choose it such that $\varepsilon = \mathcal{O}(1/k)$. However, that also implies that $\omega\left(\frac{1}{k \log 1/\varepsilon}\right)$ equals at most $\omega(1/(k \log k))$, instead of the $\omega(1/k)$ we are looking for.

Possibly, a combination of a better Orthogonal-Vectors algorithm and an improved sparsification constant could lead to a faster algorithm for k-Sat. A faster algorithm for Orthogonal-Vectors would have running time $n^{2-\omega(f(c))}$ for some function f which is asymptotically larger than $\omega(1/\log c)$. For the right choice of f and ε, this could be enough to cancel out the exponent ε from the Sparsification Lemma in the analysis above. An interesting question, which we will not attempt to answer in this thesis, is how large f must be to achieve this.

4 The Sparsification Lemma

In the k-CNF-SAT problem, there is an inherent disbalance between the parameters n and m. The number of clauses can be as large as n^k, whereas the number of variables is always $\mathcal{O}(m)$, since every clause contains at most k different variables. A CNF formula is called *sparse* if the number of clauses is linear in the number of variables. In this chapter, we study a theorem by Impagliazzo and Paturi known as the Sparsification Lemma, which converts an instance of k-CNF-SAT into a disjunction of sparse CNF formulas. This lemma is a useful tool to lower bound the running time of many NP-hard problems, as we have seen in Chapter 3. Before we state and prove the Sparsification Lemma, we prove a similar result for the problem VERTEX-COVER.

4.1 Warm-up: vertex cover

The number of edges in a graph can be quadratic in the number of vertices. This means that for graph problems, the impossibility of a $2^{o(n)}$-time algorithm does not directly imply the impossibility of a $2^{o(m)}$ algorithm. However, for VERTEX-COVER these running times turn out to be equivalent.

Suppose we want to find a vertex cover of size k on a graph G and let v be one of its vertices. To cover all edges incident to v, either v itself or all of its neighbours must be in the vertex cover. If we choose v, we can remove v and search for a vertex cover of size $k-1$ in the remaining graph. If we put its neighbours in the cover, we may remove v and its neighbours and search for a vertex cover of size $k-d(v)$. To simplify the resulting instance as much as possible, it makes sense to consider a vertex of high degree.

This observation lies at the heart of the proof of Theorem 4.1. We repeatedly choose a vertex of high degree, put either this vertex or its neighbours in the vertex cover, and remove vertices accordingly. After a number of iterations, this gives an instance where every vertex has low degree, which can be solved efficiently with a $2^{o(m)}$-time algorithm. Many vertices are discarded when all neighbours of a vertex are put in the cover, so this can only be done a few times. This greatly limits the number of possible recursion paths we can take.

Theorem 4.1. VERTEX-COVER *can be solved in* $2^{\varepsilon n}$ *time for all* $\varepsilon > 0$ *if and only if it can be solved in* $2^{\varepsilon m}$ *time for all* $\varepsilon > 0$.

Proof. Assume we can solve any instance of VERTEX-COVER in $2^{\varepsilon n}$ time for any $\varepsilon > 0$. Note that at most $2m$ vertices can have nonzero degree, so we may assume without loss of generality that $n \leq 2m$. Then the assumed algorithm solves VERTEX-COVER in $2^{\varepsilon' m}$ time for all $\varepsilon' = 2\varepsilon > 0$.

Conversely, assume that for every $\varepsilon > 0$ there exists an algorithm for VERTEX-COVER with time complexity $2^{\varepsilon m}$. Let $\varepsilon > 0$ be given and let (G, k) be an instance of VERTEX-COVER. We will reduce (G, k) to a set $\mathcal{F} = \{(G_i, k_i)\}$ of smaller VERTEX-COVER instances such that (G, k) is true if and only if some $(G_i, k_i) \in \mathcal{F}$ is true.

First note that it suffices to show that the algorithm runs in $2^{\varepsilon_0 n}$ time for some $\varepsilon_0 < \varepsilon$. We may therefore assume ε to be small enough such that $\log(1/\varepsilon) > 0$. Let $d = 12\log(1/\varepsilon)/\varepsilon$, which is positive by our assumption, and choose a vertex v with degree larger than d. A valid vertex cover of G must contain either v itself or all of its neighbours. Computing a vertex cover of size k in G is therefore equivalent to finding either a vertex cover of size $k-1$ in $G-v$ or a vertex cover of size $k-d$ in $G-(v \cup N(v))$. This results in two new cases of VERTEX-COVER which we can treat the same way. Algorithm 1 applies this procedure recursively until no vertices of degree larger than d remain and returns the resulting instances of VERTEX-COVER.

Algorithm 1: `sparsify`

Input : A VERTEX-COVER instance (G, k)
Output: A family $\mathcal{F} = \{(G_i, k_i)\}$ of instances of VERTEX-COVER such that (G, k) is solvable if and only if (G_i, k_i) is for some i

```
1 sparsify(G, k)
2 |  if ∃v ∈ V(G) such that d(v) > d then
3 |  |  return sparsify(G − v, k − 1) ∪ sparsify(G − (v ∪ N(v)), k − d)
4 |  return {(G, k)}
```

Consider an arbitrary recursion path of the algorithm. Since the original graph has n vertices and each recursive call decreases the number of vertices by at least one, it has depth at most n. If a vertex v is not included in the vertex cover, at least $d+1$ vertices are discarded, so the algorithm does not exclude more than $\lfloor n/d \rfloor$ vertices in the same path. This means that there can be at most $\sum_{i=0}^{\lfloor n/d \rfloor} \binom{n}{i}$ recursion paths. If we assume ε to be small enough such that $d \geq 2$, it follows from Lemma 2.1 that $|\mathcal{F}| \leq 2^{\frac{n}{d}\log(4d)}$. A graph returned at the end of a recursion path has at most nd edges, because if it had a vertex of degree more than d, the algorithm would branch on this vertex rather than return the instance. This means we can solve every instance in $\mathcal{F}$ in $2^{\hat{\varepsilon}nd}$ time for any $\hat{\varepsilon} > 0$ using the given algorithm. The total running time therefore equals $2^{\frac{n}{d}\log(4d)} \cdot 2^{\hat{\varepsilon}nd}$. We will show that this is at most $2^{\varepsilon n}$ for the right choice of $\hat{\varepsilon}$.

First, consider only the term $2^{\frac{n}{d}\log(4d)}$. We will upper bound this by $2^{\frac{1}{2}\varepsilon n}$. Assume ε to be small enough such that $\log d \geq 1$, then

$$\log(4d) = 2 + \log d \leq 3 \log d.$$

This means that the desired upper bound $\log(4d)/d \leq \varepsilon/2$ is achieved when

$$\begin{aligned}
\frac{3}{d}\log d &\leq \frac{1}{2}\varepsilon && \Leftrightarrow \\
\frac{3\varepsilon}{12\log\frac{1}{\varepsilon}}\log\left(\frac{12\log\frac{1}{\varepsilon}}{\varepsilon}\right) &\leq \frac{1}{2}\varepsilon && \Leftrightarrow \\
\log\left(\frac{12\log\frac{1}{\varepsilon}}{\varepsilon}\right) &\leq 2\log\frac{1}{\varepsilon} && \Leftrightarrow \\
\left(\log\left(12\log\frac{1}{\varepsilon}\right) + \log\frac{1}{\varepsilon}\right) &\leq 2\log\frac{1}{\varepsilon}. &&
\end{aligned}$$

If we assume ε to be small enough such that $\log\left(12\log(1/\varepsilon)\right) \leq \log 1/\varepsilon$, the above inequality holds and we have at most $2^{\frac{1}{2}\varepsilon n}$ recursion paths. Now set $\hat{\varepsilon} := \varepsilon/(2d)$. This brings the total running time to at most $2^{\frac{1}{2}\varepsilon n} \cdot 2^{\frac{\varepsilon}{2d}nd} = 2^{\varepsilon n}$. □

4.2 Statement and proof

A theorem to sparsify k-SAT formulas was first introduced in [16]. In this work, Impagliazzo, Paturi and Zane showed that any k-CNF formula on n variables can be reduced to a number of smaller formulas, each of which consists of at most Cn clauses. The constant C depended on k and a parameter $\varepsilon > 0$ and was doubly exponential in these values. In a later paper [4], the analysis was improved, lowering C to an exponential function in k and ε.

The Sparsification Lemma does not work directly on CNF formulas. Instead, it is formulated in terms of another NP-hard problem, HITTING-SET. A k-CNF formula φ can be reduced in polynomial time to an instance of k-HITTING-SET on a universe of size $2n$, where every element corresponds to a literal. Every clause is represented by a set of its literals and a pair $\{x_i, \neg x_i\}$ is added for every variable. A hitting set of size n then corresponds to a satisfying assignment of f. Conversely, we can translate a set system $\mathcal{F}$ whose $2n$ elements are partitioned into n pairs to a CNF formula on n variables by associating every element with a unique literal and making a clause for every set. A hitting set of size n then corresponds to a satisfying assignment. For the sake of simplicity, we avoid doing these reductions in the Sparsification Lemma. Instead, we directly view SAT as a special case of HITTING-SET where every set is a set of literals. This means we do not need to add the sets $\{x_i, \neg x_i\}$ to the set system.

The statement and proof of the Sparsification Lemma require some preliminary definitions.

Definition 4.2. *An s-flower is a collection of s-sets $S_1, \ldots, S_z$ such that the heart $H := \cap_{i=1}^{z} S_i$ is nonempty. We call $S_1 \setminus H, \ldots, S_z \setminus H$ petals and $|S_i \setminus H|$ the petal size.*

Definition 4.3. *Let $\mathcal{F}$ be a set system. We call $\mathcal{G}$ a restriction of $\mathcal{F}$ if for each $F \in \mathcal{F}$ there exists a $G \in \mathcal{G}$ with $G \subseteq F$.*

Finding a vertex cover in a graph is equivalent to finding a hitting set in a set system with sets of size two. The idea behind the Sparsification Lemma is therefore similar to Theorem 4.1. To sparsify an instance of Vertex-Cover, we branched on vertices of high degree, putting either the vertex itself or its neighbours in the vertex cover. In a set system, we consider a set to have high degree if it is contained in many larger sets, i.e. when it forms the heart of a large flower. A hitting set that hits such a flower must either hit its heart or all of the petals. This again gives us two choices to branch on.

Since we now have to consider flowers of different sizes, we have to specify in which order they are processed. Finding a hitting set is easier for sets of smaller size. If a set consists of one element, it is clear that this element should be in the hitting set, but for large sets this choice far from trivial. By replacing large flowers with their heart or petals — sets of smaller size — we therefore simplify the set system, fixing an element of the hitting set whenever a set of size one is added. To ensure this happens as often as possible, we branch on flowers of small sets first and prefer flowers with small petals when there are multiple flowers with equal set size.

By focusing on small sets first, we also prevent unnecessary branching. Suppose an element x is contained in the petals of many flowers. After branching on all these flowers, x has high degree. This may seem like a good thing, as putting x in the hitting set now considerably simplifies the system. However, it would have been more efficient to put x in the hitting set immediately by branching on the flower formed by these petals.

For Vertex-Cover, it was easy to see that the sparsification algorithm terminated after not too many steps. Every branch on a 2-flower adds at least one singleton, so at most n branches can take place. However, the situation becomes more complicated for a system which also contains 3-sets, as we may have a flower with petals of size two. In that case, the petals of the flower need no longer be disjoint and many new 2-sets are added to the system. Here, the chosen branching order becomes critical.

We only branch on a 3-flower if no 2-flowers exist. This means that not too many of its petals may overlap, otherwise they would form a 3-flower with smaller petals when combined with the heart. The algorithm would have branched on such a flower first. Since there may not be too much overlap, the number of petals, and therefore the number of added 2-sets, is bounded.

For $k > 3$, this observation becomes even more crucial. In this setting, not every branching step adds a singleton to the system, so we cannot measure our progress directly by looking at the fixed elements of the hitting set. However, with every branching step smaller sets are added to the system. If the number of added sets is bounded, we can therefore branch only a limited number of times.

Lemma 4.4 (Sparsification Lemma)**.** *Let $\varepsilon > 0$, $k \in \mathbb{N}$ be given and let U be a set of n elements. There exists an algorithm that, given a set system $\mathcal{F} \subseteq 2^U$ of sets of cardinality at most k, returns set systems $\mathcal{F}_1, \ldots, \mathcal{F}_l \subseteq 2^U$ with sets of size at most k such that the following holds:*

(1) A set $X \subseteq U$ hits $\mathcal{F}$ if and only if it hits $\mathcal{F}_i$ for some $i \in 1, \ldots, l$.

(2) $\mathcal{F}_1, \ldots, \mathcal{F}_l$ are restrictions of $\mathcal{F}$.

(3) For every $i \in 1, \ldots, l$, every element of U is in at most $d := \left(\frac{4k^3 \log \frac{1}{\varepsilon}}{\varepsilon}\right)^{k-1}$ sets of $\mathcal{F}_i$.

(4) $l \leq 2^{\varepsilon n}$.

(5) The running time is $\mathcal{O}^*(l)$.

Proof. Let $\theta_0, \ldots, \theta_{k-1}$ be constants to be defined later. We call a flower $S_1, \ldots, S_z$ with petal size p *good* if $z \geq \theta_p$. A set system $\mathcal{F}$ is *inclusion-wise minimal* if no set is contained entirely in another. We let $\pi(\mathcal{F})$ denote the set of inclusion-wise minimal sets of $\mathcal{F}$, where for every pair $F, G \in \mathcal{F}$ such that $F \subseteq G$, the set G is discarded.

The algorithm that satisfies Lemma 4.4 is outlined in Algorithm 2. If a set is hit, then so are all of its supersets. We may therefore replace the initial input $\mathcal{F}$ by $\pi(\mathcal{F})$ to ensure that it is inclusion-wise minimal. In all subsequent calls of `reduce`, this is enforced by the algorithm.

Lemma 4.5. *The collection $\mathcal{F}_1, \ldots, \mathcal{F}_l$ of set systems returned by Algorithm 2 satisfies Item (1) and (2).*

Algorithm 2: `reduce`

Input : A set system $\mathcal{F}$ with sets of size at most k
Output: A collection of set systems $\mathcal{F}_1, \dots, \mathcal{F}_l$ as described in Lemma 4.4

```
reduce(F)
  for s = 2, ..., k do
    for p = 1, ..., s − 1 do
      if there exists a good s-flower S_1, ..., S_z with petal size p then
        H = ∩_{i=1}^{z} S_i; F_heart = π(F ∪ {H}); F_petals = π(F ∪ {S_i \ H : i = 1, ..., z})
        return reduce(F_heart) ∪ reduce(F_petals)
  return {F}
```

Proof. Consider an arbitrary output $\mathcal{F}_i$ and the recursion path of Algorithm 2 leading to this output. Note that the algorithm only removes sets by means of the π-operator, so a set $F \in \mathcal{F}$ is only removed if a subset of F was added to the system. This means that there always exists an $F' \in \mathcal{F}_i$ such that $F' \subseteq F$, so $\mathcal{F}_i$ is a restriction of $\mathcal{F}$.

Observe that a hitting set X of $\mathcal{F}$ must hit the heart or all petals of a flower $S_1, \dots, S_z$, otherwise there exists an S_i such that $X \cap S_i = \emptyset$. Therefore, a hitting set of $\mathcal{F}$ also hits $\mathcal{F}_{heart}$ or $\mathcal{F}_{petals}$. Conversely, since $\mathcal{F}_{heart}$ and $\mathcal{F}_{petals}$ are restrictions of $\mathcal{F}$, it is clear that a hitting set of $\mathcal{F}_{heart}$ or $\mathcal{F}_{petals}$ is also a hitting set of $\mathcal{F}$. This holds for every recursive call of the algorithm, so it follows that X hits $\mathcal{F}$ if and only if it hits one of $\mathcal{F}_1, \dots, \mathcal{F}_l$. □

The algorithm outputs set systems that contain no good s-flowers for $s = 2, \dots, k$. Note that if an h-set H is contained in θ_{j-h} j-sets, these sets form a good j-flower with heart H and petal size $j - h$. This leads to the following observation.

Observation 4.6. *If $\mathcal{F}$ has no good j-flower, every h-set is contained in at most θ_{j-h} sets of size j.*

In particular, Observation 4.6 implies that every element is contained in at most $\theta_{i-1} - 1$ sets of size i at the end of the algorithm, and therefore in at most $\sum_{j=1}^{i}(\theta_{j-1} - 1) \leq i\theta_{i-1}$ sets of size at most i. This means that Item (3) is satisfied if we choose θ_{k-1} such that $k\theta_{k-1} \leq d$.

In order to upper bound the number of output instances, we need to define a measure of progress for the reduction algorithm. We do this by splitting every recursion path of the algorithm into $k - 1$ phases numbered $2, \dots, k$. Phase i starts when the algorithm branches on a j-flower for the first time with $j \geq i$ and ends when the first j'-flower branch takes place for some $j' > i$. Note that this definition may lead to 'empty' phases where beginning and end coincide. For example, if Algorithm 2 branches on a 5-flower after branching on a 3-flower for the first time, this marks the start and end of phase 4 and the beginning of phase 5.

Let the *i-degree* of a set X denote the number of i-sets that contain X. The following invariant holds for every phase of the algorithm.

Lemma 4.7. *After the start of phase i, every h-set has j-degree at most $2\theta_{j-h}$ for all $j \leq i$.*

Proof. Consider the state of the algorithm directly after the beginning of phase i. If phase i is empty, i.e. does not start with an i-flower, this means that no good i-flowers exist. Then Observation 4.6 ensures that every h-set has i-degree at most $\theta_{i-h} - 1 \leq 2\theta_{i-h}$ for all $h < i$. We may therefore assume that phase i is not empty.

Let the i-flower which marks the start of phase i have petal size p and heart H. Observe that at most θ_{p-h} of the petals can contain the same h-set H', otherwise they would form a good i-flower with heart $H \cup H'$ and petal size $p - h$. This flower would have been preferred by the algorithm, because of its smaller petal size. Let $\mathcal{F}_{heart}$ and $\mathcal{F}_{petals}$ be the two set systems returned by branching on this flower. Since we are branching on an i-flower, no j-flowers exist for $j < i$, hence every h-set has j-degree at most $\theta_{j-h} - 1$. Only one set is added to $\mathcal{F}_{heart}$, raising the j-degree of every h-set to at most $\theta_{j-h} < 2\theta_{j-h}$ for all $j < i$. This means that our claim still holds for $\mathcal{F}_{heart}$.

Now consider $\mathcal{F}_{petals}$, where every set of the flower is replaced by the corresponding petal of size p. It follows from the observation above, that the p-degree of every h-set increases by at most θ_{p-h}. Since $p < i$, the p-degree of any h-set was at most θ_{p-h} before this branching step. Therefore, every h-set is now contained in at most $2\theta_{p-h}$ p-sets of $\mathcal{F}_{petals}$.

If an h-set now has j-degree at least θ_{j-h} for some $j < i$ after branching, it forms the heart of a good j-flower. The algorithm will branch on this flower first before considering sets of size at least i again. We then arrive once again in a state without good j-flowers for $j < i$. □

A direct consequence of Lemma 4.7 is that after the start of phase i, no more than $2\theta_{i-h}$ sets of size i can be eliminated by a single h-set.

Define $\beta_j = (4\alpha k)^{j-1}$ for some α to be determined later and let $\theta_0 = 1$, $\theta_j = \alpha\beta_j$. Then the number of sets added can be upper bounded in the following way.

Lemma 4.8. *Along every recursion path of Algorithm 2, at most $n\beta_i$ sets of size at most i are added.*

Proof. Let s_i denote the number of sets of size at most i that are added by the algorithm. We first look at the number of sets of size exactly i.

Every added i-set X either remains in the set system until the end of the algorithm or is deleted when a subset of X is added to the system. Note that a deletion does indeed require a new set to be added to the system, because X is a petal or heart of a flower, so it is a subset of some set $Y \in F$. If X was eliminated by a subset which was already present in $\mathcal{F}$, this set was also a subset of Y, contradicting inclusion-wise minimality of $\mathcal{F}$.

Sets of size i are only added after phase i has begun, so it follows from Lemma 4.7 that at most $2\theta_{i-h}$ added sets of size i can be deleted by a single h-set. Summing over all h, we find that the number of added i-sets that are subsequently deleted equals

$$\sum_{h=1}^{i-1} \#h\text{-sets added} \cdot 2\theta_{i-h}. \tag{3}$$

At the end of the algorithm, every element is contained in at most θ_{i-1} i-sets, so at most $n\theta_{i-1}$ i-sets remain. Together with (3), this upper bounds the number of added i-sets by

$$\#i\text{-sets added} \leq n\theta_{i-1} + \sum_{h=1}^{i-1} \#h\text{-sets added} \cdot 2\theta_{i-h} = n\alpha\beta_{i-1} + 2\alpha \sum_{h=1}^{i-1} \#h\text{-sets added} \cdot \beta_{i-h}. \tag{4}$$

On top of these i-sets, we need to consider the number of j-sets added for $j < i$. Combined with (4), this gives the following relation for s_i.

$$s_i \leq n\alpha\beta_{i-1} + 2\alpha \sum_{h=1}^{i-1} \#h\text{-sets added} \cdot \beta_{i-h} + s_{i-1} \leq n\alpha\beta_{i-1} + 2\alpha \sum_{h=1}^{i-1} s_h\beta_{i-h} + s_{i-1}. \tag{5}$$

We prove by induction on i that (5) satisfies our claim.

The number of singletons added along any recursion path equals at most $n = \beta_1 n$. Suppose that for $j = 1, \ldots, i$ Algorithm 2 adds as most $n\beta_j$ sets of size at most j. Then for $j = i + 1$ we get

$$\begin{array}{rcllcl} s_{i+1} & \leq & n\alpha\beta_i + 2\alpha \sum_{h=1}^{i} s_h\beta_{i+1-h} + s_i & \leq & n\alpha\beta_i + 2\alpha \sum_{h=1}^{i} n\beta_h\beta_{i+1-h} + n\beta_i \\ & = & n\alpha\beta_i + 2\alpha n \sum_{h=1}^{i} \beta_i + n\beta_i & = & n\beta_i\,(\alpha + 2\alpha i + 1) \\ & \leq & n\beta_i\,(4\alpha k) & = & n\beta_{i+1}. \end{array}$$

□

Lemma 4.8 implies that the recursion depth of Algorithm 2 is at most $n\beta_{k-1}$, since at least one set of size at most $k - 1$ is added with every branching step. During a petal branch with petal size i, at least θ_i

sets of size i are added. In total, no more than $n\beta_i$ i-sets are added along any recursion path, so the number of petal branches in a single recursion path equals at most

$$\sum_{i=1}^{k-1} \frac{n\beta_i}{\theta_i} = \sum_{i=1}^{k-1} \frac{n}{\alpha} < \frac{kn}{\alpha}.$$

This means that the algorithm can follow at most

$$\binom{n\beta_{k-1}}{\frac{nk}{\alpha}} \tag{6}$$

different recursion paths. Lemma 2.1 implies that (6) equals at most $2^{\lambda n}$ with

$$\begin{aligned}\lambda &\leq \frac{k}{\alpha} \log\left(\frac{4\alpha\beta_{k-1}}{k}\right) \\ &\leq \frac{k}{\alpha} \log\left(\beta_k\right) \\ &= \frac{k^2}{\alpha} \log\left(4\alpha k\right).\end{aligned}$$

Let $\alpha \approx k^2 \log(1/\varepsilon)/\varepsilon$, then $\lambda \approx \varepsilon$ and we may pick d to be

$$k\theta_{k-1} = k\left(4\alpha k\right)^{k-2} \leq \left(4\alpha k\right)^{k-1} \approx \left(\frac{4k^3 \log \frac{1}{\varepsilon}}{\varepsilon}\right)^{k-1}.$$

Finally, we analyse the running time. Let t denote the total number of times that Algorithm 2 is called. Since the algorithm outputs l set systems, there are l instances of the algorithm that halt and $t - l$ which call two new instances of the algorithm. Every call to Algorithm 2, apart from the initial one, is generated by these latter $t - l$ instances, so we know that $t - 1 \leq 2(t - l)$. This simplifies to $t \leq 2l - 1$, which means that Algorithm 2 is called $\mathcal{O}(l)$ times. Since the algorithm itself takes polynomial time, the total running time is $\mathcal{O}^*(l)$, satisfying Item (5). □

As noted before, we view k-Sat as a special case of Hitting-Set, where every literal corresponds to an element of the universe. A sparsification lemma for k-Sat then follows directly from Lemma 4.4.

Corollary 4.8.1. *For all $\varepsilon > 0$ and $k \in \mathbb{N}$, there exists an algorithm which rewrites a k-Sat formula φ as a disjunction $\varphi = \varphi_1 \vee \cdots \vee \varphi_l$ of at most $2^{\varepsilon n}$ k-Sat formulas such that every variable appears in at most $d(\varepsilon, k) := \left(\frac{ck^3 \log \frac{1}{\varepsilon}}{\varepsilon}\right)^{k-1}$ clauses of φ_i for some constant c. This algorithm runs in $\mathcal{O}^*(l)$ time.*

Using Corollary 4.8.1, we can prove an analogue of Theorem 4.1 for k-Cnf-Sat.

Corollary 4.8.2. *k-Cnf-Sat can be solved in $\mathcal{O}(2^{\varepsilon n})$ time for every $\varepsilon > 0$ if and only if it can be solved in $\mathcal{O}(2^{\varepsilon m})$ time for every $\varepsilon > 0$.*

Proof. Every clause contains at most k different variables, so $n = \mathcal{O}(m)$. This immediately implies the '$\Rightarrow$' direction.

Suppose that for every $\varepsilon > 0$, there exists an algorithm $\mathtt{A}_\varepsilon$ with running time $2^{\varepsilon m}$. Let $\gamma > 0$ and choose λ such that $2\lambda d \leq \gamma$, where d denotes the sparsification constant $d(\gamma/2, k)$. Now consider the following algorithm: apply Theorem 4.8.1 with parameter $\gamma/2$ to the given CNF formula and use $\mathtt{A}_\gamma$ to solve every resulting sparse instance. Theorem 4.8.1 outputs at most $2^{\frac{1}{2}\gamma n}$ CNF formulas and solving them takes $2^{\lambda n d}$ time. For large enough n, sparsification takes $f(n)2^{\frac{1}{2}\gamma n}$ time for some polynomial function f. The total running time then equals at most

$$f(n)2^{\frac{1}{2}\gamma} + 2^{\frac{1}{2}\gamma n} \cdot 2^{\lambda n d} \leq 2f(n)2^{\frac{1}{2}\gamma n} \cdot 2^{\frac{1}{2}\gamma n} = 2f(n)2^{\gamma n}.$$

For every $\varepsilon > 0$, there exists an n_0 such that $2f(n) \leq 2^{\varepsilon n}$ for all $n \geq n_0$. This means that for arbitrary $\varepsilon > 0$, the above algorithm runs in $2^{(\varepsilon+\gamma)n}$ time for large enough values of n. Since γ was also chosen arbitrarily, the statement follows. □

In later chapters, we often assume the running time of the Sparsification Lemma to be $2^{\varepsilon n}$ for simplicity. Note that when we do this, we implicitly use the same technique as in Corollary 7 to upper bound the $\mathcal{O}^*\left(2^{\varepsilon n}\right)$ running time by an expression of the form $2^{\varepsilon' n}$ for some $\varepsilon' > \varepsilon$.

5 Lovász Local Lemma for sparse CNF formulas

A useful tool in combinatorics and computer science is the probabilistic method. Although Erdős [7] is often seen as a pioneer of this technique, it was first used by Szele in [22]. It uses a probabilistic argument to prove the existence of a combinatorial object with certain properties. Proofs of this type usually have the following structure.

Suppose we have a collection of objects and we want to prove the existence of one with some desired property. We can set up a probability space supported on the collection. If an object sampled according to this probability space has the desired property with positive probability, then there must exist one such object.

To illustrate the concept, Theorem 5.1 shows a simple application to k-Sat.

Theorem 5.1. *Let φ be an instance of k-CNF-SAT such that every clause is a disjunction of exactly k literals. If $m < 2^k$, then φ is satisfiable.*

Proof. Sample a truth assignment A by setting every variable to true with probability $1/2$. Recall that we call a clause violated if it evaluates to false under A. For every clause C, it holds that $\mathbb{P}(C \text{ violated}) = 2^{-k}$. Then the expected number of violated clauses equals

$$\mathbb{E}[\#\text{clauses violated under } A] = m2^{-k} < 1$$

by linearity of expectation. This means that there exists a truth assignment such that no clauses are violated. □

In this chapter, we use the Lovász Local Lemma, a powerful result of the probabilistic method, to show satisfiability of sparse k-SAT formulas.

5.1 Lovász Local Lemma

Many computational problems can be reformulated in terms of undesirable events. For example, for a CNF-SAT instance φ we could define 'bad' events $A_1, A_2, \ldots, A_m$ as clause $1, 2, \ldots, m$ being violated. Then φ is satisfiable if and only if there exists a truth assignment such that none of these events occur. If all events occur with low probability and are mostly independent, one would expect the probability that none of them take place to be positive. Lovász and Erdös formalised this intuition in [8]. To formulate the Lovász Local Lemma (LLL) succinctly, we need the concept of a dependency graph.

Definition 5.2. *Let $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ be a collection of events. The dependency graph $\Gamma(\mathcal{A})$ is a directed graph on vertex set $\{1, 2, \ldots, n\}$ such that A_i is mutually independent of $\{A_j : (i,j) \notin E(\Gamma(\mathcal{A}))\}$.*

Recall that an event A is *mutually independent* of a set of events $\mathcal{B}$ if A is independent of any subset of $\mathcal{B}$. We denote the complement of an event A by $\overline{A}$.

Theorem 5.3 (Asymmetric LLL)**.** *Let $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ be events with dependency graph $\Gamma(\mathcal{A})$. If there exist $x_1, x_2, \ldots, x_n \in [0,1)$ such that $\mathbb{P}(A_i) \leq x_i \prod_{j:(i,j)\in E(\Gamma)}(1 - x_j)$ for all i, then*

$$\mathbb{P}\left(\overline{A}_1 \cap \cdots \cap \overline{A}_n\right) \geq \prod_{i=1}^{n}(1 - x_i) > 0.$$

The proof of Theorem 5.3 can be found in [8] and is beyond the scope of this thesis.

If all events are bounded by the same probability, Theorem 5.3 can be simplified to a result known as the Symmetric Lovász Local Lemma. This version of LLL first appeared in [21]. The proof uses the inequality

$$\left(1 - \frac{1}{d+1}\right)^d \geq \frac{1}{e}, \tag{7}$$

which we will use multiple times throughout this chapter. This inequality follows from the fact that $\left(1 - \frac{1}{d+1}\right)^d$ is decreasing and $\lim_{d\to\infty}\left(1 - \frac{1}{d+1}\right)^d = e^{-1}$.

Theorem 5.4 (Symmetric LLL)**.** *Let $A_1, A_2, \ldots, A_n$ be events that each occur with probability at most $p < 1$. If every event is mutually independent of all but d others and $ep(d+1) \leq 1$, then $\mathbb{P}\left(\overline{A}_1 \cap \cdots \cap \overline{A}_n\right) > 0$.*

Proof. Let $x_i = 1/(d+1)$ for all x_i in the Asymmetric LLL. Then for $i = 1, \ldots, n$,

$$x_i \prod_{j:(i,j)\in E(\Gamma)} (1 - x_j) = \frac{1}{d+1} \prod_{j:(i,j)\in E(\Gamma)} \left(1 - \frac{1}{d+1}\right) \geq \frac{1}{d+1}\left(1 - \frac{1}{d+1}\right)^d \geq \frac{1}{e(d+1)}.$$

The last inequality follows from (7). If $ep(d+1) \leq 1$, then for every event A_i we have

$$\mathbb{P}(A_i) \leq p \leq \frac{1}{e(d+1)} \leq x_i \prod_{j:(i,j)\in E(\Gamma)} (1 - x_j),$$

so $\mathbb{P}\left(\overline{A}_1 \cap \cdots \cap \overline{A}_n\right) > 0$. □

5.2 Applications to Cnf-Sat

The Sparsification Lemma reduces a k-Sat formula to a disjunction of sparse k-Sat formulas. Using Lovász Local Lemma, we show that if a formula is sufficiently sparse, it is always satisfiable. An algorithmic proof of the Lovász Local Lemma was published by Moser and Tardos in [18], so if we can prove using LLL that a formula has a satisfying assignment, then this assignment can also be computed in polynomial time.

If all clauses of a formula have the same width, satisfiability can be proved with the Symmetric Lovász Local Lemma.

Theorem 5.5. *Let φ be an instance of k-Cnf-Sat such that every clause has exactly k literals. If every variable appears in at most $\frac{2^k}{ke} - \frac{1}{k} + 1$ clauses, φ is satisfiable.*

Proof. Let $A_1, A_2, \ldots, A_m$ denote the events that clause $C_1, C_2, \ldots, C_m$ of φ is not satisfied. If every variable appears in at most f clauses, then every clause shares at least one variable with at most $(f-1)k$ other clauses. Two clauses that do not share variables are mutually independent, so the dependency graph has maximum degree at most $(f-1)k$. If we uniformly assign true or false to every variable, $\mathbb{P}(\overline{A}_i) = 2^{-k}$ for all $i \in \{1, 2, \ldots, m\}$. Since $d \leq 2^k/(ke) - 1/k + 1$, we have that $e2^{-k}((f-1)k + 1) \leq 1$, so Theorem 5.4 applies. □

A slightly better bound was proved by Gebauer, Szabó and Tardos [11], using a different version of LLL known as the Lopsided Lovász Local Lemma. They showed that k-Sat formulas such that every variable appears in at most $\left\lfloor 2^{k+1}/((k+1)e)\right\rfloor$ clauses are always satisfiable. In the same paper, they proved this bound to be tight.

Theorem 5.5 only holds for CNF formulas such that all clauses have equal width. In the following theorem, we generalise this result to arbitrary CNF formulas using the Asymmetric Lovász Local Lemma.

Theorem 5.6. *Let φ be an instance of k-Cnf-Sat. If every variable appears in at most $f_w := \left\lfloor \frac{2^{w-3}}{k^2} \right\rfloor$ clauses of width w, then φ is satisfiable.*

Proof. Let $A_1, A_2, \ldots, A_m$ denote the events that clause $C_1, C_2, \ldots, C_m$ are not satisfied and let $x_j = 2^{-(w-2)}$ if event A_j corresponds to a clause of width w. Note that φ has at most $\lfloor 2^{1-3}/k^2 \rfloor = 0$ clauses of width 1 and $\lfloor 2^{2-3}/k^2 \rfloor = 0$ clauses of width 2, so $x_w \in [0, 1)$ for all w. For notational convenience, we denote $y_w := 2^{w-2}$. Then for every event A_i of width w,

$$\begin{array}{rcccl}
x_i \prod_{j:(i,j)\in E(\Gamma)}(1-x_j) & \geq & \frac{1}{y_w}\prod_{j=3}^{k}\left(1-\frac{1}{y_j}\right)^{wf_j} & = & \frac{1}{y_w}\prod_{j=3}^{k}\left(1-\frac{1}{y_j}\right)^{(y_j-1)\frac{wf_j}{y_j-1}} \\
& \geq & \frac{1}{y_w}\prod_{j=3}^{k}\exp\left(-\frac{wf_j}{y_j-1}\right) & = & \frac{1}{y_w}\exp\left(-\sum_{j=3}^{k}\frac{wf_j}{y_j-1}\right) \\
& \geq & \frac{1}{y_w}\exp\left(-\sum_{j=3}^{k}\frac{w2^{j-3}}{k^2(2^{j-2}-1)}\right) & \geq & \frac{1}{y_w}\exp\left(-\sum_{j=3}^{k}\frac{w2^{j-3}}{k^2(2^{j-3})}\right) \\
& = & \frac{1}{y_w}\exp\left(-\sum_{j=3}^{k}\frac{w}{k^2}\right) & \geq & \frac{1}{y_w e}.
\end{array}$$

Note that the second inequality follows from (7). Filling in the value of y_w gives

$$\frac{1}{y_w e} = 2^{-(w-2)}\frac{1}{e} = 2^{-w}\frac{4}{e} \geq 2^{-w} = \mathbb{P}(A_i),$$

so the condition of Theorem 5.3 holds. Then with positive probability, all clauses of φ are satisfied, which means that there exists at least one satisfying assignment. □

The upper bound f_w in Theorem 5.6 depends on k. If k is large, f_w is extremely small for low values of w. Therefore, we would like to find a bound that depends on w only. The following theorem takes a step in the right direction.

Theorem 5.7. *Let φ be an instance of k-Cnf-Sat. If every variable appears in at most $f_w := \left\lfloor \frac{2^{w-2}}{w^2 k} \right\rfloor$ clauses of width w, then φ is satisfiable.*

Proof. Let $A_1, A_2, \ldots, A_m$ be defined as in Theorem 5.6 and let $x_j = 2^{-(w-1)}$ if event A_j corresponds to a clause of width w. Note that φ has at most $\lfloor 1/(2k) \rfloor = 0$ clauses of width 1, so $x_w \in [0, 1)$ for all w. Let $y_w := 2^{w-1}$ and consider an arbitrary event A_i of size w. Similar to the proof of Theorem 5.7, we have

$$x_i \prod_{j:(i,j)\in E(\Gamma)} (1 - x_j) \geq \frac{1}{y_w}\exp\left(-\sum_{j=3}^{k} \frac{w f_j}{y_j - 1}\right).$$

Filling in y_j and f_j gives

$$\begin{aligned}
\frac{1}{y_w}\exp\left(-\sum_{j=3}^{k} \frac{w 2^{j-2}}{j^2 k\,(2^{j-1}-1)}\right) &\geq \frac{1}{y_w}\exp\left(-\frac{w}{k}\sum_{j=3}^{k} \frac{2^{j-2}}{j^2 2^{j-2}}\right) \\
&\geq \frac{1}{y_w}\exp\left(-\sum_{j=3}^{k} \frac{1}{j^2}\right) \\
&\geq \frac{1}{y_w}\exp\left(-\left(\frac{\pi^2}{6} - \frac{5}{4}\right)\right) \\
&\geq \frac{1}{y_w} e^{-0.4}.
\end{aligned} \tag{8}$$

The third inequality follows from the well-known fact that $\sum_{j=1}^{\infty} j^{-2} = \pi^2/6$. To apply Theorem 5.3, we need to verify that $\mathbb{P}(A_i) = 2^{-w} \leq 1/y_w e^{-0.4} = 2^{-(w-1)} e^{-0.4}$. This inequality simplifies to $e^{0.4} \leq 2$, which is true. Then with positive probability, all clauses of φ are satisfied. □

Using this method, we do not see a way to eliminate k completely from f_w. The summation in (8) contains a factor w which can only be upper bounded by k. A factor $1/k$ is therefore necessary to cancel this out. This means that for large k, Theorem 5.6 and 5.7 only prove satisfiability for formulas with no clauses of small width.

6 Sunflowers

The algorithm of the Sparsification Lemma branches on flowers whose petals potentially overlap. The order in which these flowers are processed is fixed entirely, even for flowers with equal set size. At the cost of a larger sparsification constant, we show that by using sunflowers instead of flowers the ordering of k-flowers can be picked freely. Sunflowers inspired one of the most famous results in extremal combinatorics, the Sunflower Lemma. We discuss this lemma and recent improvements in Section 6.1. In Section 6.2, we describe and analyse a sunflower-based sparsification theorem for 3-SAT. This theorem is meant as a warm-up to illustrate the idea of sunflower sparsification and the majority of the analysis can be done without the Sunflower Lemma. In Section 6.3, we state a similar theorem for k-SAT with arbitrary k. Here we need the Sunflower Lemma to generalise the analysis to from Section 6.2.

6.1 Sunflower Lemma

A *sunflower* is a flower with pairwise disjoint petals. Note that, contrary to our definition of a flower, the heart of a sunflower may be empty. Erdős and Rado [9] showed that every large uniform set system contains a sunflower. Our proof is loosely based on [17].

Lemma 6.1 (Sunflower Lemma)**.** *Let $\mathcal{F}$ be a family of k-sets and suppose that $|\mathcal{F}| > k!(s-1)^k$. Then $\mathcal{F}$ contains a sunflower with s petals.*

Proof. We will prove this statement with induction on k. If $k = 1$, $\mathcal{F}$ consists of singletons. These form a sunflower with $|\mathcal{F}|$ petals and an empty heart. When $|\mathcal{F}| > 1!(s-1)^1 = s-1$, this sunflower has at least s petals, so the statement holds.

Assume the statement holds for k-sets and let $\mathcal{F}$ be a family of $(k+1)$-sets such that $|\mathcal{F}| > (k+1)!(s-1)^{k+1}$. Let $\mathcal{S} = \{S_1, S_2, \ldots, S_l\} \subseteq \mathcal{F}$ be a maximal collection of pairwise disjoint sets. If $l \geq s$, $\mathcal{S}$ contains a sunflower with empty heart and s petals. Therefore, assume that $l < s$. Let $T = S_1 \cup S_2 \cup \cdots \cup S_l$. Then by the maximality of $\mathcal{S}$, T intersects every member of $\mathcal{F}$. Set T has cardinality at most $(k+1)(s-1)$, so there must be a $t \in T$ which is contained in at least

$$\frac{|\mathcal{F}|}{|T|} > \frac{(k+1)!(s-1)^{k+1}}{(k+1)(s-1)} = k!(s-1)^k \tag{9}$$

sets of $\mathcal{F}$.

Consider the set system $\mathcal{F}_t = \{S \setminus \{t\} \mid S \in \mathcal{F}, t \in S\}$. It follows from (9) that $|\mathcal{F}_t| > k!(s-1)^k$, so by the induction hypothesis, $\mathcal{F}_t$ contains a k-sunflower $\mathcal{F}'_t$ with s petals. Adding t to all sets in $\mathcal{F}'_t$ then gives a $(k+1)$-sunflower with s petals in $\mathcal{F}$. □

The proof of Lemma 6.1 can be transformed into a recursive polynomial-time algorithm. A maximal disjoint collection of sets can be found greedily; start with an arbitrary set and keep adding sets that have no overlap with your current selection until this is no longer possible. If this collection contains more than s sets, return it. Otherwise, find t and compute $\mathcal{F}_t$. Recursively apply the algorithm to $\mathcal{F}_t$ to obtain a sunflower $\mathcal{F}'_t$ in $\mathcal{F}_t$ and return $\{F \cup \{t\} \mid F \in \mathcal{F}'_t\}$.

It is a major open problem whether it possible to remove the factor $k!$ from the Sunflower Lemma. Erdős and Rado [9] conjectured, that for every fixed k there exists a constant $C = C(s)$ such that every k-uniform set system of size at least C^k contains a sunflower with s petals. However, this conjecture is still open, even for the case $k = 3$. The Sunflower Lemma was recently improved by Alweiss, Lovett, Wu and Zhang [3], who found that every k-uniform set system of size at least $(Cs^3 \log k \log\log k)^k$ contains a sunflower with s petals for some constant C.

6.2 3-SAT sparsification using sunflowers

The original Sparsification Lemma branches on flowers of sets with equal set size in a specific order, preferring flowers with small petals. This ordering is necessary for the analysis to work, as it is explicitly used in the proof of Lemma 4.7. However, if we can prove this lemma in a different way or replace it altogether, we

no longer need to worry about the petal size of the flowers. This offers more flexibility in the design of the reduction algorithm, which might lead to a better sparsification constant. In this section, we study a new sparsification algorithm which branches on sunflowers. This allows a different method of analysis which does not depend on the petal size of the flowers.

Before we look at arbitrary CNF formulas, we first work out the concept for 3-SAT. In this setting, we will refer to 2-sets as edges and 3-sets as triples. The analysis is similar to the Sparsification Lemma and splits the algorithm into two phases. We bound the degree of every singleton and edge during phase 1 and 2 respectively and calculate the maximum number of sets added along a single recursion path.

Theorem 6.2. *Let $\varepsilon > 0$ be given and let U be a set of n elements. There exists an algorithm that, given a set system $\mathcal{F} \subseteq 2^U$ of sets of size at most three, returns set systems $\mathcal{F}_1, \ldots, \mathcal{F}_l \subseteq 2^U$ with sets of size at most three such that the following holds:*

(1) A set $X \subseteq U$ hits $\mathcal{F}$ if and only if it hits $\mathcal{F}_i$ for some $i \in 1, \ldots, l$.

(2) $\mathcal{F}_1, \ldots, \mathcal{F}_l$ are restrictions of $\mathcal{F}$.

(3) For every $i \in 1, \ldots, l$, every element of U is in at most $d = d(\varepsilon)$ sets of $\mathcal{F}_i$.

(4) $l \leq 2^{\varepsilon n}$.

(5) The running time is at most $\mathcal{O}^*(l)$.

Proof. Let θ_2 and θ_3 be constants to be defined later. We adapt Algorithm 2 by changing flowers into sunflowers and removing the inner for-loop. Again, we assume $\mathcal{F}$ to be inclusion-wise minimal. We also change the role of θ, which no longer depends on the petal size. In Algorithm 3, θ_2 and θ_3 denote the minimum size of a good 2-sunflower and 3-sunflower respectively.

Algorithm 3: `3sunflowerReduce`

Input : A set system $\mathcal{F}$ with sets of size at most 3
Output: A collection of set systems $\mathcal{F}_1, \ldots, \mathcal{F}_l$ as described in Theorem 6.2

```
3sunflowerReduce(F)
    for s = 2, 3 do
        if there exists an s-sunflower S_1, ..., S_z with nonempty heart and z ≥ θ_s petals then
            H = ∩_{i=1}^{z} S_i; F_heart = π(F ∪ {H}); F_petals = π(F ∪ {S_i \ H : i = 1, ..., z})
            return 3sunflowerReduce(F_heart) ∪ 3sunflowerReduce(F_petals)
    return {F}
```

Three types of sunflowers can occur when $k = 3$: 2-sunflowers with petal size 1, 3-sunflowers with petal size 1 and 3-sunflowers with petal size 2. We will denote these as type A, B and C. Table 1 shows an overview of the different types and their effect on the number of edges and singletons when branched on. The number of deleted triples has been left out, as we do not need it for the analysis.

Note that Lemma 4.5 still holds for Algorithm 3, so item (1) and (2) are satisfied. To analyse the number of output instances and the running time, we split each recursion path of the algorithm into two phases. During phase 1, only branches of type A take place. Phase 2 starts with the first type B or C branch.

Observation 6.3. *During phase 2, every element is contained in at most θ_2 edges.*

To see this, note that type B and C branches only take place if every element has degree at most $\theta_2 - 1$, because an element of higher degree would form a type A flower, which would be preferred by the algorithm. Phase 2 starts with a type B or C branch, so at the start of phase 2 every element has degree at most $\theta_2 - 1$. During branches of type B, one edge is added and during branches of type C at most one edge is added per element, since all petals are disjoint. In both cases, the degree of every element increases by at most one, so the maximum degree equals at most θ_2. If an element of degree θ_2 exist, it is contained in a good type A flower. The algorithm will branch on this flower, reducing the maximum degree to at most $\theta_2 - 1$, before considering type B and C flowers again.

Using Observation 6.3, we can upper bound the number of edges that are added throughout the algorithm.

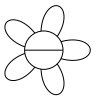
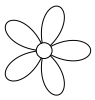
	Branch on heart	Branch on petals
Type A 	1 singleton added $\geq \theta_2$ edges deleted	$\geq \theta_2$ singletons added $\geq \theta_2$ edges deleted
Type B 	1 edge added	$z \geq \theta_3$ singletons added $\leq z(\theta_2 - 1)$ edges deleted
Type C 	1 singleton added $\leq \theta_2 - 1$ edges deleted	$z \geq \theta_3$ edges added

Table 1: Overview of the change in singletons and edges when branching on the heart or petals of each flower type.

Lemma 6.4. *During phase 2, at most $2\theta_2 n$ edges are added along any recursion path of the algorithm.*

Proof. At the end of phase 2, the number of edges present at the start plus the number of edges added must equal the number of deleted and remaining edges. Edges are only deleted when a singleton is added and it follows from Observation 6.3 that at most θ_2 edges are deleted per singleton. Since there can be at most n singletons, at most $\theta_2 n$ edges are deleted during phase 2. The total number of edges at the end of phase 2 equals at most $\theta_2 n$, as every element is contained in at most θ_2 edges. Filling in these two bounds gives

$$\#\text{edges at start} + \#\text{edges added} = \#\text{edges deleted} + \#\text{edges remaining} \leq \theta_2 n + \theta_2 n = 2\theta_2 n,$$

so at most $\leq 2\theta_2 n$ edges are added. □

Every type B or C branch adds at least one edge, so Lemma 6.4 bounds the total number of B and C type branches by $2\theta_2 n$. Type A branches create at least one singleton, which means there can be at most n. If the algorithm branches on the petals, then at least θ_2 singletons, θ_3 singletons or θ_3 edges are created for type A, B or C respectively. Then the total number of petal branches equals at most $n/\theta_2 + n/\theta_3 + 2\theta_2 n/\theta_3$. Together, this bring the total number of recursive calls to at most

$$\binom{n + 2\theta_2 n}{\frac{n}{\theta_2} + \frac{n}{\theta_3} + \frac{2\theta_2 n}{\theta_3}}.$$

By applying Lemma 2.1 with $N = n + 2\theta_2 n$ and $q = n/\theta_2 + n/\theta_3 + 2\theta_2 n/\theta_3$, we find that this equals at most $2^{\lambda n}$ with

$$\lambda \leq \left(\frac{1}{\theta_2} + \frac{1}{\theta_3} + \frac{2\theta_2}{\theta_3}\right) \log\left(\frac{4(2\theta_2 + 1)}{\frac{1}{\theta_2} + \frac{1}{\theta_3} + \frac{2\theta_2}{\theta_3}}\right). \tag{10}$$

Define $\theta_2 = \alpha$ and $\theta_3 = 12\alpha^2$, then (10) simplifies to

$$\left(\frac{1}{\alpha} + \frac{1}{12\alpha^2} + \frac{2\alpha}{12\alpha^2}\right) \log\left(\frac{4(2\alpha + 1)}{\frac{1}{\alpha} + \frac{1}{12\alpha^2} + \frac{2\alpha}{12\alpha^2}}\right) = \left(\frac{14\alpha + 1}{12\alpha^2}\right) \log\left(\frac{8\alpha + 4}{\frac{14\alpha+1}{12\alpha^2}}\right).$$

Choosing $\alpha \approx \log(1/\varepsilon)/\varepsilon$, we get $\lambda \approx \varepsilon$.

To finish the proof, we determine the value of d. Clearly, every element is contained in at most one singleton. It follows from Observation 6.3 that every element is contained in at most θ_2 edges of $\mathcal{F}_i$ for every output instance $\mathcal{F}_i$. In our analysis we did not consider the maximum number of triples that an element can be contained in, but the Sunflower Lemma gives a crude upper bound. If an element x is in more than

$(\theta_3 - 1)^2 2!$ triples, there exists a 3-sunflower with θ_3 petals and x in the heart. Then Algorithm 3 would branch on this flower rather than halt. Taking the sum of all three, we see that

$$d = 1 + \theta_2 + 2!(\theta_3 - 1)^2 = 1 + \alpha + 2(12\alpha^2 - 1)^2 \approx \left(\frac{\log \frac{1}{\varepsilon}}{\varepsilon}\right)^4$$

suffices. □

6.3 k-SAT sparsification using sunflowers

Now we will generalise the sparsification result from the previous section to k-SAT with arbitrary k. Algorithm 3 can easily be adapted to k-CNF formulas by letting the set size s run from 2 to k. However, if we try to copy the analysis of Theorem 6.2 for $k > 3$ we run into the following problem.

To estimate the number of edges added by Algorithm 3, we observed that every element is contained in at most θ_2 edges during phase 2. For $k > 3$, a similar observation is needed for sets of size $3, \ldots, k-1$. Ideally, Observation 6.3 would generalise to sets of larger size, such that for $h < j \leq k-1$, every h-set is contained in at most θ_j j-sets. However, this is not always true. Edges with a common element always form a sunflower, because their petals have size one. This means that all petals must be disjoint, otherwise two edges coincide. If $j - h > 1$, then θ_j sets of size j which contain a common h-set need not form a good sunflower, because distinct j-sets can share more than h elements.

To make Theorem 6.2 work for $k > 3$, we replace Observation 6.3 by a new observation based on the Sunflower Lemma. If more than $(\theta_i - 1)^{i-h}(i-h)!$ sets of size i contain the same h-set H, then there must exist a good i-sunflower with heart H. This gives an exponential upper bound on the number of i-sets when no good i-sunflowers exist. This bound is considerably worse than the linear upper bound of Observation 6.3, but holds for sets of every size.

Theorem 6.5. *Let $\varepsilon > 0$ be given and let U be a set of n elements. There exists an algorithm that, given a set system $\mathcal{F} \subseteq 2^U$ of sets of size at most k, returns set systems $\mathcal{F}_1, \ldots, \mathcal{F}_l \subseteq 2^U$ with sets of size at most k such that the following holds:*

(1) A set $X \subseteq U$ hits $\mathcal{F}$ if and only if it hits $\mathcal{F}_i$ for some $i \in 1, \ldots, l$.

(2) $\mathcal{F}_1, \ldots, \mathcal{F}_l$ are restrictions of $\mathcal{F}$.

(3) For every $i \in 1, \ldots, l$, every element of U is in at most $d = d(\varepsilon, k)$ sets of $\mathcal{F}_i$.

(4) $l \leq 2^{\varepsilon n}$.

(5) The running time is at most $\mathcal{O}^*(l)$.

Proof. Let $\theta_1 < \theta_2 < \cdots < \theta_k$ be constants to be defined later. We generalise Algorithm 3 to k-sets by letting s run from 2 to k.

Algorithm 4: `sunflowerReduce`

Input : A set system $\mathcal{F}$ with sets of size at most 3
Output: A collection of set systems $\mathcal{F}_1, \ldots, \mathcal{F}_l$ as described in Theorem 6.5

```
sunflowerReduce(F)
    for s = 2, ..., k do
        if there exists an s-sunflower S_1, ..., S_z with nonempty heart and z ≥ θ_s petals then
            H = ∩_{i=1}^{z} S_i; F_heart = π(F ∪ {H}); F_petals = π(F ∪ {S_i \ H : i = 1, ..., z})
            return sunflowerReduce(F_heart) ∪ sunflowerReduce(F_petals)
    return {F}
```

Once again, Lemma 4.5 remains satisfied, so only item (3)-(5) need to be verified. We split each recursion path of the algorithm into $k-1$ phases. Phase i starts when the first branch on a flower with sets of size at least i takes place.

Observation 6.6. *When Algorithm 4 branches on an i-sunflower, every $h-$ set is contained in at most $(\theta_j-1)^{j-h}(j-h)!$ j-sets for all $j<i$.*

Observation 6.6 follows directly from Lemma 6.1: if an h-set X is contained in more than $(\theta_j-1)^{j-h}(j-h)!$ j-sets, these sets form a good j-sunflower with X in the heart. Such a sunflower does not exist, because Algorithm 4 would have branched on this flower instead.

To prove the original Sparsification Lemma, we showed with Lemma 4.7 that the i-degree of every set is bounded after phase i of the algorithm. A similar result for Algorithm 4 can be derived from the Sunflower Lemma.

Lemma 6.7. *After the start of phase i, every $h-$ set is contained in at most $(\theta_j-1)^{j-h}(j-h)!+1$ j-sets for all $j\leq i$.*

Proof. Phase i starts with a flower of sets of size at least i, so at the start of phase i every $h-$ *set* is contained in at most $(\theta_j-1)^{j-h}(j-h)!$ j-sets for $j<i$. During a branching step, every h-set is added to at most one new j-set, because the petals of a sunflower are disjoint. If the j-degree of an h-set H exceeds $(\theta_j-1)^{j-h}(j-h)!$ as a result, it is contained in the heart of a good j-sunflower. The algorithm will branch on this flower first before considering flowers of larger sets again, reducing the j-degree of H to at most $(\theta_j-1)^{j-h}(j-h)!$. □

Using these observations, we formulate an analogue of Lemma 6.4.

Lemma 6.8. *Let $f_i=i!\,(\theta_i-1)^i$. Algorithm 4 adds at most nf_i i-sets along any recursion path.*

Proof. Note that i-sets are only added after phase i has begun and that i-sets are deleted when an h-set is added with $h<i$. Lemma 6.7 ensures that no h-set is contained in more than $(\theta_j-1)^{j-h}(j-h)!$ sets of size j for $j\leq i$, which means that at most $(\theta_j-1)^{j-h}(j-h)!$ deletions take place for every added h-set. At the end of the algorithm, Observation 6.6 implies that every element is contained in at most $(\theta_i-1)^{i-1}(i-1)!$ i-sets, so no more than $n(\theta_i-1)^{i-1}(i-1)!$ i-sets remain. Together, this gives the following recurrence relation:

$$\begin{aligned}\#i\text{-sets added} &\leq \#i\text{-sets remaining} + \#i\text{-sets deleted} \\ &\leq n(\theta_i-1)^{i-1}(i-1)! + \sum_{h=1}^{i-1} \#h\text{-sets added}\cdot(\theta_i-1)^{i-h}(i-h)!.\end{aligned} \tag{11}$$

We will prove by induction on i that (11) satisfies our claim using the following lemma.

Lemma 6.9. *For $i=1,2,\ldots,$ it holds that*

$$\sum_{h=1}^{i} h!(i+1-h)! \leq i\cdot i!.$$

Proof. We prove this inequality by induction on i. If $i=1$, then $\sum_{h=1}^{i} h!(i+1-h)! = 1!(1+1-1)! = 1 = 1\cdot 1!$, so the inequality is satisfied. Suppose that $\sum_{h=1}^{i} h!(i+1-h)! \leq i\cdot i!$. Then for $i+1$, we get

$$\begin{aligned}\sum_{h=1}^{i+1} h!(i+2-h)! &= 1!(i+1)! + 2!i! + \cdots + i!2! + (i+1)!1! \\ &\leq (i+1)\left(1!i! + 2!(i-1)! + \cdots + i!1!\right) + (i+1)!1! \\ &= (i+1)\left(\sum_{h=1}^{i} h!(i+1-h)!\right) + (i+1)! \\ &\leq (i+1)\cdot i\cdot i! + (i+1)! \\ &= (i+1)\cdot(i+1)!.\end{aligned}$$

□

The number of singletons added equals at most n. If we choose $\theta_1 \geq 2$, then $n \leq n(\theta_1 - 1) = n \cdot 1! \, (\theta_1 - 1)^1$. Suppose that after the start of phase i at most nf_i i-sets are added. Then for phase $i+1$, we get

$$\begin{aligned} \#(i+1)\text{-sets added} &\leq n \cdot i! \, (\theta_{i+1} - 1)^i + \sum_{h=1}^{i} \#h\text{-sets added} \cdot (\theta_{i+1} - 1)^{i+1-h} \, (i+1-h)! \qquad (12) \\ &\leq n \cdot i! \, (\theta_{i+1} - 1)^{i+1} + \sum_{h=1}^{i} n \cdot h! \, (\theta_h - 1)^h \, (\theta_{i+1} - 1)^{i+1-h} \, (i+1-h)! \\ &= n \cdot i! \, (\theta_{i+1} - 1)^{i+1} + n \, (\theta_{i+1} - 1)^{i+1} \sum_{h=1}^{i} h!(i+1-h)!. \end{aligned}$$

It follows from Lemma 6.9 that (12) equals at most

$$n \cdot i! \, (\theta_{i+1} - 1)^{i+1} + n \, (\theta_{i+1} - 1)^{i+1} \, i \cdot i! = n \cdot (i+1)! \, (\theta_{i+1} - 1)^{i+1} = nf_{i+1}.$$

□

A direct consequence of Lemma 6.8 is that no more than nif_i sets of size at most i are added.

Let $\theta_1 = \alpha, \theta_i = f_{i-1}(i-1)\alpha$ for some α to be determined later. The maximum depth of a recursion branch equals $\sum_{i=1}^{k} nf_i \leq nkf_k$, since every branch adds at least one set. During a petal branch on an i-sunflower, at least θ_i sets of size at most $i-1$ are added to the set system. Then the number of petal branches on i-sunflowers is upper bounded by

$$\frac{n(i-1)f_{i-1}}{\theta_i} = \frac{n(i-1)f_{i-1}}{f_{i-1}(i-1)\alpha} = \frac{n}{\alpha}$$

and the total number of petal branches equals at most kn/α. This limits the total number of recursive calls to $\binom{nkf_k}{nk/\alpha}$. Set $N = nkf_k$ and $q = nk/\alpha$, then by Lemma 2.1, this equals at most $2^{\lambda n}$ with

$$\lambda \leq \frac{k}{\alpha} \log(4\alpha f_k) = \frac{k}{\alpha} \log\left(4\alpha \, (\theta_k - 1)^k \, k!\right) \leq \frac{k}{\alpha} \log\left(4\alpha\theta_k^k k!\right). \qquad (13)$$

Note that θ_i is defined recursively by the formula

$$\theta_{i+1} = if_i\alpha = i \cdot i! \, (\theta_i - 1)^i \, \alpha \leq i \cdot i! \theta_i^i \alpha. \qquad (14)$$

Lemma 6.10. *For $i = 1, \ldots, k-1$, it holds that $\theta_{i+1} \leq (i \cdot i!)^{i^{i-1}} \, \alpha^{i! + i^{i-1}}$.*

Proof. We prove the claim by induction on i. For $i = 1$, (14) gives $\theta_2 \leq \alpha^2 = (1 \cdot 1!)^{1^0} \alpha^{1! + 1^0}$. Suppose the statement holds for θ_i. Then for θ_{i+1}, we get

$$\begin{aligned} \theta_{i+1} &\leq i \cdot i! \, (\theta_i)^i \, \alpha \\ &\leq i \cdot i! \left(((i-1) \cdot (i-1)!)^{(i-1)^{i-2}} \, \alpha^{(i-1)! + (i-1)^{i-2}}\right)^i \alpha \\ &\leq (i \cdot i!)^{1 + (i-1)^{i-2} \cdot i} \alpha^{1 + \left((i-1)! + (i-1)^{i-2}\right) \cdot i} \\ &\leq (i \cdot i!)^{i^{i-2} \cdot i} \alpha^{\left(i! + i^{i-2}\right) \cdot i} \\ &\leq (i \cdot i!)^{i^{i-1}} \alpha^{(i+1)! + i^{i-1}}. \end{aligned}$$

□

It follows from Lemma 6.10 and Equation (13) that λ equals at most

$$\frac{k}{\alpha} \log\left(4k \left(((k-1) \cdot (k-1)!)^{(k-1)^{k-2}} \, \alpha^{(k-1)! + (k-1)^{k-2}}\right)^k k!\right),$$

which can be upper-bounded by

$$\frac{k}{\alpha}\log\left(4(k\cdot k!)^{k^{k-1}}\alpha^{k!+k^{k-1}}\right) \tag{15}$$

following the same steps as the proof of Lemma 6.10. We may assume without loss of generality that $k \geq 2$, since 1-Hitting-Set is trivial. Equation (15) then simplifies to

$$\begin{aligned}\frac{k}{\alpha}\log\left(4(k\cdot k!)^{k^{k-1}}\alpha^{k!+k^{k-1}}\right) &\leq \frac{k}{\alpha}\log\left((k\cdot k!)^{k^k}\alpha^{k!+k^{k-1}}\right)\\ &\leq \frac{k}{\alpha}\log\left((k\cdot k!)^{k^k}\alpha^{2k^k}\right)\\ &\leq \frac{k^{1+k}}{\alpha}\log\left(k\cdot k!\alpha^2\right).\end{aligned}$$

Let $\alpha \approx k^{1+k}\log(1/\varepsilon)/\varepsilon$, then $\lambda \approx \varepsilon$, satisfying item (4).

At the end of Algorithm 4, no good sunflowers with nonempty heart exist, hence every element is contained in at most

$$\begin{aligned}(\theta_j-1)^{j-1}(j-1)! &\leq \theta_j^{j-1}(j-1)!\\ &= \left(((j-1)\cdot(j-1)!)^{(j-1)^{j-2}}\alpha^{(j-1)!+(j-1)^{j-2}}\right)^{j-1}(j-1)!\\ &\leq (j-1)!^{j^{j-1}}(j-1)^{(j-1)^{j-1}}\alpha^{(j-1)\cdot(j-1)!+(j-1)^{j-1}}\\ &\leq j!^{j^j}\alpha^{(j-1)^{j-1}+(j-1)^{j-1}}\\ &\leq j!^{j^j}\alpha^{j^j}\\ &= (j!\alpha)^{j^j}\end{aligned}$$

j-sets. Then we may pick d to be

$$d = \sum_{j=1}^{k}(j!\alpha)^{j^j} \approx \sum_{j=1}^{k}\left(\frac{j!k^{1+2k}\log\frac{1}{\varepsilon}}{\varepsilon}\right)^{j^j}.$$

□

The sparsification constant of Theorem 6.5 is double exponential, which is considerably worse than the exponential constant of the Sparsification Lemma. With our current analysis we do not see a way to improve this. To achieve an single exponential bound, we need θ_i to be a single exponential function in α. However, to complete the proof our recursive definition $\theta_{i+1} = i\alpha f_i \leq i\cdot i!\theta_i^i\alpha$ seems unavoidable. An equality of the form $\theta_{i+1} = c\cdot\alpha f_i$ is needed to estimate the number of petal branches on an i-flower, which is bounded by $n(i-1)f_{i-1}/\theta_i$. This bound must depend on ε, hence on α, and to keep it from becoming zero or larger than the total number of branches, θ_i should be of the same order of magnitude as f_{i-1}. The inequality $f_i \leq i!\theta_i^i$ follows from an application of the Sunflower Lemma in Lemma 6.8. This bound is optimal in terms of θ_i: we know that for all $s, k \in \mathbb{N}$ there exist k-uniform set systems of size $(s-1)^k$ which have no s-sunflower. We therefore do not expect f_i to be smaller than $(\theta_i-1)^i$.

Under these restrictions, any upper bound of θ_i must be be double exponential in α. If it is possible to achieve a bound which is at least as good as the original Sparsification Lemma using sunflowers, it likely requires a different type of analysis or possibly a modification of the algorithm.

7 Hitting set

The Sparsification Lemma of Impagliazzo, Paturi and Zane sparsifies an instance of Hitting-Set such that all of its hitting sets are preserved, without making assumptions about size of these hitting sets. While this makes the lemma more flexible, it is not the best fit for the Hitting-Set problem.

Given a set system, the natural question to ask is not whether it has a hitting set — we can always construct one by adding every element of the universe U — but whether there exists one of size x for some $x \leq |U|$. This gives us an additional parameter, which can be used to speed up the sparsification process. In this chapter, we analyse a sparsification lemma whose number of outputs and running time depend on x only and compare it to known results obtained by kernelisation.

7.1 Kernelisation

So far, we have only considered sparsification algorithms which take one input and produce several equivalent outputs. A different technique, known as kernelisation, returns only one output whose size depends on a fixed parameter associated with the input problem. This is particularly interesting if the input parameter is small, for example when we want to find a hitting set of size x with $x << n$.

Definition 7.1. *Let A be an instance of a computational problem with a set of parameters P. A kernelisation* `K` *is an algorithm which, given A and P, returns an instance A' satisfying the following requirements.*

- *A' has a solution if and only if A does.*
- *The running time of* `K` *is polynomial in the size of A.*
- *The size of A' depends on P only.*

The output A' is called a kernel of A.

For (k, x)-Hitting-Set, a kernelisation algorithm can be derived from the Sunflower Lemma.

Theorem 7.2 (For example [10])**.** *(k, x)-Hitting-Set admits a kernel with at most $k \cdot k! x^k$ sets and $k^2 \cdot k! x^k$ elements.*

Proof. The reduction algorithm for this problem is based on the following observations.

Observation 7.3. *If an $(x+1)$-sunflower exists, a hitting set of size at most x must hit its heart.*

Indeed, a hitting set which does not hit the heart, must hit all $x+1$ disjoint petals, which is not possible.

Observation 7.4. *A set system $\mathcal{F}$ with more than $k \cdot k! x^k$ sets of size at most k always contains a sunflower with $x+1$ petals.*

If $|F| > k \cdot k! x^k$, there exists an $s \leq k$ such that $\mathcal{F}$ contains more than $k! x^k$ sets of size s. The Sunflower Lemma then implies that $\mathcal{F}$ contains an s-sunflower with $x+1$ petals.

Observation 7.3 and 7.4 lead to the following kernelisation algorithm.

Algorithm 5: `kernelise`

Input : A set system $\mathcal{F}$ with sets of size at most k
Output: A set system $\mathcal{F}'$ such that $\mathcal{F}$ has a hitting set of size at most x if and only if $\mathcal{F}'$ does.

```
1 kernelise(F)
2     if |F| > k · k! x^k then
3         Find a sunflower S with x + 1 petals and heart H
4         F' = (F \ S) ∪ H
5         return kernelise(F')
6     return F
```

Let $\mathcal{F}$ be a set system with sets of cardinality at most k and let $\mathcal{G} =$`kernelise`$(\mathcal{F})$. We verify that $\mathcal{G}$ is a kernel of $\mathcal{F}$.

Consider a single instance of Algorithm 5 which takes a set system $\mathcal{F}$ as input and transforms it into $\mathcal{F}'$. By Observation 7.3, a hitting set of $\mathcal{F}$ must hit the heart of $(x+1)$-sunflower S. Since S is replaced by its heart, this hitting set also hits $\mathcal{F}'$. On the other hand, a sunflower is satisfied when its heart is. Therefore, a hitting set of $\mathcal{F}'$ also hits $\mathcal{F}$. Since this holds for every recursive call of the algorithm, every hitting set of $\mathcal{F}$ hits $\mathcal{G}$ and vice versa.

Line 3 of Algorithm 5 can be implemented using the polynomial-time algorithm for the Sunflower Lemma described in Section 6.1. Every time a sunflower is found, the number of sets in $\mathcal{F}$ is reduced by x. This means that at most $|F|/x$ iterations of Algorithm 5 can take place. The total running time is therefore polynomial in the input size.

Finally, Line 2 ensures that Algorithm 5 halts when the number of sets is at most $k \cdot k!x^k$, which is a function of parameters k and x. Since every set contains no more than k elements, the number of elements equals at most $k^2 \cdot k!x^k$. □

In addition to a kernel, Algorithm 5 sometimes gives us information about the existence of a hitting set. If $\emptyset \in$ `kernelise`$(\mathcal{F})$, a sunflower with $x+1$ petals and empty heart was found. A hitting set of size at most x cannot hit $x+1$ disjoint petals, so $\mathcal{F}$ is a 'no'-instance.

Theorem 7.2 can be improved by relaxing the disjointness property of a sunflower. Let $\tau(\mathcal{F})$ denote the minimum cardinality of a hitting set of $\mathcal{F}$. We call a flower $S_1, \ldots, S_z$ with core Y a *relaxed sunflower* with s petals if $\tau(\{S_1 \setminus Y, \ldots, S_z \setminus Y\}) \geq s$. Håstad [13] showed that the Sunflower Lemma can easily be adapted for relaxed sunflowers.

Lemma 7.5. *Let $\mathcal{F}$ be a family of k-sets and $s \geq 1$. If $|\mathcal{F}| > (s-1)^k$, then $\mathcal{F}$ contains a relaxed sunflower with s petals.*

With this lemma, we can formulate a kernelisation algorithm similar to Algorithm 5.

Theorem 7.6. (k,x)-Hitting-Set *admits a kernel with at most kx^k sets and k^2x^k elements.*

Proof. Note that Observation 7.3 still holds for relaxed sunflowers: a hitting set avoiding the heart must hit all petals of the flower and therefore be necessarily of cardinality at least $x+1$. Thus, a hitting set of size at most x must always hit the heart of a relaxed sunflower with $x+1$ petals. If we replace the sunflowers in Observation 7.3 by relaxed sunflowers, Lemma 7.5 implies that a set system with more than $k(x-1)^k$ sets of size at most k always contains a relaxed sunflower with $x+1$ petals. This gives the following kernelisation algorithm.

Algorithm 6: `kerneliseRelaxed`

Input : A set system $\mathcal{F}$ with sets of size at most k
Output: A set system $\mathcal{F}'$ such that $\mathcal{F}$ has a hitting set of size at most x if and only if $\mathcal{F}'$ does.

```
1 kerneliseRelaxed(F)
2   if |F| > kx^k then
3     Find a relaxed sunflower S with x + 1 petals and heart H
4     F' = F \ S ∪ H
5     return kerneliseRelaxed(F')
6   return F
```

Lemma 7.5 is proved by induction on k and directly gives a polynomial-time recursive algorithm to find relaxed sunflowers. The number of recursion steps is bounded by $|F|/x$, as each of them reduces the number of sets by x. Algorithm 6 therefore runs in polynomial time. When it halts, the number of sets is at most kx^k. Since every set contains no more than k elements, the number of elements equals at most k^2x^k. □

A slightly better kernelisation algorithm was found by Abu-Khzam [2]. It returns a kernel of at most $(2k-1)x^{k-1} + x$ elements using crown decompositions.

7.2 A sparsification lemma for (k, x)-Hitting-Set

The Sparsification Lemma works for any instance of Hitting-Set, only using the maximum set size k as a parameter. However, Hitting-Set has a second natural parameter, the size of the hitting set, which is not used in the analysis. In this section, we will show that, given an upper bound x on the size of the hitting set, we can design a sparsification lemma which outputs only $2^{\varepsilon x}$ instances of Hitting-Set, while maintaining the same sparsification constant. Since $x \leq n$, this is never worse than the original Sparsification Lemma and for $x << n$ it is significantly better. This improvement is achieved by also considering the whether the input has a solution. If we see that the current recursion path cannot lead to a 'yes'-instance of (k, x)-Hitting-Set, we terminate this path immediately.

The key observation of this parametrised sparsification lemma is derived from Lemma 4.7. Recall that this lemma states, that every set has bounded i-degree after phase i of the sparsification process. In particular, every element cannot occur in too many sets of size i. Every element of a hitting set can therefore hit only a limited number of i-sets. If there are many i-sets, we may therefore conclude that $\mathcal{F}$ is a 'no'-instance and terminate the current recursion path.

Theorem 7.7. *Let $\varepsilon > 0$, $x, k \in \mathbb{N}$ be given and let U be a set of n elements. There exists an algorithm that, given a set system $\mathcal{F} \subseteq 2^U$ of sets of cardinality at most k, returns set systems $\mathcal{F}_1, \ldots, \mathcal{F}_l \subseteq 2^U$ with sets of size at most k such that the following holds:*

(1) If $\mathcal{F}_1 \cup \cdots \cup \mathcal{F}_l = \emptyset$, then $\mathcal{F}$ has no hitting set of size at most x. Otherwise, $\mathcal{F}_1, \ldots, \mathcal{F}_l$ are restrictions of $\mathcal{F}$ and a set $X \subseteq U$ of cardinality $|X| \leq x$ hits $\mathcal{F}$ if and only if it hits $\mathcal{F}_i$ for some $i \in 1, \ldots, l$.

(2) For every $i \in 1, \ldots, l$, every element of U is contained in at most $d := \left(\frac{4k^3 \log \frac{1}{\varepsilon}}{\varepsilon}\right)^{k-1}$ sets of $\mathcal{F}_i$.

(3) $l \leq 2^{\varepsilon x}$.

(4) The running time of the algorithm is $\mathcal{O}^*(l)$.

Proof. Let $\theta_0, \ldots, \theta_{k-1}$ be constants to be defined later and call a flower $S_1, \ldots, S_z$ with petal size p good if $z \geq \theta_p$. Assume without loss of generality that $\mathcal{F}$ is inclusion-wise minimal. Then Algorithm 7 with initial input $\{\mathcal{F}, c = 0\}$ satisfies the requirements of Theorem 7.7. The algorithm returns an empty collection if $\mathcal{F}$ is a 'no'-instance and a collection of restrictions of $\mathcal{F}$ otherwise. Note that when the output is not the empty collection, this does not automatically mean that $\mathcal{F}$ is a 'yes'-instance.

Similar to Theorem 4.4, we will split every recursion path of Algorithm 7 into $k - 1$ phases. Recall that phase i starts when the algorithm branches on a j-flower with $j \geq i$ for the first time. The parameter c keeps track of the current phase.

Algorithm 7: `reduceHS`

Input : A set system $\mathcal{F}$ with sets of size at most k and an integer c
Output: A collection of set systems $\mathcal{F}_1, \ldots, \mathcal{F}_l$ satisfying Theorem 7.7

```
1  reduceHS(F, c)
2      for i = 1, ..., c do
3          if F contains more than 2xθ_{i-1} sets of size i then
4              return {∅}
5      for s = 2, ..., k do
6          for p = 1, ..., s − 1 do
7              if there exists a good s-flower S_1, ..., S_z with petal size p then
8                  H = ∩_{i=1}^{z} S_i; F_heart = π(F ∪ {H}); F_petals = π(F ∪ {S_i \ H : i = 1, ..., z})
9                  H = reduceHS(F_heart, max{s, c}); P = reduceHS(F_petals, max{s, c})
10                 return H ∪ P
11     return {F}
```

The proof follows the same structure as the proof of Theorem 4.4 and borrows some of its elements. From now on, we will assume that in the final output of the algorithm is nonempty. If this is not the case, we

know that the given instance $\mathcal{F}$ has no hitting set of size at most x, so the algorithm has effectively solved it.

As long as the condition on Line 3 is not satisfied, the algorithm acts the same as Algorithm 2. This means that Lemma 4.5 is still applicable, so the second part of Item (1) holds. We show that $\mathcal{F}$ is indeed a 'no'-instance if the condition on Line 3 is met.

Lemma 7.8. *If $\mathcal{F}$ contains more than $2x\theta_{i-1}$ sets of size i for some $i \leq c$, then $\mathcal{F}$ is a 'no'-instance.*

Proof. Lemma 4.7 still holds for this new algorithm, so every h-set has i-degree at most $2\theta_{i-h}$ for all $i \leq c$. In particular, no element is contained in more than $2\theta_{i-1}$ i-sets. This means that every element of a hitting set can hit at most $2\theta_{i-1}$ sets of size i, hence a hitting set of size at most x can hit at most $2x\theta_{i-1}$ i-sets. □

Define $\beta_j = (4\alpha k)^{j-1}$ for some α to be determined later and let $\theta_0 = 1$, $\theta_j = \alpha\beta_j$. For every recursion path, we will bound the number of sets added before the last branching step in terms of these constants. By the last branching step, we mean the last time the algorithm processes a flower before the recursion path ends. This may seem like an odd point of the algorithm to analyse, but it is necessary to get a guaranteed upper bound.

Before branching on a flower, the algorithm verifies that there are not too many i-sets for all i smaller or equal to the phase number c. On the other hand, there may be a huge number of $(c+1)$-sets, as Lemma 7.8 does not hold for sets of larger size. Therefore, a $(c+1)$-flower may have a large number of petals, which are all added to the system if we branch on it. However, if the number of added sets was large enough to violate Lemma 7.8, this will be detected immediately in Line 3 of the next recursive call.

Lemma 7.9. *Before the last branching step, Algorithm 7 adds less than $2x\beta_i$ sets of size at most i.*

Proof. Let s_i denote the number of sets of size at most i added before the last branching step. Every recursion path of Algorithm 7 either ends when no good flowers remain or when an element is contained in more than $2x\theta_{i-1}$ sets of size i during phase $c \geq i$.

In the first case, Lemma 4.7 ensures that every element has i-degree at most $2\theta_{i-1}$ for all $i \leq c$. Then for these values of i, at most θ_{i-h} i-sets can be deleted by a single h-set and at most $2x\theta_{i-1}$ i-sets remain before the last branching step. This leads to the same recursive formula as in Lemma 4.8, this time with $\#i$-sets remaining $\leq 2x\theta_{i-1}$. In other words,

$$s_i \leq 2x\alpha\beta_{i-1} + 2\alpha \sum_{h=1}^{i-1} s_h\beta_{i-h} + s_{i-1}. \tag{16}$$

The same inductive argument as in Lemma 4.8 can be used to show that $s_i < 2x\beta_i$ satisfies this relation.

For $i > c$, we know that no i-sets have been added at all, because this only happens after phase i. Then s_i equals the number of sets of size at most $c-1$ that have been added along this recursion path, so $s_i < 2x\beta_{c-1} < 2x\beta_i$.

We now consider the second case where an element is contained in more than $2x\theta_{i-1}$ sets of size i during phase $c \geq i$. Suppose that Line 3 is satisfied right after an s-branch and let t denote the phase number before this s-branch took place. The condition in Line 3 was not met before the s-branch, so for $i = 1, \ldots, t$, we know that there existed at most $2x\theta_{i-1}$ i-sets before the last branching step. This means that (16) also holds for these values of i. Observe that the s-branch is the first branch in which sets of size $i > t$ are potentially added. This means that $s_i < 2x\beta_t < 2x\beta_i$ for all $i > t$. □

Lemma 7.9 implies that the recursion depth of Algorithm 7 is at most $2x\beta_{k-1}$, since at least one set of size at most $k-1$ is added with every branch. During a petal branch with petal size i, at least θ_i sets of size i are added. In total, no more than $2x\beta_i$ i-sets are added along any recursion path, so the number of petal branches equals at most

$$\sum_{i=1}^{k-1} \frac{2x\beta_i}{\theta_i} = \sum_{i=1}^{k-1} \frac{2x}{\alpha} < \frac{2kx}{\alpha}.$$

This means that the algorithm can follow at most

$$\binom{2x\beta_{k-1}}{\frac{2xk}{\alpha}} \tag{17}$$

different recursion paths. Using Lemma 2.1 with $N = 2x\beta_{k-1}$ and $q = 2xk/\alpha$, we can upper bound this by $2^{\lambda x}$ with

$$\begin{aligned}\lambda &\leq \frac{2k}{\alpha} \log\left(\frac{4\alpha\beta_{k-1}}{k}\right) \\ &\leq \frac{2k}{\alpha} \log\left(\beta_k\right) \\ &< \frac{2k^2}{\alpha} \log\left(4\alpha k\right).\end{aligned}$$

Let $\alpha \approx k^2 \log(1/\varepsilon)/\varepsilon$, then $\lambda \approx \varepsilon$ and we may pick d to be

$$k\theta_{k-1} = k\left(4\alpha k\right)^{k-2} \leq \left(4\alpha k\right)^{k-1} \approx \left(\frac{4k^3 \log\frac{1}{\varepsilon}}{\varepsilon}\right)^{k-1}.$$

The running time of Algorithm 7 follows from the same argument as in Theorem 4.4. □

For small values of ε, Theorem 7.7 becomes a kernelisation algorithm. If $\varepsilon < 1/x$, say $\varepsilon = 1/2x$, Algorithm 7 runs in polynomial time and outputs one instance $\mathcal{F}_1$ such that every element is contained in at most $d = \left(8xk^3 \log(2x)\right)^{k-1}$ sets of $\mathcal{F}_1$. Then every element of a hitting set hits at most d sets, so we may assume without loss of generality that $|\mathcal{F}_1| \leq xd$. The total number of sets is therefore upper bounded by

$$|\mathcal{F}_1| \leq x\left(8xk^3 \log(2x)\right)^{k-1} = \mathcal{O}\left(\left(x \log x\right)^k\right),$$

which is slightly worse than the kernelisation algorithms in Section 7.1.

To match the $\mathcal{O}\left(x^k\right)$ kernel of Theorem 7.2, choosing $\varepsilon \approx \log x/x$ suffices. Algorithm 7 then outputs $\mathcal{O}\left(x\right)$ instances of size $\mathcal{O}\left(x^k\right)$ and runs in $\mathcal{O}^*\left(x\right)$ time. Apart from the fact that it produces multiple outputs, the algorithm still satisfies all requirements of a kernel. This type of algorithm is known as a Turing kernelisation.

Outputs of size $\mathcal{O}\left(x^{k-1}\right)$, the size of the smallest known Hitting-Set kernel, can be obtained by setting $\varepsilon \approx \log x/(x^{1-\frac{1}{k-1}})$. However, for this value of ε, our sparsification lemma produces $\mathcal{O}\left(x^{x^{1/(k-1)}}\right)$ set systems and has running time $\mathcal{O}^*\left(x^{x^{1/(k-1)}}\right)$.

If we are looking for a hitting set of small size, Theorem 7.7 is a more efficient preprocessing algorithm than the Sparsification Lemma, returning a smaller output in a shorter running time. However, for ETH reductions, the main application of sparsification, it does not help us achieve better bounds. The standard reduction from 3-Sat on n variables and m clauses gives an instance of $(3, n)$-Hitting-Set with $n + m$ sets on a universe of size $2n$. If we first sparsify a CNF formula with the Sparsification Lemma and then perform the reduction, we at most obtain $2^{\varepsilon n}$ instances of $(3, n)$-Hitting-Set with at most $(d(\varepsilon) + 1)n$ sets each. If we first reduce to Hitting-Set and then sparsify using Theorem 7.7, we get at most $2^{\varepsilon n}$ instances of size at most $d(\varepsilon)n$. In both cases, the size of the outputs is linear in n and for ETH reductions the constant is irrelevant.

A naive algorithm can solve (k, x)-Hitting-Set in $\mathcal{O}^*\left(n^x\right)$ steps by looping through al sets of size x. Using iterative compression, this running time can be improved to $\mathcal{O}^*\left(k^x\right)$. However, if ETH holds we cannot do significantly better than that.

Theorem 7.10. *Unless ETH fails, there exists no $k^{o(x)}$ algorithm for (k, x)-Hitting-Set.*

Proof. Let $\varepsilon > 0$ and let φ be a 3-CNF formula with M clauses and N variables. Using the Sparsification Lemma with parameters 3 and $\varepsilon/2$, we obtain an equivalent disjunction of at most $2^{\frac{1}{2}\varepsilon N}$ 3-CNF formulas,

each of which has at most $d(\varepsilon)N$ clauses. The reduction from Section 4.2 then transforms these formulas into set systems of size at most $(d(\varepsilon)+1)N$ on a universe of $2N$ elements. Suppose that there exists an algorithm which solves (k,x)-Hitting-Set in k^{cx} time for any $c>0$. By setting $c=\varepsilon\log_3(2)/2$, we can verify in $2^{\frac{1}{2}\varepsilon N}$ time whether each set system has a hitting set of size N. This means that we can solve 3-Sat in

$$\mathcal{O}^*\left(2^{\frac{1}{2}\varepsilon N}\right)+\mathrm{poly}(N)+2^{\frac{1}{2}\varepsilon N}2^{\frac{1}{2}\varepsilon N}=\mathcal{O}^*\left(2^{\varepsilon N}\right)$$

for any $\varepsilon>0$, contradicting ETH. □

7.3 Lovász Local Lemma and (k,x)-Hitting-Set

In Section 5.2, conditions were given under which k-Sat is always satisfiable. Using the Lovász Local Lemma, we showed that certain CNF formulas are satisfied with positive probability under a random truth assignment. In this Section, we prove a similar condition under which (k,x)-Hitting-Set always has a solution.

For k-Sat it was sufficient to show that a valid solution always exists. However, (k,x)-Hitting-Set also poses a size restriction on the solution. This means that a simple application of the Asymmetric Lovász Local Lemma is not enough, as a randomly sampled set may contain more than x elements. To tackle this problem, we will separately upper bound the probability that a randomly sampled set is not a hitting set and that it is larger than x. The union bound then gives an upper bound on the probability that this random set is not a valid solution. If this probability is smaller than one, there exists at least one hitting hitting set of size at most x.

Theorem 7.11. *Let $\mathcal{F}=\{S_1,\ldots,S_m\}$ be a set system on n elements with sets of size at most $k\in\mathbb{N}$, $p\geq n/k$, and $q=\left(1-\frac{p}{n}\right)^{-1}$. Define l to be the smallest positive integer such that $\frac{q^l}{e}>1$. If every element appears in at most $f_s:=\left\lfloor\frac{c\cdot e\cdot q^{s-1}}{k^2}\right\rfloor$ sets of size s with $c=q-\frac{e}{q^{l-1}}$, then $\mathcal{F}$ has a hitting set of size $2.719p$.*

Proof. Sample a set X by including every element with probability p/n. Let $A_1,A_2,\ldots,A_m$ denote the event that set $S_1,S_2,\ldots,S_m$ is not hit by X. Before we apply the Lovász Local Lemma, we make some useful observations, which will be needed to check the requirements of the lemma.

Observation 7.12.

(1) As $\frac{q^l}{e}>1$ and $\frac{q^{l-1}}{e}\leq 1$, it follows that $1\leq\frac{e}{q^{l-1}}<q$. This means that $0<c\leq q-1$.

(2) For $j<l$, the upper bound f_j becomes

$$f_j=\left\lfloor\frac{c\cdot q^{j-1}}{k^2}e\right\rfloor=\left\lfloor\frac{q^j}{e}\cdot\frac{c}{k^2q}\right\rfloor\leq\left\lfloor\frac{q-1}{k^2q}\right\rfloor=0,$$

where the last inequality follows from Item (1) and the fact that $\frac{q^j}{e}\leq 1$ by the definition of l. Therefore, there are no sets of size smaller than l.

(3) For every $j\geq l$, it holds that $\frac{1}{e}q^j-1\geq\frac{c}{e}q^{j-1}$. If $j=l$, this inequality becomes $q-\frac{e}{q^{l-1}}\geq c=q-\frac{e}{q^{l-1}}$. Since the left-hand side of the inequality grows faster than the right-hand side, it holds for all $j\geq l$.

Define $x_i=eq^{-|S_i|}$ and let $y_i=q^i/e$. It follows from Item (2) of Observation 7.12 that $x_i\in[0,1)$. Following the proof of Theorem 5.6, we find that

$$x_i\prod_{j:(i,j)\in E(\Gamma)}(1-x_j)\geq\frac{1}{y_{|S_i|}}\prod_{j=l}^{k}\left(1-\frac{1}{y_j}\right)^{|S_i|f_j}\geq\frac{1}{y_{|S_i|}}\exp\left(-\sum_{j=l}^{k}\frac{|S_i|f_j}{y_j-1}\right).\tag{18}$$

From Item (3) of Observation 7.12, we know that for all $j\geq l$ it holds that

$$\frac{f_j}{y_j-1}\leq\frac{\frac{c}{e}q^{j-1}}{k^2\left(\frac{1}{e}q^j-1\right)}\leq\frac{1}{k^2}.\tag{19}$$

Equation (18) is then lower bounded by

$$\frac{1}{y_{|S_i|}}\exp\left(-\sum_{j=l}^{k}\frac{|S_i|}{k^2}\right) \geq \frac{1}{y_{|S_i|}e}.$$

This means that

$$x_i \prod_{j:(i,j)\in E(\Gamma)}(1-x_j) \geq \frac{1}{y_{|S_i|}e} = q^{|S_i|} = \mathbb{P}(A_i),$$

hence the Asymmetric Lovász Local Lemma implies that

$$\mathbb{P}\left(X \text{ is a hitting set}\right) = \mathbb{P}\left(\overline{A}_1 \cap \cdots \cap \overline{A}_n\right) \geq \prod_{i=1}^{n}(1-x_i) > 0.$$

This probability is lower bounded by

$$\prod_{i=1}^{n}(1-x_i) \geq \prod_{j=l}^{k}\left(1-\frac{1}{e}q^j\right)^{\frac{nf_j}{j}} \geq \exp\left(-\sum_{j=l}^{k}\frac{nf_j}{j(y_j-1)}\right) \geq \exp\left(-\sum_{j=l}^{k}\frac{n}{jk^2}\right) \geq \exp\left(-\frac{n}{k}\right).$$

The third inequality follows from (19).

Even if X is a valid hitting set, it need not necessarily be of size at most $2.719p$. We upper bound the probability that it is too large using the Chernoff bound.

Lemma 7.13 (Chernoff bound, [5])**.** *Let $X_1, ..., X_n$ be independent random variables which take values in $\{0,1\}$. Let $X = X_1 + \cdots + X_n$. Then for any $\delta > 0$,*

$$\mathbb{P}(X > (1+\delta)\,\mathbb{E}(X)) < \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^{\mathbb{E}(X)}.$$

It follows from Lemma 7.13 that

$$\mathbb{P}(|X| > (1+\delta)p) < \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^{p} = e^{\delta p}(1+\delta)^{-p(1+\delta)}$$

for all $\delta > 0$. Set $\delta = 1.719$, then the probability that X is not a valid hitting set of size at most $2.719p$ is

$$\begin{aligned}\mathbb{P}\left((|X| > p) \vee (X \text{ not a hitting set})\right) &\leq \mathbb{P}\left(|X| > 2.719p\right) + \mathbb{P}\left(X \text{ not a hitting set}\right)\\ &< e^{1.719p}2.719^{-2.719p} + \left(1-e^{-\frac{n}{k}}\right)\\ &\leq 0.367615^p + 1 - e^{-\frac{n}{k}}. \qquad (20)\end{aligned}$$

If (20) equals at most 1, X is a valid hitting set of size at most $2.719p$ with positive probability. This condition can be rewritten as

$$0.367615^p \leq e^{-\frac{n}{k}} \qquad \Leftrightarrow \qquad p \geq \frac{n}{k}. \qquad (21)$$

Equation (21) is satisfied by assumption, hence the statement follows. □

8 DNF sparsification

In the previous chapters, we studied CNF formulas, i.e. conjunctions of disjunctions. The converse of this form, a disjunction of conjunctions, is called *disjunctive normal form* (*DNF*). Conjunctions of literals are referred to as *terms* instead of clauses. All other CNF terminology directly translates to DNF.

The concept of sparsification is also studied with respect to DNF formulas, but in a very different way. The Sparsification Lemma returns several CNF formulas, one of which is satisfiable if and only if the original formula was. DNF sparsification takes a probabilistic approach. The goal is, to construct two DNF formulas which upper and lower bound the original formula and, for an arbitrary assignment of the variables, differ from it with small probability. For notational convenience, we will view DNF formulas on n variables as functions from $\{0,1\}^n$ to $\{0,1\}$.

Definition 8.1. *Let f be a Boolean function on n variables. The functions f_u, f_l are ε-sandwiching approximators for f if $f_l(x) \leq f(x) \leq f_u(x)$ for every $x \in \{0,1\}^n$ and*

$$\mathop{\mathbb{P}}_{x\in\{0,1\}^n}(f(x) \neq f_l(x)) \leq \varepsilon, \qquad \mathop{\mathbb{P}}_{x\in\{0,1\}^n}(f(x) \neq f_u(x)) \leq \varepsilon.$$

In this chapter, we look at two DNF sparsification theorems by Gopalan, Meka and Reingold. The first, which makes use of the Sunflower Lemma, is studied in detail, as well as its connections to CNF. For the second theorem, we briefly sketch the proof structure and key concepts, as the proof is beyond the scope of this thesis.

8.1 Sparsification using sunflowers

In Section 6.3, we sparsified CNF formulas by discarding a part of every large sunflower. A similar approach is used in [12] to find ε-sandwiching approximators for DNF formulas. This particular theorem is restricted to DNF formulas in which only one of the literals $x_i, \neg x_i$ appears for every variable x_i. These formulas are called *unate*. Without loss of generality, we may assume that they are also inclusion-wise maximal, as any subset of literals from a satisfied term is also satisfied.

Theorem 8.2. *Let f be a unate DNF formula of width w with m clauses. For every $\varepsilon > 0$, there exist DNF formulas f_u, f_l of width w with at most $\left(w2^w \ln \frac{m}{\varepsilon}\right)^{\mathcal{O}(w)}$ terms that are ε-sandwiching approximators for f.*

Proof. Let $f = \vee_{i=1}^m T_i$ be a unate DNF formula of width w and set $k = 2^w \ln(m/\varepsilon)$. If

$$m > \left(w2^w \ln \frac{m}{\varepsilon}\right)^w = w^w k^w \geq w!(k-1)^w,$$

it follows from the Sunflower Lemma that there exists a sunflower $\vee_{j=1}^k T_{i_j}$ with heart $Y = \cap_{j=1}^k T_{i_j}$ and disjoint petals $p = \vee_{j=1}^k (T_{i_j} \setminus Y)$. Let f' be the DNF formula obtained by replacing $\vee_{j=1}^k T_{i_j}$ with Y. If a sunflower is satisfied, then so is its heart, but the converse is not always true. Therefore, it holds that $f'(x) \geq f(x)$ for any truth assignment x. Moreover, the probability that f and f' differ under a random assignment is upper bounded by

$$\begin{array}{rcll} \mathbb{P}_x\left(f(x) = 0 \wedge f'(x) = 1\right) & = & \mathbb{P}_x\left(\vee_{j=1}^k T_{i_j} = 0 \wedge Y = 1\right) & \leq \quad \mathbb{P}_x\left(p(x) = 0\right) \\ & = & \prod_{j=1}^k \mathbb{P}_x\left(T_{i_j} \setminus Y = 0\right) & \leq \quad \left(1 - \frac{1}{2^w}\right)^k \\ & = & \left(1 - \frac{1}{2^w}\right)^{2^w \ln(\frac{m}{\varepsilon})} & \leq \quad \frac{1}{e}^{\ln(\frac{m}{\varepsilon})} \\ & = & \frac{\varepsilon}{m}. & \end{array}$$

The third inequality follows from the fact that for $d \geq 1$, $(1 - 1/d)^d$ is an increasing function which converges to $1/e$.

We can repeat this procedure as long as the number of terms exceeds $(w2^w \ln(m/\varepsilon))^w$. Let f_u be the resulting DNF. With every replacement, the number of terms decreases by $k-1$. Therefore, at most $m/(k-1)$ replacements take place and

$$\mathop{\mathbb{P}}_x(f(x) \neq f_u(x)) \leq \frac{m}{k-1} \cdot \frac{\varepsilon}{m} \leq \varepsilon.$$

To construct a lower approximator, we again consider a sunflower $\vee_{j=1}^{k} T_{i_j}$. Create a new DNF formula f'' by dropping the first term T_{i_1}. If $\vee_{j=2}^{k} T_{i_j}$ is satisfied, then so is the whole flower, but the converse need not hold if $T_{i_1} = 1$. This means that $f''(x) \leq f(x)$ for all assignments x. The probability that f and f'' differ, equals

$$\begin{array}{rcccl} \mathbb{P}_x\left(f''(x) = 0 \wedge f(x) = 1\right) & \leq & \mathbb{P}_x\left(T_{i_1} = 1 \wedge \vee_{j=2}^{k} T_{i_j} = 0\right) & = & \mathbb{P}_x\left(T_{i_1} = 1\right) \cdot \mathbb{P}_x\left(\vee_{j=2}^{k} T_{i_j} = 0 \mid T_{i_1} = 1\right) \\ & \leq & \frac{1}{2}\,\mathbb{P}_x\left(\vee_{j=2}^{k} T_{i_j} = 0 \mid T_{i_1} = 1\right) & = & \frac{1}{2}\,\mathbb{P}_x\left(\vee_{j=2}^{k} (T_{i_j} \setminus Y) = 0\right) \\ & \leq & \frac{1}{2}\left(1 - \frac{1}{2^w}\right)^{k-1} & \leq & \frac{\varepsilon}{m}. \end{array}$$

Repeat this procedure while more than $\left(w2^w \ln(m/\varepsilon)\right)^w$ terms remain and let f_l be the resulting DNF. With every repetition, one term is dropped, so this happens at most m times. Therefore,

$$\mathop{\mathbb{P}}_{x}\left(f(x) \neq f_l(x)\right) \leq m \cdot \frac{\varepsilon}{m} \leq \varepsilon.$$

□

A DNF (CNF) formula f can be transformed into a CNF (DNF) formula by interchanging '$\vee$' and '$\wedge$' and replacing every literal with its negation. The resulting formula is called the *complement* of f, denoted by $\bar{f}$. For every assignment x, it holds that $\bar{f}(x) = \neg f(x)$. This leads to the following observation.

Observation 8.3. *Let f be a DNF formula. If f_u and f_l are ε-sandwiching approximators for f, then $\bar{f}_l$ and $\bar{f}_u$ are ε-sandwiching approximators for $\bar{f}$.*

In combination with Theorem 8.2, Observation 8.3 directly gives us an algorithm to find ε-sandwiching approximators for unate CNF formulas.

Let φ be a unate width-w CNF formula with m clauses. Its complement $\bar{\varphi}$ is a unate width-w DNF formula with the same number of clauses. By Theorem 8.2, there exist ε-sandwiching approximators f_u and f_l for $\bar{\varphi}$ of size $\left(w2^w \ln(m/\varepsilon)\right)^{\mathcal{O}(w)}$. It follows from Observation 8.3 that $\bar{f}_l$ and $\bar{f}_u$ are ε-sandwiching approximators for φ.

It is not surprising that the roles of f_u and f_l are reversed for CNF formulas; the reduction rules of Theorem 8.2 have the opposite result on a CNF sunflower. For a DNF sunflower, Y is always satisfied if $\vee_{j=1}^{k} T_{i_j}$ is satisfied, but a CNF sunflower $\wedge_{j=1}^{k} T_{i_j}$ is satisfied whenever Y is satisfied. Replacing a flower by its heart therefore gives an upper approximator for DNF and a lower approximator for CNF. This also holds for the second reduction rule. By swapping these rules, we could in fact alter the proof of Theorem 8.2 to work for CNF formulas and obtain the same result as above.

8.2 Sparsification using quasi-sunflowers

In general, overlapping terms make it more difficult to satisfy a DNF formula. Consider for example a formula f such that one variable x appears in many terms. To satisfy any of these terms, x must equal 1, so many satisfying assignments of f require that $x = 1$. However, under a uniformly random truth assignment, the probability that x equals 1 is no larger than for any other variable. Theorem 8.2 exploits this fact by using sunflowers, which have disjoint petals that are therefore 'easy' to satisfy. We can generalise the concept of sunflowers based on this property.

Definition 8.4. *Let $f = \vee_{i=1}^{k} T_i$ be a unate DNF formula with $k > 1$. f is called a γ-quasi-sunflower with heart $Y = \cap_{i=1}^{k} T_i$ if*

$$\mathop{\mathbb{P}}_{x}\left(\vee_{i=1}^{k} (T_i \setminus Y) = 1\right) \geq 1 - e^{-\gamma}.$$

In [12], a DNF sparsification lemma was introduced which makes use of quasi-sunflowers. It achieves a much sharper bound than Theorem 8.2 and also works for non-unate formulas.

Theorem 8.5. *Let f be a DNF formula of width w. For every $\varepsilon > 0$, there exist DNF formulas f_u, f_l of width w with at most $\left(w \log \frac{1}{\varepsilon}\right)^{\mathcal{O}(w)}$ terms that are ε-sandwiching approximators for f.*

We give a short outline of the proof of Theorem 8.5, which is based on the following observations.

(1) Every DNF formula contains a large unate formula.

(2) The Quasi-sunflower Lemma [20]: a unate DNF formula of width w with m terms contains a $\gamma(m)$-quasi-sunflower with

$$\gamma(m) := \frac{1}{5}\left(\frac{m}{w!}\right)^{\frac{1}{w}}.$$

(3) If $f = \vee_{i=1}^{k} T_i$ is a unate DNF, then $\mathbb{P}_x\left(T_1 = 1 \wedge \vee_{i=2}^{k} T_i = 0\right) \leq \mathbb{P}_x\left(\vee_{i=1}^{k} T_i = 0\right)$. In other words, it is more likely that f is not satisfied, than that only the first term T_1 is satisfied.

(4) If $f = g \vee h$ and g_u, g_l are ε-sandwiching approximators of g, then $g_u \vee h, g_l \vee h$ are ε-sandwiching approximators for f.

The first item is a consequence of the probabilistic method. For every pair of literals $x_i, \neg x_i$, discard one of them uniformly at random and remove terms which contain discarded literals. The resulting formula is always unate. Every term remains with probability at least 2^{-w}, so by linearity of expectation, there exists a unate formula with at least $m \cdot 2^{-w}$ terms.

Let f be a DNF formula. Using the first observation, f can be split into $f = g \vee h$, where g is a large unate formula with m' terms. It follows from the Quasi-sunflower Lemma that it contains a $\gamma(m')$-quasi-sunflower. The same reduction rules as in Theorem 8.2 can be applied to this sunflower: to obtain an upper approximator g', replace the flower by its heart and to obtain a lower approximator g'', remove its first term. The probability that g' and g'' differ from g under a random variable assignment can be upper bounded using Item (3). Finally, applying Item (4) gives us approximators $g' \vee h$ and $g'' \vee h$ for f. If we repeat this process until at most $(w \log(1/\varepsilon))^{\mathcal{O}(w)}$ terms remain, we obtain ε-sandwiching approximators which satisfy Theorem 8.5.

8.3 A probabilistic sparsification lemma for CNF

Both the original Sparsification Lemma and the DNF sparsification lemma have their own advantages. The output of the Sparsification Lemma is satisfiable if and only if the input formula is satisfiable, whereas the sandwiching approximators of Theorem 8.5 match with the input formula with high probability. However, this probabilistic approach does allow us to output only two formulas, while for the Sparsification Lemma this could be up to $2^{\varepsilon n}$. In this section, combine elements from both theorems in a probabilistic version of the Sparsification Lemma.

In Theorem 8.2, the randomness is in the equivalence of the input and output. A DNF formula f and its sandwiching approximators correspond with large probability under a random truth assignment. However, the algorithm itself is deterministic. We take the opposite approach: using a randomised algorithm, we output one CNF formula such that every assignment x satisfies the output if and only if it satisfies the original formula with probability at least $2^{-\varepsilon n}$. An easy way to achieve this result is to apply the Sparsification Lemma and simply pick one of the outputs uniformly at random. However, this method is very inefficient, as it needs to compute the entire recursion tree. We present a faster algorithm which only computes one recursion path.

Theorem 8.6. *There exists an algorithm which, given $k \in \mathbb{N}$, $\varepsilon > 0$, and a k-CNF formula φ on n variables, returns a $d(\varepsilon, k)$-sparse k-CNF formula φ' in polynomial time such that*

(1) with probability at least $2^{-\varepsilon n}$, a truth assignment x satisfies φ if and only if it satisfies φ' ,

(2) φ' is a restriction of f.

Proof. Let φ be a k-CNF formula and let $\varepsilon > 0$ be given. This proof will make use of the analysis of Sparsification Lemma, hence we will view φ as a Hitting-Set instance $\mathcal{F}$ and assume it to be inclusion-wise minimal. Let $\mathcal{F}_1, \ldots, \mathcal{F}_l$ be the set systems obtained by applying the Sparsification Lemma to $\mathcal{F}$. A set X hits $\mathcal{F}$ if and only if it hits $\mathcal{F}_i$ for at least one $i \in \{1, \ldots, l\}$. An algorithm which chooses any of the $\mathcal{F}_i$ with probability at least $2^{-\varepsilon n}$ therefore preserves a hitting set X with probability at least $2^{-\varepsilon n}$. Since every satisfying assignment of φ corresponds to a hitting set in the underlying set system, this means that such an algorithm preserves satisfiability with probability at least $2^{-\varepsilon n}$.

Let $T = (V, E)$ be the binary recursion tree of the Sparsification Lemma on input $\mathcal{F}$. Every vertex represents to a recursive call of the sparsification algorithm and its two children correspond to the two recursion options, $\mathcal{F}_{heart}$ and $\mathcal{F}_{petals}$. Every output $\mathcal{F}_i$ corresponds to a leaf of T. Generating a random output $\mathcal{F}_i$ therefore corresponds generating a random path from the root to a leaf of the recursion tree.

The number of leaves of T can be upper bounded using (6) from the proof of the Sparsification Lemma and equals at most $\binom{n\beta_{k-1}}{nk/\alpha}$. Recall that the upper part of this binomial denotes the maximum number of branching steps and the bottom part denotes the maximum number of petal branches. Suppose we start at the root and move down to the vertex v which represents a heart branch. Then the total number of branches in the subtree rooted at v decreases by one, so it has at most $\binom{n\beta_{k-1}-1}{nk/\alpha}$ leaves. Similarly, we can upper bound the number of leaves in the petal branch subtree by $\binom{n\beta_{k-1}-1}{nk/\alpha-1}$.

By doing this recursively, we can for every vertex find upper bounds on the number of leaves in the subtrees rooted at its children. Every vertex can then be represented by a triple $(\mathcal{F}, x, y)$: the set system at that stage of the algorithm and these two upper bounds. Using this extra information, Algorithm 8 satisfies our claim.

Algorithm 8: `randomReduce`

Input : A triple $\left(\mathcal{F}, x = \binom{x_1}{x_2}, y = \binom{y_1}{y_2}\right)$ consisting of a set system and upper bounds on the number of outputs that the Sparsification Lemma would return after branching on $\mathcal{F}_{heart}$ and $\mathcal{F}_{petals}$ respectively

Output: A set system $\mathcal{F}'$ as described in Theorem 8.6

```
randomReduce(F, x, y)
    for s = 2, ..., k do
        for p = 1, ..., s − 1 do
            if there exists a good s-flower S_1, ..., S_z with petal size p then
                H = ∩_{i=1}^{z} S_i
                generate a random bit b which equals 1 with probability x/(x+y)
                if b = 1 then
                    F_heart = π(F ∪ {H}); x = C(x_1 − 1, x_2)
                    return randomReduce(F_heart, x, y)
                else
                    F_petals = π(F ∪ {S_i \ H : i = 1, ..., z}); ; y = C(y_1 − 1, y_2 − 2)
                    return randomReduce(F_petals, x, y)
                end
    return F
```

Since the output of the algorithm corresponds to one of the outputs of the Sparsification Lemma, we know that it is sparse with the same sparsification constant $d(\varepsilon, k)$ and that it is a restriction of φ. It remains to show that the algorithm picks every leaf with probability at least $2^{-\varepsilon n}$.

Lemma 8.7. *Let $v = (\mathcal{F}, x, y)$ be a vertex of T and let w be an arbitrary leaf of the subtree rooted at v. The probability that Algorithm 8 ends w is at least $\frac{1}{x+y}$.*

Proof. Assume without loss of generality that w is in the subtree of v which corresponds to $\mathcal{F}_{heart}$. We proceed by induction.

If v is the parent of w, the probability that the algorithm chooses w equals $x/(x+y)$, which is at least $1/(x+y)$. Suppose that the statement holds for both subtrees of v. The probability that the algorithm picks the child of v corresponding to $\mathcal{F}_{heart}$ equals $x/(x+y)$. The subtree rooted at this child has at most x leaves. Then by the induction hypothesis, the probability that the algorithm ends up in w after recursing on this child is at least $1/x$. Therefore, the probability that we reach w from v is at least $1/x \cdot x/(x+y) = 1/(x+y)$. □

If we fill in the initial upper bounds $x = \binom{n\beta_{k-1}-1}{nk/\alpha}$ and $y = \binom{n\beta_{k-1}-1}{nk/\alpha-1}$ of the root, 8.7 implies that the

probability of ending up in an arbitrary leaf l of T equals at least

$$\frac{1}{\binom{n\beta_{k-1}-1}{\frac{nk}{\alpha}} + \binom{n\beta_{k-1}-1}{\frac{nk}{\alpha}-1}} = \frac{1}{\binom{n\beta_{k-1}}{\frac{nk}{\alpha}}} \geq 2^{-\varepsilon n}.$$

Every branching step of the Sparsification Lemma takes polynomial time. The depth of the recursion tree is at most $n\beta_{k-1}$, so the running time of the algorithm is polynomial in the input size. □

Looking at the statement of Theorem 8.6, it may seem as if we can choose ε as small as we like without any consequences for the running time. It should be noted however, that the constant β_{k-1} depends on ε and increases as ε becomes small. Although the running time is polynomial in theory, choosing a very small ε will result in a large constant, which will dominate the running time for small values of n.

9 Conclusion

9.1 Summary

In this thesis, we looked at the influence of sparseness on the difficulty of a problem instance. We did this by analysing three sparsification algorithms based on the Sparsification Lemma and by formulating conditions under which sparse CNF formulas are always satisfiable. Our main results were the following.

In Chapter 6, we replaced the flowers in the Sparsification Lemma by sunflowers. This gave a more flexible algorithm with regard to the order in which flowers are processed. If there are multiple sunflowers with equal set size, one may be chosen arbitrarily, whereas the original Sparsification Lemma imposed a fixed order on these flowers. However, the algorithm has a double exponential sparsification constant. We do not see a way to improve our current analysis such that this constant matches the original Sparsification Lemma. Likely, this requires a different type of proof or a different algorithm altogether.

In Chapter 7, we proved a sparsification lemma for parametrised Hitting-Set. We showed that for every $\varepsilon > 0$, a set system $\mathcal{F}$ can be rewritten as a collection of $2^{\varepsilon x}$ sparse set systems $\mathcal{F}_1, \ldots, \mathcal{F}_l$ such that every set X with $|X| \leq x$ is a hitting set of $\mathcal{F}$ if and only it hits some $\mathcal{F}_i$. The sparsification constant equals that of Sparsification Lemma and for $x << n$ the running time and number of outputs of our algorithm are much smaller. For small values of ε, the algorithm behaves like a (polynomial Turing) kernelisation with slightly larger kernel than the best known kernelisation algorithms for Hitting-Set.

In Chapter 8, we proposed a randomised sparsification algorithm which branches on the heart or petals of a sunflower according to a cleverly chosen probability distribution. It outputs one CNF formula such that every assignment x satisfies the output if and only if it satisfies the original formula with probability at least $2^{-\varepsilon n}$. The running time of the algorithm is polynomial in the input size. However, the leading coefficient of this polynomial increases sharply when ε becomes small, which means that in practice the algorithm might not be very efficient for very small values of ε.

Our sparsification results showed that any CNF formula and set system can be made sparse. In Chapter 5 we saw that there is a lower bound to the level of sparseness that can be achieved. We proved that a CNF formula is always satisfiable if every variable appears in $f_w := \lfloor 2^{w-2}/(w^2 k) \rfloor$ clauses of width w, generalising a well-known result of Lovász Local Lemma for uniform CNF formulas. A similar result holds for Hitting-Set. For $p \geq n/k$, and $q = (1 - p/n)^{-1}$, a set system always has a hitting set of size $2.719p$ if every element appears in at most $f_s := \left\lfloor c \cdot e \cdot q^{s-1}/k^2 \right\rfloor$ sets of size s, with $c = q - e \cdot q^{1-l}$ and l the smallest positive integer such that $q^l/e > 1$.

9.2 Further research

A major open problem in the field of complexity theory and sparsification is whether the constant of the Sparsification Lemma can be improved. Although our sunflower sparsification lemma did not answer this question, it has two nice characteristics: the order in which flowers with similar set size are processed can be chosen arbitrarily and, since the petals of a sunflower are disjoint, the degree of every element does not increase much during every branching step. Both of these characteristics also hold for quasi-sunflowers. Perhaps a sparsification algorithm which branches on quasi-sunflowers could therefore lead to a better sparsification constant.

As we have seen in Chapter 7, our sparsification lemma for parametrised Hitting-Set does not immediately lead to new theoretical results with regard to ETH. However, it might be useful for practical applications, for example a fast parametrised algorithm which solves (k, x)-Hitting-Set. Since we did not consider such algorithms in this thesis, it would be interesting to explore the hitting set lemma, as well as our other sparsification results, in this context.

An interesting question regarding lower bounds on sparsification is, whether the bounds presented in Section 5.2 and 7.3 are tight. For Theorem 5.7, it would be desirable to eliminate the factor $1/w$ in f_w, because for large w it excludes formulas with clauses of small size. We do not see a way to improve our current analysis to achieve this. Constructing sparse counterexamples which are not trivially satisfiable could give some insight into whether this is possible at all. Theorem 7.11 would benefit from a cleaner statement and a smaller lower bound on the size of the hitting set. In particular, eliminating l and lowering δ would be a great improvement to the theorem. With a tighter analysis of (18) it might be possible to achieve this.

Bibliography

[1] A. Abboud, R. Williams, and H. Yu. More applications of the polynomial method to algorithm design. In *Proceedings of the twenty-sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 218–230. SIAM, 2014.

[2] F. N. Abu-Khzam. A kernelization algorithm for d-hitting set. *Journal of Computer and System Sciences*, 76(7):524–531, 2010.

[3] R. Alweiss, S. Lovett, K. Wu, and J. Zhang. Improved bounds for the sunflower lemma. *arXiv preprint arXiv:1908.08483*, 2019.

[4] C. Calabro, R. Impagliazzo, and R. Paturi. A duality between clause width and clause density for sat. In *21st Annual IEEE Conference on Computational Complexity (CCC'06)*, pages 7–pp. IEEE, 2006.

[5] H. Chernoff. A career in statistics. *Past, Present, and Future of Statistical Science*, 29, 2014.

[6] T. H. Cormen. *Introduction to algorithms*, pages 517–523. MIT press, 2009.

[7] P. Erdős. Some remarks on the theory of graphs. *Bulletin of the American Mathematical Society*, 53(4):292–294, 1947.

[8] P. Erdős and L. Lovász. Problems and results on 3-chromatic hypergraphs and some related questions. *Coll Math Soc J Bolyai*, 10, 01 1974.

[9] P. Erdős and R. Rado. Intersection theorems for systems of sets. *Journal of the London Mathematical Society*, 1(1):85–90, 1960.

[10] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg, 2006.

[11] H. Gebauer, T. Szabó, and G. Tardos. The local lemma is asymptotically tight for sat. *Journal of the ACM (JACM)*, 63(5):1–32, 2016.

[12] P. Gopalan, R. Meka, and O. Reingold. Dnf sparsification and a faster deterministic counting algorithm. *Computational Complexity*, 22(2):275–310, 2013.

[13] J. Håstad, S. Jukna, and P. Pudlák. Top-down lower bounds for depth-three circuits. *Computational Complexity*, 5(2):99–112, 1995.

[14] T. Hertli. 3-sat faster and simpler—unique-sat bounds for ppsz hold in general. *SIAM Journal on Computing*, 43(2):718–729, 2014.

[15] R. Impagliazzo and R. Paturi. Complexity of k-sat. In *Proceedings. Fourteenth Annual IEEE Conference on Computational Complexity (Formerly: Structure in Complexity Theory Conference) (Cat.No.99CB36317)*, pages 237–240, 1999.

[16] R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity? In *Proceedings 39th Annual Symposium on Foundations of Computer Science (Cat. No. 98CB36280)*, pages 653–662. IEEE, 1998.

[17] S. Jukna. *Extremal combinatorics: with applications in computer science*. Springer Science & Business Media, 2011.

[18] R. A. Moser and G. Tardos. A constructive proof of the general lovász local lemma. *Journal of the ACM (JACM)*, 57(2):1–15, 2010.

[19] R. Paturi, P. Pudlák, M. E. Saks, and F. Zane. An improved exponential-time algorithm for k-sat. *Journal of the ACM (JACM)*, 52(3):337–364, 2005.

[20] B. Rossman. The monotone complexity of k-clique on random graphs. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, pages 193–201. IEEE, 2010.

[21] J. Spencer. Asymptotic lower bounds for ramsey functions. *Discrete Mathematics*, 20:69 – 76, 1977.

[22] T. Szele. Kombinatorikai vizsgalatok az iranyitott teljes graffal kapcsolatban. *Mat. Fiz. Lapok*, 50(1943):223–256, 1943.

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

Index