Theory and Practice of Exact Algorithms for NP-Hard Problems: Branching and Reduction
(NP困難問題に対する厳密解法の理論と応用: 探索と帰着について)

by

Yoichi Iwata
岩田 陽一

A Doctor Thesis
博士論文

Submitted to
the Graduate School of the University of Tokyo
on December 11, 2015
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Information Science and Technology
in Computer Science

Thesis Supervisor: Hiroshi Imai　今井 浩
Professor of Computer Science

ABSTRACT

Algorithms for computing exact solutions of NP-hard problems are widely studied both theoretically and empirically. In theoretical studies, there are two major fields: exact exponential-time algorithms and FPT algorithms. In these theoretical research fields, many interesting techniques have been developed, and faster and faster algorithms have been obtained over the years. On the other hand, in practice, these theoretical methods have been rarely used, and we often use general solvers such as integer programming solvers (e.g., CPLEX) and SAT solvers (e.g., MiniSAT) or branch-and-bound methods using problem specific lower bounds. However, for these practical methods, few theoretical analysis has been discussed. As just described, there is a gap between the theory and practice. In this thesis, we aim to close the gap by applying theoretical methods to practical problems, and conversely by giving theoretical analysis to practical methods. Especially, we focus on branching algorithms and reductions used for solving problems by SAT solvers.

The branch-and-bound method that uses an LP relaxation lower bound to prune the search space is a widely used practical technique to solve NP-hard problems. In recent years, this practical method has been rediscovered as useful for obtaining theoretically faster FPT algorithms. However, in the existing research, there is a problem on the applicability; it can be applied only to a small number of problems whose LP relaxations are known to admit some nice properties. In this thesis, we first introduce a new concept called the *discrete relaxations*. By applying the discrete relaxations instead of using the LP relaxations (actually, these two relaxations are essentially the same), we make it possible to apply the branch-and-bound methods to wider range of problems. Then, we develop an efficient method to solve the relaxation problems by exploiting the maximum flow algorithm and improve the time complexity for various problems. Finally we propose an algorithm that is obtained by combining this new theoretical algorithm with the branch-and-reduce method which are developed in the field of exact exponential-time algorithms. By conducting experiments on real-world datasets, we empirically show that the algorithm, which are obtained from theoretical studies, is quite practical.

Then, we focus on SAT. SAT is the most basic NP-complete problem and any NP problem can be reduced to SAT in polynomial time. In theory, despite many attempts, no algorithms faster than the naive algorithm trying all the possible 2^n assignments are known. On the other hand, in practice, recent SAT solvers can successfully solve instances with millions of variables reduced from industrial problems, and SAT solvers are widely used to efficiently solve practical problems by reductions. Why SAT solvers can solve very huge real-world instances despite its theoretical hardness? We give an explanation to this question by considering a *width* that measures structuredness of the input. We first propose a new reduction method called the *decomposition-based reductions*. By using the decomposition-based reductions, we show that for many problems, we can reduce an instance of a problem to an instance of SAT by preserving the width and vice versa. From this theoretical result, we can say that nice structures of an original instance can be preserved through a reduction, and by using such nice structures, SAT solvers can solve the reduced instance very efficiently. Finally, by conducting experiments, we empirically confirm the power of decomposition-based reductions.

論文要旨

NP 困難問題に対する厳密解を求めるアルゴリズムは理論・実用の両面で盛んに研究されている．理論の分野では厳密指数時間アルゴリズム，FPT アルゴリズムと呼ばれる二つの研究分野が主流となり研究されており，様々な新しい手法が開発され，計算量も大幅に改善されてきた．一方で現実の実用に於いてはこれら理論の手法が使われることは少なく，整数計画ソルバ (CPLEX など) や SAT Solver(MiniSat など) のような汎用ソルバが用いられたり，問題独自の下界を用いた分枝限定法が使われることが多く，こういった実用手法の理論的な解析は殆ど行われていない．このように理論と実用の間には大きな隔たりがあるのが現状である．本論文では，この隔たりを埋めるために，理論分野で研究されてきた手法の現実への応用及び，その逆の，現実の問題を解くために使われている手法の理論的な解析を行う．特にブランチングに基づく探索手法と，SAT Solver に関連する帰着について扱う．

最適解の下界を与える LP 緩和を用いて探索の枝刈りを行う分枝限定法は，実用に於いて広く用いられている手法であるが，この応用の手法が理論の FPT アルゴリズムに於いてより良い計算量を持つアルゴリズムを設計するのにも有用であることが近年の研究で明らかになってきた．しかし，既存研究では緩和問題がある良い性質を持つことが知られている極わずかな問題にしか適用することが出来ないという問題点があった．本論文ではまず，離散緩和という新しい概念を導入し，LP 緩和の代わりに離散緩和を考えることにより (実は本質的には両者は同じなのであるが)，より幅広い問題への分枝限定法の適用を可能にする．次にこの緩和問題を最大流を用いて非常に効率良く解く手法を開発し，様々な問題に対して計算量の改善を行う．そして最後に，この新しい手法と指数時間アルゴリズムの理論分野で研究されてきた Branch-and-Reduce と呼ばれる探索手法を組み合わせたアルゴリズムを提案し，現実の入力に対しても非常に高速に動作することを実験的に示す．

次に SAT に関連する話題を扱う．SAT は最も基礎的な NP 完全問題であり，任意の NP 問題は SAT に多項式時間帰着が可能である．理論研究に於いては全探索によって 2^n 通り全ての真理値割り当てを全て試すより真に効率的なアルゴリズムは見つかっていない．一方，実用に於いては最新の SAT Solver は現実の問題から帰着された数百万変数といった大規模な入力を解くことに成功しており，様々な現実の問題が SAT に帰着することによって効率的に解かれている．本論文ではまず，理論的には非常に難しいとされる SAT がなぜ応用において非常に高速に解くことが出来るのかについて，入力の構造の良さを測る「幅」に着目することで説明を与える．幅を保つ Decomposition-based Reduction という新しい帰着手法を提案し，これを用いて様々な問題について，幅を保ったまま SAT との相互帰着が可能であることを示す．これにより，元の問題のもつ構造の良さが SAT への帰着後にも保たれており，その良い構造を SAT Solver が活用することで効率的に問題が解かれていると考えることが出来る．最後に，この新しい帰着手法の現実的有用性を実験により確かめる．

Acknowledgements

First of all, I would like to express my deep gratitude to my supervisor, Professor Hiroshi Imai for his precious advice and warm encouragement. I have learned many important things for researchers from his valuable comments. Without his guidance, my five-years research in the graduate school would not have been possible.

I am also grateful to my Ph.D. committee chair, Prof. Naoki Kobayashi, and the committee members, Prof. Ichiro Hasuo, Prof. Satoru Iwata, Prof. Tetsuo Shibuya, and Prof. Reiji Suda, for taking the time and effort to examine this thesis and for giving me valuable comments.

I would also thank all my collaborators, Takuya Akiba, Bingkai Lin, Takanori Hayashi, Hidefumi Hiraishi, Ken-ichi Kawarabayashi, Yuki Kawata, Takanori Maehara, Nozomi Nori, Keigo Oka, Magnus Wahlström, Yosuke Yano, and Yuichi Yoshida. I have been really lucky to have opportunities to work with such talented researchers. Especially, I want to express my special thanks to Takuya Akiba and Ken-ichi Kawarabayashi for providing me a lot of opportunities of collaboration, to Yuichi Yoshida for having extensive discussions over and over, and also to Magnus Wahlström for kindly having welcomed me as a co-author of a joint paper.

I also want to thank all members in Imai Laboratory. I have received a lot of valuable feedback in internal seminars, and also learned a lot of new things from their research. I am also grateful to the secretary of our laboratory, Mami Takahashi, for helping me to go through many procedures. I would like to list the members bellow; Vorapong Suppakitpaisarn, Mami Takahashi, Hidefumi Hiraishi, Alonso Gragera, Toshihiro Tanuma, Bingkai Lin, Prompong Pakawanwong, Yosuke Yano, Naoto Ohsaka, Takeo Asai, Takuto Ikuta, Takanori Hayashi, Shuichi Hirahara, Fumiya Satoh, Makoto Soejima, Tomohiro Katayama, Shogo Nakajima, Kohei Uezato, Satoru Yasuda, Amaury Josse, Rohit Kumar Singh, Masato Edahiro, Francois Le Gall, Kenta Takahashi, Akitoshi Kawamura, Takahiko Satoh, Jean-Francois Baffier, Takuya Akiba, Yuki Kawata, Chihiro Komaki, Seiya Takahara, Simon Klein, Florian Steinberg, Jeremy Cohen, Keigo Oka, Akira Motoyama, Yuto Hirakuri, Holger Thies, Kentaro Yamamoto, Kenta Takahashi, Vorapong Suppakitpaisarn, Norie Fu, Hiroyuki Ohta, Junya Fukawa, Chitchanok Chuengsatiansup, Hiroyuki Miyata, Yoshikazu Aoshima, Akihiro Hashikura, Shunichi Matsuda, and Ly Nguyen.

In addition to the laboratory, I have been a member of complex network and map graph group at JST ERATO Kawarabayashi Large Graph Project. Through weekly seminars and occasional workshops, I have acquired a wide range of knowledge of practical algorithms for real-world networks. In addition to the results contained in this thesis, I have obtained several results in this field by collaborating with group members. I would like to thank all the members of the group, Ken-ichi Kawarabayashi, Yuichi Yoshida, Yutaka Horita, Junichi Teruyama, Taro Takaguchi, Masato Abe, Yosuke Yano, Naoto Ohsaka, Mao Fukudai, Naoki Ma-

suda, Takehisa Hasegawa, Kazuhiro Inaba, Takuya Akiba, Mitsuru Kusumoto, Tatsuro Kawamoto, Ryosuke Nishi, Yuki Kawata, Yukie Sano, Leo Speidel, Ryohei Hisano, Kodai Saito, Daiki Takeuchi, Yoshitaka Murai, and Takuto Ikuta.

I would also like to thank Japan Society for the Promotion of Science for the financial support I received through the Grant-in-Aid for JSPS Fellows (256487).

Finally, I would like to express my gratitude to my family and friends for their generous support.

Contents

1 Introduction **1**
1.1 Branching Algorithms . . . 3
1.2 Reductions . . . 9
1.3 Organization . . . 14

2 Preliminaries **16**
2.1 Notations . . . 16
2.2 Tree-decomposition . . . 18
2.3 Parameterized Complexity and SETH . . . 19

3 Linear-time FPT Branch-and-Bound via Network Flow **21**
3.1 Review of the Existing Algorithms . . . 21
3.1.1 LP Relaxation of Vertex Cover . . . 21
3.1.2 FPT Algorithm for Vertex Cover above LP . . . 22
3.1.3 Reduction from Odd Cycle Transversal . . . 22
3.1.4 Solving the LP Relaxation by Network Flow . . . 23
3.2 Linear-time FPT Algorithm . . . 23
3.2.1 Algorithm Overview . . . 23
3.2.2 Updating the Maximum Flow . . . 25
3.2.3 Computing the Extreme Optimal Solution . . . 25
3.2.4 Computing the Initial Maximum Flow . . . 27

4 Branch-and-Reduce Algorithms in Practice **29**
4.1 Algorithm Overview . . . 29
4.2 Branching Rules . . . 29
4.2.1 Vertex Selection . . . 29
4.2.2 Mirror Branching . . . 30
4.2.3 Satellite Branching . . . 30
4.2.4 Packing Branching . . . 31
4.3 Reduction Rules . . . 32
4.3.1 Reductions from Exponential Algorithms . . . 32
4.3.2 Reductions from FPT Algorithms . . . 33
4.3.3 Reductions from Exponential Algorithms for Sparse Graphs 33
4.3.4 Packing Reductions . . . 34
4.4 Lower Bounds . . . 35
4.4.1 Clique Cover . . . 35
4.4.2 LP Relaxation . . . 35
4.4.3 Cycle Cover . . . 36
4.5 Parameterized Complexity of Vertex Cover above Lower Bounds . 36
4.5.1 Vertex Cover above Clique Cover . . . 36
4.5.2 Vertex Cover above Cycle Cover . . . 37
4.6 Experiments . . . 38

4.6.1 Setup . . . 38
4.6.2 Algorithm Comparison . . . 39
4.6.3 Observations . . . 40

5 FPT Algorithms via Discrete Relaxations 50
5.1 Valued Constraint Satisfaction Problems . . . 50
5.1.1 Definitions . . . 50
5.1.2 Tractable VCSPs . . . 51
5.2 Discrete Relaxations . . . 52
5.3 k-submodular Relaxations . . . 54
5.3.1 Basic k-submodular Functions . . . 55
5.3.2 Bisubmodular Relaxations . . . 56
5.3.3 k-submodular Relaxations . . . 57
5.4 Linear-time FPT Algorithms . . . 58
5.5 Discrete Relaxations for Vertex-deletion Problems . . . 63

6 Equivalence among Problems of Bounded Width 68
6.1 Useful Lemmas for Bounding Tree-width . . . 68
6.2 Overview of Decomposition-based Reductions . . . 69
6.3 Equivalence among Problems of Bounded Tree-width . . . 70
6.3.1 Reduction from Max 2-SAT to SAT . . . 70
6.3.2 Reduction from 3-SAT to Independent Set . . . 72
6.3.3 Reduction from SAT to 3-SAT . . . 75
6.3.4 Reduction from Independent Set to Max 2-SAT . . . 75
6.4 Equivalence between Tree-width and Clique-width . . . 76
6.4.1 From Independent Set Parameterized by Clique-width to SAT Parameterized by Tree-width . . . 77
6.4.2 From 3-SAT Parameterized by Tree-width to Independent Set Parameterized by Clique-width . . . 78
6.5 Reduction from #Perfect Matching to #SAT . . . 79
6.6 Equivalence among Problems of Bounded Branch-width . . . 83

7 Exactly Parameterized NL 87
7.1 Definitions . . . 87
7.2 Problems in EPNL . . . 88
7.3 EPNL-completeness of SAT Parameterized by Path-width . . . 89
7.4 EPNL-completeness of Problems Parameterized by Path-width . . . 91

8 Decomposition-based Reductions in Practice 93
8.1 Review of the Existing Research on Max SAT . . . 93
8.1.1 Algorithms for Partial Max SAT . . . 94
8.1.2 Encoding of Cardinality Constraints . . . 95
8.2 Decomposition-based Reduction . . . 96
8.3 Experiments . . . 99
8.3.1 Setup . . . 99
8.3.2 Comparison and Analysis . . . 100

9 Conclusions 108

References 110

Chapter 1

Introduction

P vs NP is one of the most fundamental problems in theoretical computer science. Today, it is commonly believed that P $\neq$ NP and thus, from the theoretical point of view, we cannot expect efficient (i.e., polynomial-time) algorithms for NP-hard problems. On the other hand, in practice, many real-world instances of NP-hard problems can be efficiently solved by heuristic methods despite the theoretical hardness. Why there exists such a gap?

There would be two major reasons. The first reason is the possibility of fast *exponential-time* algorithms. Under the P $\neq$ NP assumption, we cannot solve NP-hard problems in polynomial time. However, it does not rule out the possibility of algorithms faster than the exhaustive search. Thus, there might be a practical algorithm that can solve an NP-hard problem in very small exponential time (e.g., 1.0001^n), or even in subexponential time. Actually, for many problems, algorithms faster than the exhaustive search are already known and the studies for obtaining much faster algorithms are continuing. This kind of research field is called the *exact exponential-time algorithms*.

We give two examples. 3-SAT is a special case of the satisfiability problem (SAT) where the length of each clause is restricted to be at most three. For this problem, Monien and Speckenmeyer [79] developed an $O^*(1.618^n)$[1]-time algorithm improving the $O^*(2^n)$-time exhaustive search in 1985. After that, a series of improved algorithms have been proposed, and an $O^*(1.308^n)$-time algorithm by Hertli [49] is the current fastest. Another example is Hamiltonicity of undirected graphs. This is a problem to find a simple cycle that passes each vertex exactly once and the trivial brute-force search can solve the problem in $O^*(n!)$ time. In 1962, Bellman [14] and Held and Karp [47] developed an $O^*(2^n)$-time dynamic programming algorithm. After that, no progress had been made for a long time. And finally, for the first time in about a half century, Björklund [17] improved the running time to $O^*(1.657^n)$ in 2010.

When considering polynomial-time tractability, all the NP-complete problems are equivalent, that is, if one of them can be solved in polynomial time, then all of them can be also solved in polynomial time. However, if we look at the exponential time complexity for solving each NP-complete problem more closely, the situation changes; whereas there are problems for which a series of improvements have been made as we saw in the above examples, there are many problems for which no algorithms faster than the exhaustive search or a simple dynamic programming are found. Especially, the current fastest algorithm for SAT, the most basic NP-complete problem, is still the naive $O^*(2^n)$-time brute-force search. Impagliazzo and Paturi [53] conjectured that SAT cannot be solved in $O^*((2-\epsilon)^n)$ time for any $\epsilon > 0$, and this conjecture is called the *Strong Exponential-Time*

[1] $O^*(\cdot)$ hides a factor polynomial in the input size.

Hypothesis (SETH). Although SETH is a relatively minor assumption compared to P $\neq$ NP, it has been widely used to establish conditional lower bounds for many problems in recent years (we will see examples later). From practical point of view, SETH would contradict the current situation of practical SAT solving; recent SAT solvers can successfully solve real-world instances with millions of variables, which would be far from the 2^n lower bound of SETH. Thus the first reason is not enough to fully explain the reason of the gap.

The second reason is that real-world instances have a (hidden) nice structure, and by exploiting the structure (intentionally or unintentionally), practical methods can solve the real-world instances efficiently despite the theoretical worst-case hardness. In order to theoretically analyze the efficiency of algorithms for such special instances, a field of *parameterized complexity* arose. In the standard complexity theory, we analyze the complexity as a function of the input size n. In the parameterized complexity theory, we assume that the hardness of an instance depends not only on the input size n but also on an additional value k which measures the true hardness of the instance. The value k is called a *parameter* and a problem with a parameter is called a *parameterized problem*. There are multiple choices of the parameter for the same problem and if parameters are different, we consider them as different parameterized problems.

Let us see several examples of parameterized problems. The most basic parameter is the solution size for optimization problems. VERTEX COVER is a problem to find the minimum vertex subset C such that for any edge, at least one of its endpoints are contained in C. Then, its parameterization by the solution size is a problem to find a vertex cover of size at most the given value k. Since the number of the subsets of size at most k is bounded by n^k, the exhaustive search algorithm can solve the problem in $O^*(n^k)$ time. Moreover, we can improve the running time to $O^*(2^k)$ by a simple branching algorithm (we will see the detail of this algorithm later). From these complexities, we can see that VERTEX COVER can be solved efficiently if the parameter (i.e., the solution size) is small. Although both algorithms run in polynomial time for a constant parameter value, the situation differs; while the degree of the polynomial of the former one depends on the parameter value k, the degree is independent of the parameter for the latter one. We call algorithms running in time $n^{f(k)}$ for some function f, i.e., polynomial time for any constant value of k, as *XP algorithms*, and algorithms running in time $f(k)n^{O(1)}$, i.e., polynomial time for any constant k and the degree of the polynomial is independent of k, as *fixed-parameter tractable (FPT) algorithms*. The positive goal of parameterized complexity is to establish an FPT algorithm, and if exists, a faster (i.e., with a small $f(k)$ and/or a small $n^{O(1)}$) FPT algorithm.

Another famous parameter is *width* that measures structuredness of the input graph[2]. There are many different width measures and it has been shown that many NP-hard problems can be solved efficiently (i.e., in FPT time or XP time) if the input has a small width. Among them, the most famous one is *tree-width*. The concept of tree-width was independently discovered by several groups of researchers, and Robertson and Seymour introduced the notions *tree-width* and *tree-decomposition* in their graph minor project [92]. Intuitively speaking, tree-width measures how much a graph looks like a tree. Many NP-hard problems can be solved in polynomial time if the input graph is a tree. Similarly, if the input graph has a tree-width k, many problems can be solved in $O^*(c^k)$ time, where

[2]For CNF problems, we consider a *primal graph* of the CNF for defining the width of the input. See Chapter 2 for the definition.

c is a problem-dependent constant. It appears that many graphs in real-world applications have relatively small tree-width. For example, it is shown that outer planar graphs have tree-width at most two and planar graphs have tree-width at most $O(\sqrt{n})$ [69]. Thus, we can assume that real-world spatial networks, such as road networks, have relatively small tree-width, and for such graphs, NP-hard problems can be solved efficiently.

In these theoretical research fields of exact exponential-time algorithms and FPT algorithms, many interesting techniques have been developed, and faster and faster algorithms have been obtained over the years. From these theoretical results, we can now somewhat understand why NP-hard problems can be solved efficiently in practice; even if we cannot expect polynomial-time algorithms, we might be able to solve problems in small exponential time, or real-world instances have nice properties (i.e., have small parameters) which can be exploited to establish efficient (FPT) algorithms. On the other hand, in practice, these theoretically fast methods have been rarely used, but we often use general solvers such as integer programming solvers (e.g., CPLEX) and SAT solvers (e.g., MiniSAT) or branch-and-bound methods using problem specific lower bounds. However, for these practical methods, few theoretical analysis has been discussed. As just described, there is a gap between the theoretically fast (i.e., having small time complexity) methods and practically fast (i.e., having small running time in experiments) methods. In this thesis, we aim to close the gap by applying theoretical methods to practical problems, and conversely by giving theoretical analysis to practical methods. Especially, we focus on branching algorithms and reductions used for solving problems by SAT solvers.

1.1 Branching Algorithms

Branching is a method to exactly solve a problem by recursively splitting the search space into smaller subspaces. Although branching methods are widely used in both theoretical and empirical studies, there is a difference between them. In practice, *branch-and-bound* methods that involve problem-specific lower bounds or *branch-and-cut* methods, which generate new cuts to improve the LP lower bounds, are often used. On the other hand, in the theoretical research on exact algorithms, a kind of branching methods called *branch-and-reduce* methods are often used to obtain better worst-case time complexity. In this method, instead of using lower bounds to prune the search, we apply a plethora of reduction rules to avoid the worst case. For a large number of important problems, such as Independent Set (or, equivalently, Vertex Cover) [106, 27], Dominating Set [57], and Directed Feedback Vertex Set [88], the current fastest algorithms are obtained by this method.

By explaining how the algorithms are designed and analyzed using a simple example, we show why reduction rules are important to obtain better worst-case complexity. Let us consider a very simple algorithm for Vertex Cover that selects a vertex v and branches into two cases: either 1) including v to the vertex cover or 2) discarding v while including its neighbors to the vertex cover. Apparently, this algorithm runs in $O^*(2^n)$ time. Can we prove a better complexity? The answer to this question would be No. When a graph is a set of n isolated vertices, the algorithm needs to branch on each vertex, which takes $\Omega^*(2^n)$ time. To avoid this worst case, we can add the following reduction rule: if a graph is not connected, we can solve each connected component separately. Now, we can assume that v has a degree of at least one. Then, after the second case of the branching, where v is discarded and its neighbors are included, the number of vertices to be

considered decreases by at least two. Let $T(n)$ be a time bound for solving an n-vertex instance. By solving the recurrence of $T(n) \leq T(n-1) + T(n-2)$, we can prove a complexity of $O^*(1.6181^n)$. The worst case occurs when we continue to select a vertex of degree one. Here, we note that if n is at least three, a vertex of degree at least two always exists. Thus, by adding the following branching rule, we can avoid this worst case: select a vertex of the maximum degree. Now, we can assume that v has a degree of at least two, and by solving the recurrence of $T(n) \leq T(n-1) + T(n-3)$, we can prove the complexity of $O^*(1.4656^n)$. We continue this process and create increasingly complex rules to avoid the worst case and improve the complexity. Thus, currently, the theoretically fastest algorithms involve a number of complicated rules. Although much of the current research uses a more sophisticated analytical tool called the measure and conquer analysis [40], the design process is basically the same.

Branching is also the most basic technique to obtain FPT algorithms (called *bounded search tree*). For example, an $O(2^k m)$-time FPT algorithm for Vertex Cover parameterized by the solution size can be easily obtained as follows. We choose an arbitrary uncovered edge uv. Since any vertex cover must contain at least one of u and v, we can branch into two cases: either 1) including u to the vertex cover or 2) including v to the vertex cover. Since in each case, one vertex is included to the vertex cover, when we want to obtain a vertex cover of size at most k, we can bound the depth of the search tree to k. Thus, the number of nodes in the search tree is bounded by 2^k, which leads to the time complexity of $O(2^k m)$. Branch-and-reduce method can also be applied to FPT algorithms for obtaining algorithms with smaller $f(k)$ part. For example, the current fastest FPT algorithms for Vertex Cover parameterized by the solution size is by Chen, Kanj, and Xia [27] which is based on branch-and-reduce and runs in $O^*(1.2738^k)$ time.

In recent years, the branch-and-bound methods based on LP lower bounds have been rediscovered as useful for obtaining FPT algorithms with a smaller $f(k)$ part. Cygan, Marcin Pilipczuk, Michal Pilipczuk, and Wojtaszczyk [34] developed an $O^*(4^{k'})$-time algorithm for Node Multiway Cut, which is a problem of finding a small subset of vertices whose removal makes a given set of terminals separated, parameterized by the *difference between the size of the optimal solution and the LP lower bound* (called Node Multiway Cut above LP). Using this algorithm, they obtained an $O^*(4^k)$-time algorithm for Max 2-SAT parameterized by the number of unsatisfied clauses (called Almost 2-SAT) improving the previous best of $O^*(9^k)$ time [86]. Lokshtanov, Narayanaswamy, Raman, Ramanujan, and Saurabh [71] developed an $O^*(2.3146^{k'})$-time algorithm for Vertex Cover parameterized by the difference between the size of the optimal solution and the LP lower bound (called Vertex Cover above LP). Using this algorithm, they improved the time complexity for Almost 2-SAT to $O^*(2.3146^k)$. These FPT algorithms indicate that the problems Node Multiway Cut and Vertex Cover can be solved efficiently when the LP relaxation provides a lower bound close to the optimum.

While a number of interesting techniques have been developed to improve time complexity in theoretical research, they seldom have been used for empirical studies. There would be two major reasons. First, in theoretical studies of exact exponential-time and FPT algorithms, we often ignore the polynomial part hidden in the O^* notation. This is because when considering exponential-time complexity, an $O(1.999^n n^{100})$-time algorithm is considered to be faster than an $O(2^n n)$-time algorithm, and when considering parameterized complexity, we mainly focus on the $f(k)$ part of the time complexity. Although we may some-

times focus on the $n^{O(1)}$ part of the FPT algorithms, in that case, we often ignore the $f(k)$ part by considering k as a constant. Thus, theoretically fast time complexity does not directly lead to empirically fast running time. And second, as indicated in the above example of reduction rules of branch-and-reduce methods, most techniques are designed only for improving theoretical worst-case analysis and we do not know whether they are really useful for solving real-world instances which would be far from the worst case.

The use of branch-and-bound methods in FPT algorithms is an interesting direction of research for bridging theory and practice. However, there is a problem on the applicability of the FPT branch-and-bound methods; it can be applied only to a small number of problems whose LP relaxations are known to admit nice properties called the *half-integrality* and the *persistency*. Until now, the only problems known to admit these properties are problems related to VERTEX COVER and NODE MULTIWAY CUT. Moreover, the proofs of the properties are very different for VERTEX COVER and NODE MULTIWAY CUT. Thus, applying the method to other problems seems to be a hard task.

In this thesis, we aim to overcome the above three problems by 1) developing FPT algorithms simultaneously having a small $f(k)$ part and a small $n^{O(1)}$ part, 2) showing a practical impact of theoretical branching algorithms, and 3) widening the range of applications of the FPT branch-and-bound methods.

Contribution B1: Linear-time FPT Branch-and-Bound via Network Flow

Though the initial motivation of parameterized complexity is making NP-Hard problems more tractable, unfortunately many FPT algorithms have a disadvantage in their time complexity. For example, the function $f(k)$ might be an astronomical tower of exponentials such as 2^{2^k} or the degree d of the polynomial in n might be quite huge such as n^{10}. Although there have been many studies on FPT algorithms with small $f(k)$ or d, many works pursuing small d often neglect how large $f(k)$ is, and many works pursuing small $f(k)$ often neglect how large d is, which makes the algorithm not practical. Thus, it is desirable to improve these FPT algorithms so that $f(k)$ and d become small *simultaneously*.

To describe our contribution, we need to review previous results. ODD CYCLE TRANSVERSAL is a problem of finding the minimum vertex set whose removal makes the input graph to be bipartite. We use the size of the optimal solution as a parameter. Reed, Smith, and Vetta [91] proved that the problem is FPT by introducing a new technique called *iterative compression*. The running time of their algorithm is $O(3^k knm)$. Based on the graph minor theory, Fiorini, Hardy, Reed and Vetta [38] improved the polynomial part to be linear for planar graphs. For general graphs, Kawarabayashi and Reed [62] developed an $O(f(k)(n+m)\alpha(n+m))$-time algorithm, where $f(k)$ is some (huge) function and $\alpha(\cdot)$ denotes the inverse of the Ackermann function.

ALMOST 2-SAT is a parameterized version of MAX 2-SAT, where the parameter is the minimum number of unsatisfied clauses. Razgon and O'Sullivan [90] proved that ALMOST 2-SAT is FPT by designing an $O(15^k km^3)$-time algorithm, where m is the number of clauses. Their algorithm is also based on iterative compression. Raman, Ramanujan, and Saurabh [86] improved the function $f(k)$ to 9^k by reducing the problem to VERTEX COVER parameterized by the difference between the size of the optimal solution and the size of the maximum matching. Cygan, Marcin Pilipczuk, Michal Pilipczuk, and Wojtaszczyk [34] further improved the $f(k)$ part to 4^k by reducing the problem to NODE MULTIWAY

Cut above LP. And finally, Lokshtanov, Narayanaswamy, Raman, Ramanujan, and Saurabh [71] obtained an $O^*(2.3146^k)$ algorithm by reducing the problem to Vertex Cover above LP. They also showed that many other problems such as Odd Cycle Transversal can be reduced to Vertex Cover above LP and obtained faster (having a smaller $f(k)$ part) algorithms. Here, Odd Cycle Transversal is a problem to find a minimum vertex set whose removal makes the input graph bipartite.

Our first contribution in this thesis is giving an FPT branch-and-bound algorithm for Vertex Cover above LP whose running time is $O(4^k(n+m)+T)$, where T is a time complexity for computing the initial solution of the LP relaxation. In general, we can solve an LP relaxation of a Vertex Cover instance in $O(m\sqrt{n})$ time by using the Hopcroft-Karp algorithm [50]. Moreover, for many important applications such as instances reduced from Almost 2-SAT and Odd Cycle Transversal, the initial solution of the LP relaxation can be easily obtained in linear time. Thus, we can make the polynomial part to *linear* in the input size while keeping its $f(k)$ part to a single exponential time. A key ingredient in our algorithm is network flow. During a search, the instance gradually changes and we need to update the optimal LP solution. To avoid computing it from scratch, we express the LP solution as a flow. Then for each branching, in linear time, we update the flow and extract the optimal LP solution for the resulting instance from the flow. Moreover, by exploiting the structure of the minimum cuts, we can apply the LP-based reduction rule, which is a key to obtain FPT branch-and-bound algorithms, in linear-time.

This result was achieved in joint work with Keigo Oka and Yuichi Yoshida. An extended abstract of the result was published on Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA) 2014 [58]. We mention here that, independently of our work, Ramanujan and Saurabh [87] also have obtained an $O(4^k k^4 n)$-time algorithm for Almost 2-SAT by a different approach, which was published on the same proceedings. In the extended abstract, we gave general results in addition to the results described above. However, we omit them here because they are contained in a more general version described in our third contribution.

Contribution B2: Branch-and-Reduce Algorithms in Practice

Our second contribution is showing a practical impact of theoretical research on branching algorithms. As a benchmark problem, we choose Vertex Cover because it has been both theoretically and empirically well studied. We design an algorithm that combines a variety of rules and lower bounds from several theoretical studies. We also develop new rules, called the *packing branching* and *packing reduction* rules, which are inspired by these previous studies. Then, we conduct experiments on a variety of instances and compare our algorithm with two state-of-the-art empirical methods: a branch-and-cut method by a commercial integer programming solver, CPLEX, and a branch-and-bound method called MCS [99]. Although the rules in our algorithm are not designed for specific instances but are developed for theoretical purposes, the results show that our algorithm is actually quite practical and competitive with other state-of-the-art approaches for several cases.

We will review relations between our algorithm and theoretical studies on exact algorithms for Vertex Cover. As for exact exponential-time algorithms, since Fomin, Grandoni, and Kratsch [40] gave an $O^*(1.2210^n)$-time algorithm by developing the measure and conquer analysis, several improved algorithms have

been developed [64, 26, 106]. Since improving the complexity on sparse graphs is known to also improve the complexity on general graphs [26], algorithms for sparse graphs also have been well studied [89, 26, 106]. Among these algorithms, we use rules from the algorithm for general graphs by Fomin et al. [40], and the algorithm for sparse graphs by Xiao and Nagamochi [106]. These rules are also contained in many of the other algorithms. We also develop new rules inspired from the satellite rule presented by Kneis, Langer, and Rossmanith [64]. Since our algorithm completely contains the rules of the algorithm by Fomin et al., with a slight modification to the other reduction rules, our algorithm also can be proved to run in $O^*(1.2210^n)$ time.

On FPT algorithms, Vertex Cover has been studied under various parameterizations. Among them, the difference between the LP lower bound and the IP optimum is a recently developed parameter; however, many interesting results have already been obtained [34, 71, 58, 59]. While the exact exponential-time algorithms do not use any lower bounds to prune the search, with this parameterization, the current fastest algorithms are based on the branch-and-bound method and use a (simple) LP lower bound to prune the search. In our algorithm, we use the LP-based reduction rule and the fast LP computation method in our first contribution. Since we do not give the parameter (i.e., the difference between the LP lower bound and the solution size) to the algorithm, its search space may not be bounded by the parameter. However, if we were to use the iterative deepening strategy (we do not use it in this experiment), the running time of our algorithm would also be bounded by $O^*(4^k)$. The research on this parameterization also suggests that for many other problems, including Odd Cycle Transversal, the fastest way to solve them is to reduce them into Vertex Cover. Therefore, we conduct experiments on the graph reduced from an instance of Odd Cycle Transversal. The results show that solving Odd Cycle Transversal through the reduction to Vertex Cover strongly outperforms the state-of-the-art algorithm for Odd Cycle Transversal. In our experiments, we used two different lower bounds that may give a better lower bound than LP relaxation. We also investigated the parameterized complexity above these lower bounds.

This result was achieved in joint work with Takuya Akiba. An extended abstract of the result was published on Proceedings of the Seventeenth Workshop on Algorithm Engineering and Experiments (ALENEX) 2015 [1], and a full version was published on Theoretical Computer Science [2].

Contribution B3: FPT Algorithms via Discrete Relaxations

Our third contribution is to widen the range of applications of FPT branch-and-bound methods. Currently, there are two types of problems for which branch-and-bound methods run in FPT time. The first one is Node Multiway Cut, for which Cygan et al. [34] developed an $O^*(4^{k'})$-time algorithm, where k' is the difference between the size of the optimal solution and the LP lower bound. And the second one is Vertex Cover, for which Lokshtanov et al. [71] developed an $O^*(2.3146^{k'})$-time algorithm. Both algorithms are very similar and exploit the following two properties of the LP relaxations:

- There exists an optimal LP solution such that each variable takes a value 0, 1, or $\frac{1}{2}$. (*Half-integrality*)
- If a variable x_v takes an integer value in an optimal LP solution, there always exists an optimal integer solution in which x_v takes the same value.

(*Persistency*)

For Vertex Cover, Nemhauser and Trotter [81] proved the half-integrality and the persistency of the LP relaxation. For Node Multiway Cut, Garg, Vazirani, and Yannakakis [44] proved the half-integrality, and Guillemot [46] and Cygan et al. [34] proved that for a set of vertices reachable from the terminals, the persistency holds. Until now, only the problems related to these two problems are known to admit these properties. Moreover, the proofs of the properties are very different for Vertex Cover and Node Multiway Cut, which makes it difficult to apply the method to other problems.

In order to overcome the problem of applicability, we reverse the way of thinking; if the LP relaxation admit the half-integrality, the problem on a *discrete* domain of $\{0, \frac{1}{2}, 1\}$ would be tractable (because the LP can be solved in polynomial time). Thus, instead of showing the half-integrality of the LP relaxation, we relax the domain by adding $\frac{1}{2}$ and aim to show the tractability of the relaxed problem. For showing the tractability, we can use powerful tools from the study of valued constraint satisfaction problems (VCSPs).

The VCSPs are optimization variants of the constraint satisfaction problems. An instance of VCSPs consists of a set of cost functions, and the objective is to minimize the sum of the cost functions. In the most common setting of the study of VCSPs, for a fixed set of cost functions $\mathcal{F}$, we ask what the complexity becomes when each cost function is restricted to be from the fixed set $\mathcal{F}$. By taking the set $\mathcal{F}$ accordingly, we can express many important problems as VCSPs, and thus, we can systematically study the complexity of various optimization problems. A series of work by Thapper and Živný and by Kolmogorov [97, 65, 98] gave a complete dichotomy theorem to this question; intuitively speaking, if every cost function in $\mathcal{F}$ admits a kind of *submodularity*, then a simple LP relaxation (called the *basic LP relaxation*) can solve the problem exactly, and otherwise, the problem becomes NP-hard. Thus, for proving the tractability of the relaxed problem, it suffices to show that the relaxed problem admits a kind of submodularity. Moreover, interestingly, the proof of the persistency can be easily obtained from this submodularity.

Our framework for establishing FPT branch-and-bound algorithms is as follows. First, we express a problem as VCSPs by defining the domain D and the set of cost functions $\mathcal{F}$ accordingly. Then, we relax the domain to $D' \supset D$ and define the relaxed cost functions $\mathcal{F}'$ on the relaxed domain D'. Here, we call the relaxed problem as a *discrete relaxation* of the original problem. We show that each cost function in $\mathcal{F}'$ admits a kind of submodularity (and thus the discrete relaxation is tractable). And finally, we prove the persistency by exploiting the submodularity.

Using the above framework, we can unify the known proofs for Vertex Cover and Node Multiway Cut. Moreover, we can obtain new results for problems including Unique Label Cover. Unique Label Cover is the defining problem of the Unique Games Conjecture [63], which is of central importance to the field of approximation algorithms. For this problem, Chitnis, Cygan, Hajiaghayi, Marcin Pilipczuk, and Michal Pilipczuk [29] gave the first FPT algorithm that runs in $O^*(|\Sigma|^{O(p^2 \log p)})$ time, where Σ is the label set, and p is the solution cost. Using our approach, we can obtain a significantly improved FPT algorithm that runs in $O^*(|\Sigma|^{2p})$ time for both the edge- and vertex-deletion versions.

For Vertex Cover and the related problems, such as Almost 2-SAT and Odd Cycle Transversal, we can use *bisubmodular* functions, which occur as rank functions of delta-matroids [25]. Moreover, we can generalize the result

to Submodular Vertex Cover. For Edge Multiway Cut and Unique Label Cover, we can use *k-submodular* functions, which are generalizations of submodular and bisubmodular functions (becoming submodular when $k = 1$ and bisubmodular when $k = 2$) and occur as rank functions of multimatroids [51]. By using more complicated submodularity, we can obtain FPT algorithms for Node Multiway Cut and vertex-deletion Unique Label Cover.

As we have described above, the branch-and-bound based on discrete relaxations is a promising approach to establish FPT algorithms and to reduce $f(k)$ part of the running time. However, its $n^{O(1)}$ part is not so small since it relies on linear programming to solve the relaxations. Here the idea of our first contribution (solving the LP relaxation via network flow and applying the LP-based reduction rule in linear time by exploiting the structure of minimum cuts) works for several of our discrete relaxations. This approach generalizes the linear-time FPT algorithms in our first contribution and gives the first linear-time FPT algorithm for Unique Label Cover that runs in $O(|\Sigma|^{2p}m)$ time. Thus the branch-and-bound based on discrete relaxations has a potential to reduce both $f(k)$ and $n^{O(1)}$ simultaneously.

The concept of discrete relaxation was independently discovered by Yoichi Iwata and Yuichi Yoshida, and by Magnus Wahlström, and an extended abstract by Wahlström was published on Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA) 2014 [101] By combining our two results, we wrote a joint version and submitted it to a journal. A preprint is available on arxiv [59]. The joint version contains the following results which are not presented in this thesis (these results had been obtained by Wahlström); 1) another proof for vertex-deletion problems, and 2) an FPT algorithm for Group Feedback Vertex Set, which includes Subset Feedback Vertex Set.

1.2 Reductions

Reductions are widely used in both theory and practice. In theory, reduction is a key tool in the complexity theory. By reducing an instance of a hard (under some complexity assumption) problem to an instance of a target problem, we can prove the hardness of the problem. For example, if we want to show a super-polynomial-time lower bound for a problem A under $\mathrm{P} \neq \mathrm{NP}$, it suffices to show a polynomial-time reduction from SAT to the problem A. Although, a polynomial-time reduction can preserve the polynomial-time solvability, it may increase the exponential-time complexity. For example, in order to prove a $(2-\epsilon)^n$-time lower bound under SETH, showing a polynomial-time reduction from SAT is not enough and we need to reduce an n-variable instance of SAT to a target problem of size at most $n + O(\log n)$, which is much a hard task.

SAT is a fundamental problem in complexity theory. Today, it is widely believed that SAT cannot be solved in polynomial time. This is not only because anyone could not find a polynomial-time algorithm for SAT despite many attempts, but also because if SAT can be solved in polynomial time, any problem in NP can be solved in polynomial time (NP-completeness). When considering polynomial-time tractability, all the NP-complete problems are equivalent, that is, if one of them can be solved in polynomial time, then all of them can be also solved in polynomial time. Similarly, when considering subexponential-time tractability, all the SNP-complete problems are equivalent [54]. However, if we look at the exponential-time complexity for solving each NP-complete problem more closely, the situation changes; whereas the current fastest algorithm for SAT is the naive $O^*(2^n)$-time exhaustive search algorithm, faster algorithms have

been proposed for many other NP-complete problems such as 3-SAT [49], Max 2-SAT [102], and Independent Set [106]. Although there are many problems, including Set Cover and Directed Hamiltonicity[3], for which the current fastest algorithms take $O^*(2^n)$ time, we do not know whether a faster algorithm for one of these problems leads to a faster algorithms for SAT and vice versa. Actually, only a few problems, such as Hitting Set and Set Splitting, are known to be equivalent to SAT in terms of exponential-time complexity [32].

Impagliazzo and Paturi [53] conjectured that SAT cannot be solved in $O^*((2-\epsilon)^n)$ time for any $\epsilon > 0$ (i.e., the exhaustive search is optimal), and this conjecture is called the *Strong Exponential-Time Hypothesis (SETH)*. Under the SETH, conditional lower bounds for several problems have been obtained, including k-Dominating Set [84], problems of bounded tree-width [70, 33], and Edit Distance [12]. However, as we described above, the $(2-\epsilon)^n$-time hardness of SAT does not related to $(2-\epsilon)^n$-time hardness of many other problems such as Set Cover and Directed Hamiltonicity. This raises the following questions; Is it really enough for showing a hardness under SETH? Shouldn't we prove more robust hardness like NP-hardness or SNP-hardness?

In practice, reductions are widely used for solving problems by general solvers such as SAT solvers. For example, if we want to solve a Max SAT instance by using a SAT solver, we reduce (or encode) the instance to an instance of SAT. However, as we described above, for many problems (including Max SAT), no reductions to SAT that preserves the exponential-time complexity are known. Thus, from the theoretical point of view, using SAT solvers to solve these problems seems a bad idea because it may increase the difficulty of the problems. For example, the naive exhaustive search can solve an instance of Max SAT of n variables and m clauses in $O^*(2^n)$ time. However, the standard reduction from Max SAT to SAT increases the number of variables to $O(n+m)$, which leads to the time complexity of $O^*(2^{n+m})$. Isn't it inefficient to solve problems by reductions? Shouldn't we solve problems directly without reductions?

In this thesis, we give answers to the above questions by 1) showing that we can preserve the *width* through reductions by proposing a new reduction technique called *decomposition-based reduction* and 2) proving more robust hardness by introducing a new parameterized complexity class called *Exactly Parameterized NL*. Moreover, 3) we investigate the practical performance of the decomposition-based reduction.

Contribution R1: Equivalence among Problems of Bounded Width

Our fourth contribution in this thesis is proposing a new reduction technique called *decomposition-based reductions* and showing width-preserving reduction among various problems. Using our technique, we can reduce a structured (i.e., having small width) instance to a structured instance of another problem. Thus, if a SAT solver can exploit these structures, it is promising to solve these problems by reductions. Actually for tree-width, it is theoretically proved that current SAT solvers, which are not designed to exploit the tree-width, can efficiently solve instances with small tree-width [11]. Although the idea of decomposition-based reductions is simple, we can obtain various interesting results, which are described blow.

[3]For Undirected Hamiltonicity, a faster algorithm has been proposed in a recent paper by Björklund [17]. However, for Directed Hamiltonicity, the trivial $O^*(2^n)$-time dynamic programming algorithm is still the current fastest.

It has been shown that many NP-hard graph optimization problems can be solved efficiently if the input graph has a nice decomposition. One of the most famous decompositions is *tree-decomposition*, and a graph is parameterized by *tree-width*, the size of the largest bag in the given tree-decomposition of the graph. Intuitively speaking, tree-width measures how much a graph looks like a tree. If we are given a graph and its tree-decomposition of width $\mathbf{tw}$[4], many problems can be solved in $O^*(c^{\mathbf{tw}})$ time, where c is a problem-dependent constant. Recently, Lokshtanov, Marx, and Saurabh [70] showed that many of these algorithms are optimal under the SETH. These results are obtained by reducing an n-variable instance of SAT to an instance of the target problem with tree-width approximately $\frac{n}{\log c}$, where c is a problem dependent constant. However, these reductions are one-way, and thus a faster SAT algorithm may not lead to faster algorithms for these problems. Moreover, there is a possibility that one of these problems has a faster algorithm but the others do not.

The first result is showing the exponential-time equivalence among the problems SAT, 3-SAT, MAX 2-SAT, and INDEPENDENT SET parameterized by the tree-width $\mathbf{tw}$. That is, we show that if one of these problems can be solved in $O^*(2^{\alpha\mathbf{tw}})$ time for some $\alpha > 0$, then any of then can be solved in the same $O^*(2^{\alpha\mathbf{tw}})$ time. For all of these problems, the fastest known algorithms run in $O^*(2^{\mathbf{tw}})$ time [82] and our equivalence result states that this is not a coincidence. Note that an n-variable instance of SAT has tree-width at most $n-1$. Hence from our equivalence result, an $O^*((2-\epsilon)^{\mathbf{tw}})$-time algorithm for INDEPENDENT SET of bounded tree-width implies an $O^*((2-\epsilon)^n)$-time algorithm for the general SAT. Therefore, our equivalence result includes the hardness result by Lokshtanov et al. [70]. We believe that the same technique can be applied to many other problems.

The second result is about equivalence between tree-width and another width measure called *clique-width*. Intuitively, clique-width measures how much a graph looks like a clique, and is defined as the number of labels we need to construct the given graph by iteratively performing certain operations. Similarly to the tree-width case, many problems can be solved in $O^*(c^{\mathbf{cw}})$ time if the given graph has a clique-width $\mathbf{cw}$, where c is a problem-dependent constant [31]. Using the decomposition-based reduction, we show that INDEPENDENT SET parameterized by tree-width and INDEPENDENT SET parameterized by clique-width are equivalent. That is, if INDEPENDENT SET can be solved in $O^*(2^{\alpha\mathbf{tw}})$ time, then it can also be solved in $O^*(2^{\alpha\mathbf{cw}})$ time, and vice versa. The fastest known algorithms for INDEPENDENT SET parameterized by clique-width runs in $O^*(2^{\mathbf{cw}})$ time [31].

It is surprising that we can obtain such strong connections between problems of bounded tree-width and a problem of bounded clique-width because tree-width and clique-width are very different parameters in nature; a complete graph of n vertices has a clique-width two whereas its tree-width is $n-1$. Hence, even if there is an efficient algorithm for a problem of bounded tree-width, it does not immediately imply that there is an efficient algorithm for the same problem of bounded clique-width. However, our result states that a faster algorithm for INDEPENDENT SET of bounded tree-width implies a faster algorithm for INDEPENDENT SET of bounded clique-width. We note that INDEPENDENT SET is chosen because SAT, 3-SAT, and MAX 2-SAT are still NP-complete when its primal graph is a clique ($\mathbf{cw} = 2$). Hence, these problems parameterized by tree-

[4]Obtaining a tree-decomposition of the minimum width is NP-hard. In this thesis, we assume that we are given a decomposition as a part of the input, and a problem is parameterized by the width of the given decomposition.

width and clique-width are not equivalent unless P = NP. We believe that we can obtain similar results for many other problems that can be solved efficiently on graphs of bounded clique-width.

The third result is showing a connection between counting problems of bounded tree-width. We give a tree-width preserving reduction from #Perfect Matching to #SAT and show that if #SAT can be solved in $O^*(2^{\alpha \mathbf{tw}})$ time, then #Perfect Matching can also be solved in $O^*(2^{\alpha \mathbf{tw}})$ time. Here, #SAT is the problem of counting the number of satisfying assignments in the given instance of SAT, and #Perfect Matching is the problem of counting the number of perfect matchings in the given graph. While the standard dynamic programming can solve #SAT in $O^*(2^{\mathbf{tw}})$ time, it takes $O^*(3^{\mathbf{tw}})$ for solving #Perfect Matching. By exploiting fast subset convolution [18], van Rooij, Bodlaender, and Rossmanith [100] improved the running time to $O^*(2^{\mathbf{tw}})$. Hence, #Perfect Matching seems more difficult than #SAT. However, our result states that if we can improve the running time of #SAT of bounded tree-width, then we can improve the running time of #Perfect Matching of bounded tree-width. An interesting aspect of our reduction is that, using gadgets, we can simulate the *zeta transform* and the *Möbius transform*, which are main technical tools used in fast subset convolution. We note that the reduction is one-way and showing the equivalence is left as an open problem.

Finally, we consider another famous width measure called *branch-width* $\mathbf{bw}$, which was developed in Robertson and Seymour's Graph Minors project [93]. Branch-width is closely related to tree-width and there is a linear relation of $\mathbf{bw} \leq \mathbf{tw} + 1 \leq \frac{3}{2}\mathbf{bw}$. Therefore, if a problem can be solved in $O^*(c^{\mathbf{tw}})$ time, then it is also solvable in $O^*(c^{\frac{3}{2}\mathbf{bw}})$ time. Dorn [36] improved this running time for several problems by developing a technique that exploits fast matrix multiplication to speed-up the computation of the recurrence in dynamic programming. For example, Max 2-SAT and Independent Set can be solved in $O^*(2^{\frac{\omega}{2}\mathbf{bw}})$, where $\omega < 2.3729$ [68] is the matrix multiplication exponent.

From the hardness result for problems of bounded tree-width by Lokshtanov *et al.* [70], we can obtain lower bounds of the form $(c - \epsilon)^{\mathbf{bw}}$ for many problems under the SETH using the relation between branch-width and tree-width, where c is a problem-dependent constant. On the other hand, the current fastest algorithms for many problems exploit fast matrix multiplication and have running times of $O^*(c^{\frac{\omega}{2}\mathbf{bw}})$. Bodlaender, van Leeuwen, van Rooij, and Vatshelle [22] asked whether we can obtain $O^*(c^{\mathbf{bw}})$-time algorithms and close the gap without assuming $\omega = 2$. Although it is nice if we can show lower bounds of $(c - \epsilon)^{\frac{\omega}{2}\mathbf{bw}}$ under the SETH, it seems implausible since SETH is apparently nothing to do with matrix multiplication.

Our fourth result is resolving this problem from a different point of view; instead of showing such lower bounds, we show Max 2-SAT and Independent Set parameterized by branch-width are equivalent. That is, we show that if Max 2-SAT can be solved in $O^*(2^{\alpha \mathbf{bw}})$ time, then Independent Set can also be solved in the same $O^*(2^{\alpha \mathbf{bw}})$ time, and vise versa. In contrast to the equivalence result for tree-width, the result for branch-width does not contain SAT and 3-SAT because for these problems, no $O^*(2^{\frac{\omega}{2}\mathbf{bw}})$-time algorithms are known. Actually, if SAT can be solved in $O^*(2^{\frac{\omega}{2}\mathbf{bw}})$ time, then the SETH fails because branch-width is at most $\lceil \frac{2}{3}n \rceil$ and it follows that SAT can be solved in $O^*(2^{\frac{\omega}{2}\mathbf{bw}}) = O^*(2^{\frac{\omega}{3}n}) = O^*(1.7303^n)$ time, using $\omega < 2.3729$ [68]. The current fastest algorithm for Max 2-SAT in terms of the input size was developed by Williams [102] and its running time is $O^*(2^{\frac{\omega}{3}n})$. Obtaining a faster algorithm

for this problem is one of the most famous open problems in the area of exact exponential-time algorithms [103, 104]. Since the branch-width is at most $\lceil \frac{2}{3}n \rceil$, our result implies that improving the running time of Independent Set of bounded branch-width is as hard as obtaining a faster algorithm for the general Max 2-SAT.

These results were achieved in joint work with Yuichi Yoshida and a subset of them is contained in an extended abstract published on Proceedings of the 23rd Annual European Symposium on Algorithms (ESA) 2015 [60].

Contribution R2: Exactly Parameterized NL

Our fifth contribution in this thesis is introducing a new parameterized complexity class EPNL (Exactly Parameterized NL) for showing more robust exponential-time hardness. Intuitively, EPNL is a class of parameterized problems that can be solved by a non-deterministic Turing machine with the space of $k + O(\log n)$ bits, where k is the parameter. Then, we show that many problems parameterized by *path-width* are EPNL-complete[5]. Here, path-width is a graph structural parameter which is related to tree-width and intuitively measures how much a graph looks like a path.

If one of the NP-hard problems can be solved in polynomial time, any problem in NP can be solved in polynomial time. Similarly, if one of the EPNL-hard problems can be solved in $O^*(c^k)$ time, any problem in EPNL can be solved in $O^*(c^k)$ time. Since the class EPNL contains many famous problems, such as Set Cover parameterized by the number of elements and Directed Hamiltonicity parameterized by the number of vertices for which no $O^*((2-\epsilon)^n)$-time algorithms are known, EPNL-hardness is much more robust than SETH-hardness which relies only on the hardness of SAT.

Our proofs of EPNL-completeness are as follows. First, we show the EPNL-completeness of SAT parameterized by path-width by showing that any problem in EPNL can be reduced to SAT parameterized by path-width by simulating the Turing machine by CNF. Then, using decomposition-based reductions, we show that 3-SAT, Max 2-SAT, and Independent Set parameterized by path-width are also EPNL-complete. Since path-width is always at least the tree-width, this immediately implies that the problems parameterized by tree-width are EPNL-hard. We note that the problems parameterized by tree-width would not be EPNL-complete because it is known that SAT of tree-width w cannot be reduced to SAT of path-width $O(w)$ under the assumption of $\mathrm{NL} \neq \mathrm{SAC}^1$ [3].

For these problems, $(2-\epsilon)^{\mathbf{pw}}$-time lower bounds under SETH were already obtained by Lokshtanov et al.[70] However, as we discussed above, our EPNL-hardness results are much more robust than the SETH-hardness. Moreover, EPNL-hardness can be used to drive the lower bounds on the *space* complexity; if one of the EPNL-hard problems can be solved in $O^*(c^k)$ time and $O^*(d^k)$ space, any problem in EPNL can be solved in $O^*(c^k)$ time and $O^*(d^k)$ space. Since EPNL contains problems for which no $O^*(2^n)$-time and $O^*((2-\epsilon)^n)$-space algorithms are known (e.g., Optimal Linear Arrangement and Directed Feedback Arc Set [21]), our results imply that obtaining an $O^*(2^{\mathbf{pw}})$-time and $O^*((2-\epsilon)^{\mathbf{pw}})$-space algorithm for the problems parameterized by path-width are at least as hard as obtaining an $O^*(2^n)$-time and $O^*((2-\epsilon)^n)$-space algorithms

[5] Flum and Grohe [39] introduced a similar class, called para-NL, that can be solved in $f(k) + O(\log n)$ space. Although they showed that a trivial parameterization of an NL-complete problem is para-NL-complete under the standard parameterized reduction, this does not hold in our case because we use a different reduction to define the complete problems.

for these problems. Because SAT can be solved in $O^*(2^n)$ time and polynomial space, lower bounds on the space complexity cannot be obtained under SETH.

This result was achieved in joint work with Yuichi Yoshida and also contained in the extended abstract published on Proceedings of the 23rd Annual European Symposium on Algorithms (ESA) 2015 [60].

Contribution R3: Decomposition-based Reductions in Practice

The final contribution in this thesis is to investigate the power of decomposition-based reductions in practice. As a benchmark problem, we choose a reduction from Max SAT to SAT. There are three reasons for this choice.

First, since 2006, annual competitions of Max SAT solvers (*MaxSAT Evaluation*[6]) have been held, and a variety of industrial Max SAT instances are publicly available.

Second, the use of reductions for solving Max SAT has been shown to be practical. The development of practical MaxSAT solvers has been attracted a lot of attention in recent years and a variety of new solvers have been developed. Among them, the current state-of-the-art approaches for exact Max SAT solving involve reductions to SAT and exploit the power of the state-of-the-art SAT solvers. Especially, one of the approaches, called a *satisfiability-based* approach, uses a simple reduction to SAT. Solvers based on this approach won the first place in MaxSAT Evaluations 2008 [15] and 2010–2012 [66].

Third, it is theoretically shown that if the input CNF has small tree-width, current SAT solvers, which are not designed to exploit the tree-width, can efficiently solve the problem [11]. Although the proved running time is $\mathbf{tw}^{O(\mathbf{tw})}$, which is worse than the dynamic programing of $O^*(2^{\mathbf{tw}})$ time, we can expect that avoiding the increase of tree-width by exploiting the decomposition-based reduction would lead to a faster running time for solving the reduced CNF.

By applying the idea of the decomposition-based reduction to the reduction used in the satisfiability-based approach, we empirically evaluate the power of decomposition-based reduction. Although the decomposition-based reduction does not outperform the existing reduction, through a precise analysis, we confirm that avoiding the increase of the tree-width is important for practical SAT solving. Thus, the decomposition-based reduction has a potential for practical applications.

1.3 Organization

This thesis is organized as follows. In Chapter 2, we introduce notations and definitions used throughout the thesis. Chapters 3–5 contain our contributions related to branching algorithms. We propose an improved FPT branch-and-bound algorithm for Vertex Cover above LP in Chapter 3. By using this algorithm together with ideas from theoretical research on branching algorithms, we empirically show a practical impact of theoretical branching algorithms in Chapter 4. In Chapter 5, we introduce the concept of discrete relaxation and generalize the result of Chapter 3. Chapters 6–8 contain our contributions related to reductions. In Chapter 6, we introduce the idea of decomposition-based reductions and show equivalence among various problems. In Chapter 7, we introduce a new parameterized complexity class EPNL and show EPNL-complete

[6] http://www.maxsat.udl.cat/

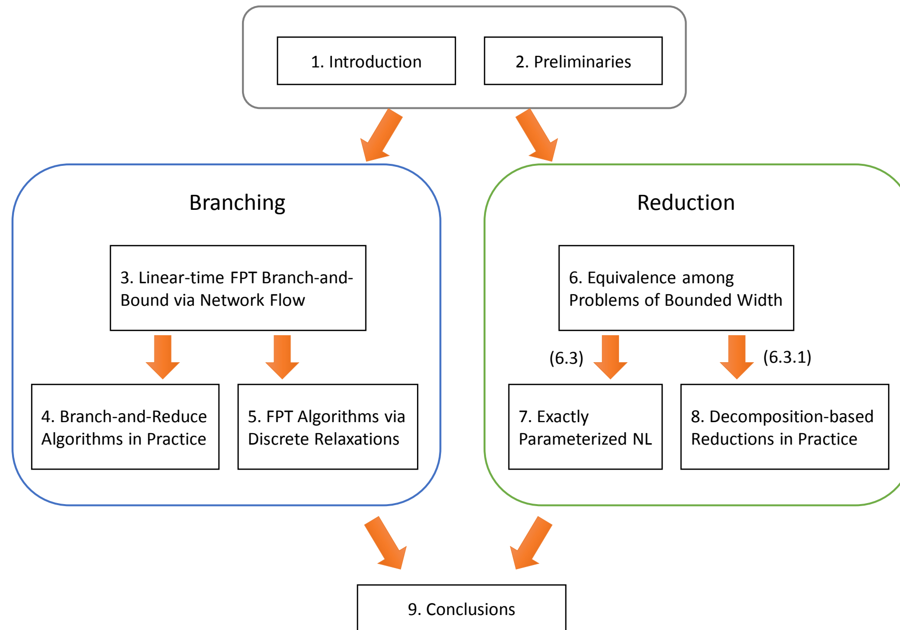

Figure 1.1: The dependencies between the chapters

problems by using the decomposition-based reductions. In Chapter 8, we empirically investigate the power of the decomposition-based reductions. Finally, we conclude in Chapter 9.

Figure 1.1 illustrates the dependencies between the chapters. Chapters 4 and 5 depends on the result of Chapter 3. Chapter 7 depends on Sections 6.1–6.3 of Chapter 6, and Chapter 8 depends on Sections 6.1–6.3.1 of Chapter 6.

Chapter 2

Preliminaries

In this chapter, we introduce notations and definitions used throughout the thesis. First, we give the notations of graphs and CNFs in Section 2.1. Then, we define the tree-decomposition of graphs in Section 2.2. Finally, we introduce the parameterized complexity and the Strong Exponential Time Hypothesis (SETH) in Section 2.3.

2.1 Notations

For an integer k, we denote the set $\{1, 2, \dots, k\}$ by $[k]$ and the set $\{0, 1, \dots, k-1\}$ by $[k]'$. Let $f : U \to \mathbb{R}$ be a function. We write the sum of $f(a)$ over $a \in S \subseteq U$ by $f(S) = \sum_{a \in S} f(a)$.

Undirected Graphs

Let $G = (V, E)$ be an undirected graph. We often use n and m to denote the number of vertices $|V|$ and the number of edges $|E|$. We denote the *degree* of a vertex u as $d_G(u)$. We denote the *neighborhood* of a vertex u by $N_G(u) = \{v \in V \mid \{u, v\} \in E\}$, and the *closed neighborhood* of u by $N_G[u] = N_G(u) \cup \{u\}$. Similarly, we denote the neighborhood of a subset $S \subseteq V$ by $N_G(S) = \bigcup_{v \in S} N_G(v) \setminus S$, and the closed neighborhood by $N_G[S] = N_G(S) \cup S$. The set of vertices at distance d from a vertex u is denoted by $N_G^d(u)$. We drop the subscript G when it is clear from the context. For a subset $S \subseteq V$, let $G[S] = (S, \{\{u, v\} \in E \mid u \in S, v \in S\})$ denote the *subgraph induced by S*. For a vertex subset $S \subseteq V$, we denote the induced subgraph on the vertex set $V \setminus S$ by $G - S$. When S is a single vertex v, we simply write $G - v$.

A *vertex cover* of a graph $G = (V, E)$ is a vertex subset $C \subseteq V$ such that, for any edge $e \in E$, at least one of its endpoints are contained in C. Vertex Cover is a problem to find a minimum vertex cover of a given graph. An *independent set* of a graph is a vertex set $S \subseteq V$ such that $G[S]$ has no edges. Independent Set is a problem to find a maximum independent set of a given graph. For any vertex cover C, $V \setminus C$ becomes an independent set. Therefore Vertex Cover and Independent Set are equivalent problems. However, if we consider their parameterizations by the solution size, they become different parameterized problems. Actually, Vertex Cover parameterized by the solution size is in FPT but Independent Set parameterized by the solution size is W[1]-complete (it is believed that FPT $\neq$ W[1]). An *odd cycle transversal* of a graph is a vertex subset $T \subseteq V$ such that $G - T$ becomes a bipartite graph. Odd Cycle Transversal is a problem to find a minimum odd cycle transversal.

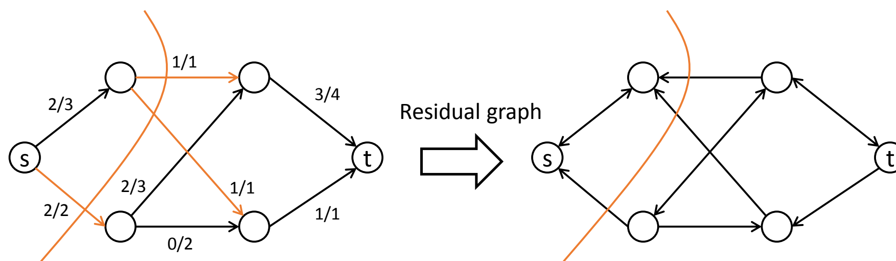

Figure 2.1: An example of maximum flow

Directed Graphs and Networks

Let $G = (V, E)$ be a directed graph. For a vertex subset $S \subseteq V$, we denote the *out-neighbors* of S by $N^+(S) = \{v \in V \setminus S \mid \exists u \in S, uv \in E\}$. For a vertex subset $S \subseteq V$, we denote the edges outgoing from S by $\delta^+(S)$ and the edges incoming to S by $\delta^-(S)$. When S is a single-element set $\{v\}$, we write $\delta^+(v)$ and $\delta^-(v)$, respectively. A vertex set $S \subseteq V$ is called *closed* if $\delta^+(S)$ is an empty set. A vertex set $S \subseteq V$ is called *strongly connected* if for any two vertices $u, v \in S$, there is an directed path from u to v in S. It is known that we can compute strongly connected components in $O(|V| + |E|)$ time [96]. We call a strongly connected component by an *scc* for short.

A *network* is a pair (G, c) of a directed graph $G = (V, E)$ and a capacity function $c : E \to \mathbb{R}_{\geq 0}$. For $s, t \in V$, an *s-t flow* of amount M is a function $f : E \to \mathbb{R}_{\geq 0}$ that satisfies $f(e) \leq c(e)$ for any $e \in E$ and

$$f(\delta^+(v)) - f(\delta^-(v)) = \begin{cases} M & \text{for } v = s, \\ -M & \text{for } v = t, \\ 0 & \text{for any } v \in V \setminus \{s, t\}. \end{cases}$$

A *maximum s-t flow* is an s-t flow with the maximum possible amount. For convenience, we define $c(e) = f(e) = 0$ if $e \notin E$. The vertex s is called the *source* and the vertex t is called the *sink*. A vertex subset S is called an s-t cut if $s \in S$ and $t \notin S$, and its *capacity* is defined as $c(S) = c(\delta^+(S))$. A *minimum s-t cut* is an s-t cut with the minimum possible capacity. The amount of the maximum s-t flow and the capacity of the minimum s-t cut coincide (*max-flow min-cut theorem*) and a minimum cut can be computed in linear time from a maximum flow. The *residual graph* of a network (G, c) with respect to a flow f is the directed graph $G_f = (V, E_f)$ with $E_f = \{(u, v) \mid f(u, v) < c(u, v) \text{ or } f(v, u) > 0\}$. An s-t path in the residual graph is called an *augmenting path*. A flow f is the maximum flow if and only if there are no augmenting paths. Figure 2.1 illustrates an example, where numbers f/c on each edge denote the value of flow f and the capacity c.

CNFs

Let x be a Boolean variable. We denote the negation of x by $\overline{x}$. A *literal* is either a variable or its negation, and a *clause* is a disjunction of several literals $l_1, \ldots, l_k$, where k is called the *length* of the clause. We call a clause of length k a *k-clause*. A *CNF* is a conjunction of clauses and we often write a CNF as a set of clauses. If all the clauses have length at most k, it is called a *k-CNF*. We say that a CNF on a variable set X is *satisfiable* if there is an assignment to X that makes the CNF true. (k-)SAT is a problem in which, given a (k-)CNF $\mathcal{C}$, the objective is to determine whether $\mathcal{C}$ is satisfiable or not. #(k-)SAT is a

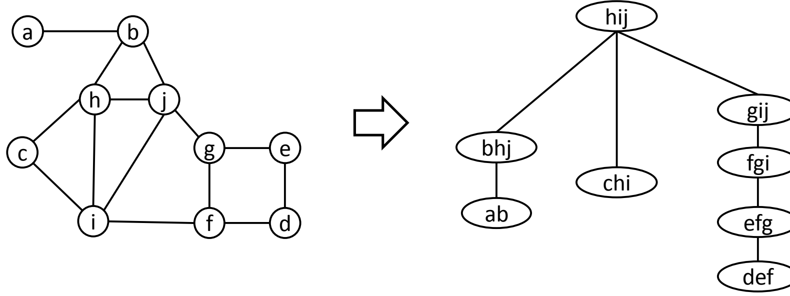

Figure 2.2: An example of tree-decomposition

problem of counting the number of satisfying assignments of $\mathcal{C}$. Max (k-)SAT is a problem in which, given a (k-)CNF $\mathcal{C}$, the objective is to find a maximum satisfiable clause subset $\mathcal{C}' \subseteq \mathcal{C}$. Almost 2-SAT is a problem in which, given a 2-CNF $\mathcal{C}$, the objective is to find a minimum clause subset $\mathcal{C}' \subseteq \mathcal{C}$ such that the CNF $\mathcal{C} \setminus \mathcal{C}'$ becomes satisfiable. We note that Max 2-SAT and Almost 2-SAT are different problems when parameterized by the solution size.

2.2 Tree-decomposition

A *tree-decomposition* of a graph $G = (V, E)$ is a pair (T, χ), where $T = (I, F)$ is a tree and $\chi = \{X_i \subseteq V \mid i \in I\}$ is a collection of subsets of vertices (called *bags*), with the following properties:

1. $\bigcup_i X_i = V$.
2. For each edge $uv \in E$, there exists a bag that contains both of u and v.
3. For each vertex $v \in V$, the bags containing v form a connected subtree in T.

In order to avoid confusion between a graph and its decomposition tree T, we call a vertex of the tree a *node*, and an edge of the tree an *arc*. We identify a node $i \in I$ of the tree and the corresponding bag X_i. The *width* of a tree-decomposition is the maximum of $|X_i| - 1$ over all nodes $i \in I$. The *tree-width* of a graph G, $\mathbf{tw}(G)$, is the minimum width among all the possible tree-decompositions of G. Figure 2.2 illustrates an example of tree-decomposition.

Intuitively, tree-width measures tree-like-ness of graphs. The tree-width of a forest is at most one. If a graph contains a clique, from the definition of tree-decomposition, there must be a bag containing all the vertices of the clique. Thus, the tree-width is at least the size of the clique minus one.

A *nice tree-decomposition* is a tree decomposition such that the root bag X_r is an empty set and each node i is one of the following types:

1. Leaf: a leaf node with $X_i = \emptyset$.
2. Introduce(v): a node with one child c such that $X_i = X_c \cup \{v\}$ and $v \notin X_c$.
3. Introduce(uv): a node with one child c such that $u, v \in X_i = X_c$. We require that this node appears exactly once for each edge uv of G.
4. Forget(v): a node with one child c such that $X_i = X_c \setminus \{v\}$ and $v \in X_c$. From the definition of tree-decompositions, this node appears exactly once for each vertex of G.
5. Join: a node with two children l and r with $X_i = X_l = X_r$.

Any tree-decomposition can be easily converted into a nice tree-decomposition of the same width in polynomial time by inserting intermediate bags between each adjacent bags. Thus, in this thesis, we use nice tree-decompositions to make discussions simple.

A (nice) *path-decomposition* is a (nice) tree-decomposition (T, χ) such that the decomposition tree $T = (I, F)$ is a path. For convenience, we assume that $I = [N]$ for some integer N and $F = \{(i, i+1) \mid i \in [N-1]\}$. The *path-width* of a graph G, $\mathbf{pw}(G)$, is the minimum width among all the possible path-decompositions of G.

Let $\mathcal{C}$ be a CNF on variables X. The *primal graph* of $\mathcal{C}$ is the graph $G = (X, E)$ such that there exists an edge between two vertices if and only if their corresponding variables appear in the same clause. For readability, we identify a variable or a literal as the corresponding vertex in the primal graph. That is, we may use the same symbol x to indicate both a variable in a CNF and the corresponding vertex in the primal graph, and both literals x and $\overline{x}$ correspond to the identical vertex in the primal graph. By a decomposition of a CNF, we mean a decomposition of its primal graph. For a CNF $\mathcal{C}$, we slightly change the definition of the nice tree-decomposition as follows:

3′. Introduce(C): an internal node with one child c such that $X_i = X_c$ and all the variables in C are in X_i. We require that this node appears exactly once for each clause $C \in \mathcal{C}$.

Note that because the variables in the same clause form a clique in the primal graph, there always exists a bag that contains all of them. Thus, we can always construct a nice tree-decomposition of the same width in polynomial time.

Finding Tree-decompositions

It is NP-complete to determine whether a given graph has tree-width at most a given value w [8]. Although there exists linear-time algorithm for every fixed w [19] (i.e., linear-time FPT), its dependency on w is very huge ($2^{O(w^3)}$). There exist two types of approximation algorithms: a polynomial-time approximation algorithm which computes a tree-decomposition of width $O(\mathbf{tw}\sqrt{\log \mathbf{tw}})$ in polynomial time [37], and an FPT approximation algorithm which computes a tree-decomposition of width $5\mathbf{tw} + 4$ in $2^{O(\mathbf{tw})}m$ time [20], where $\mathbf{tw}$ is the (optimal) tree-width of the input graph. In practice, heuristic methods, such as min-degree heuristic [16], are often used to compute tree-decompositions.

2.3 Parameterized Complexity and SETH

A *parameterized problem* is a language $L \subseteq \Sigma^* \times \mathbb{N}$, where Σ is a fixed finite alphabet. For an instance $(x, k) \in \Sigma^* \times \mathbb{N}$, k is called the *parameter*, and the size of the instance is defined as $|(x, k)| = |x| + k$. A parameterized problem L is called *fixed-parameter tractable (FPT)* if there exists an algorithm $\mathcal{A}$ and a computable function $f : \mathbb{N} \to \mathbb{N}$ such that, for any instance $(x, k) \in \Sigma^* \times \mathbb{N}$, $\mathcal{A}$ correctly decides whether $(x, k) \in L$ in $f(k)|(x, k)|^{O(1)}$ time. Such an algorithm $\mathcal{A}$ is called an FPT algorithm. An FPT algorithm runs in polynomial time for any constant value k, and moreover, the degree of the polynomial does not depends on k. If an algorithm $\mathcal{A}$ runs in linear time for any constant k (i.e., in time $f(k)|(x, k)|$), it is called a *linear-time FPT* algorithm. We often use the O^* notation which suppresses a factor polynomial in the input size (e.g., $O(2^k n^3)$ is denoted by $O^*(2^k)$).

We will give several examples of parameterized problems. The most basic parameter is the solution size for optimization problems. For example, when parameterized by the solution size, Vertex Cover becomes the following parameterized problem.

Vertex Cover	**Parameter:** k
Input: An undirected graph G and an integer k.	
Question: Is there a vertex cover of G whose size is at most k?	

We note that by iteratively incrementing the parameter value k, an algorithm for Vertex Cover parameterized by the solution size can compute the size of the minimum vertex cover. Moreover, if the running time of the algorithm is exponential in k (e.g., $O(2^k m)$), such an iteration does not affect the total running time because it holds that $\sum_{i=0}^{k} c^i = O(c^k)$.

In this thesis, parameters called *above relaxation lower bound* are often used. For example, when parameterized above LP lower bound, Vertex Cover becomes the following parameterized problem.

Vertex Cover above LP	**Parameter:** k'
Input: An undirected graph G and an integer k'.	
Question: Is there a vertex cover of G whose size is at most the (standard) LP relaxation lower bound plus k'?	

Other basic parameters are *width* of the input graph. For example, when parameterized by the tree-width, Vertex Cover becomes the following parameterized problem.

Vertex Cover parameterized by tree-width	**Parameter:** w
Input: An undirected graph G, an integer k, and a tree-decomposition of G of width w.	
Question: Is there a vertex cover of G whose size is at most k?	

Here, we note that, instead of parameterizing problems by the minimum width of the input graph, we assume that a decomposition of the input graph, which may not be optimal, is given as a part of the input, and problems are parameterized by the width of the given decomposition.

For an integer $k \geq 3$, let s_k be the infimum of the real numbers δ such that k-SAT can be solved in $O^*(2^{\delta n})$ time, where n is the number of variables. The *Exponential-Time Hypothesis (ETH)* and the *String Exponential-Time Hypothesis (SETH)* are defined as follows.

Conjecture 2.1 (ETH [53])**.**

$$s_3 > 0$$

Conjecture 2.2 (SETH [53])**.**

$$\lim_{k \to \infty} s_k = 1$$

Impagliazzo and Paturi [53] proved that SETH implies ETH. SETH implies that SAT cannot be solved in $(2-\epsilon)^n$ time for any $\epsilon > 0$, and we often use the implied lower bound of SAT for proving hardness under SETH. Note that the converse may not hold (e.g., there might be an algorithm for k-SAT that runs in $O^*(k^k 1.99^n)$ time).

Chapter 3

Linear-time FPT Branch-and-Bound via Network Flow

In this chapter, we give an FPT branch-and-bound algorithm for VERTEX COVER ABOVE LP that runs in $O(4^{k'}(n+m)+T)$ time, where the parameter k' is the difference between the solution size and the LP lower bound and T is a time complexity for computing the initial solution of the LP relaxation. Using this algorithm, we can solve ODD CYCLE TRANSVERSAL and ALMOST 2-SAT in $O(4^k(n+m))$ time, where the parameter k is the solution size. This is the first linear-time FPT algorithm for these problems. First, in Section 3.1, we review the existing algorithms for VERTEX COVER including the $O^*(4^{k'})$-time FPT algorithm by Lokshtanov et al.[71] Then, we improve it to $O(4^{k'}(n+m)+T)$ time in Section 3.2. The algorithm in this chapter is used in the empirical research in Chapter 4 and more general version of the algorithm is described in Section 5.4 of Chapter 5.

3.1 Review of the Existing Algorithms

3.1.1 LP Relaxation of Vertex Cover

The LP relaxation of VERTEX COVER can be written as follows:

$$\begin{aligned} \text{minimize} & \sum_{v\in V} x_v \\ \text{s.t.} \quad & x_u + x_v \geq 1 \quad \forall uv \in E, \\ & x_v \geq 0 \quad\quad\quad \forall v \in V. \end{aligned}$$

Nemhauser and Trotter [81] showed that the above LP has the following two properties:

- There exists an optimal solution such that each variable takes a value 0, 1, or $\frac{1}{2}$ (*Half-integrality*).
- If a variable x_v takes an integer value in an optimal LP solution, there always exists an optimal integer solution in which x_v takes the same value (*Persistency*).

We note that, from the persistency, we can *simultaneously* fix all the variables with integral values.

Algorithm 1 $O^*(4^{k'})$-time algorithm for VERTEX COVER ABOVE LP [71]

INPUT: a graph G and an upper bound k
OUTPUT: whether the graph contains a vertex cover of size at most k

```
1: procedure SOLVE(G, k)
2:     if LP(G) > k then return false
3:     if V = ∅ then return true
4:     pick an arbitrary vertex v ∈ V
5:     if 1 + LP(G − v) = LP(G) then                    ▷ Try to fix x_v to 1
6:         return SOLVE(G − v, k − 1)
7:     else if d(v) + LP(G − N[v]) = LP(G) then          ▷ Try to fix x_v to 0
8:         return SOLVE(G − N[v], k − d(v))
9:     else                                              ▷ Branch into two cases
10:        return SOLVE(G − v, k − 1) ∨ SOLVE(G − N[v], k − d(v))
```

3.1.2 FPT Algorithm for Vertex Cover above LP

Using the properties of the LP relaxation, Lokshtanov et al. [71] obtained a simple $O^*(4^{k'})$-time[1] algorithm for VERTEX COVER ABOVE LP which is described in Algorithm 1. Let $\mathrm{LP}(G)$ denote the LP lower bound for a graph G. Our task is to find a vertex cover of size at most $k = \mathrm{LP}(G) + k'$. In the recursive call of the algorithm, we first check the LP lower bound. If the lower bound $\mathrm{LP}(G)$ is larger than the size k of the solution we want to find, we prune the search (line 2). If the graph becomes empty, we finish the algorithm (line 3). Otherwise, we pick an arbitrary vertex $v \in V$ and try two cases: 1) including v to the vertex cover or 2) discarding v while including its neighbors to the vertex cover. If in at least one of these two cases, the LP lower bound does not increase, we can fix them without branching from the persistency (lines 5–8). Otherwise, we branch into the two cases (line 10). From the half-integrality, the value of the LP lower bound is always a multiple of $\frac{1}{2}$. Thus, the lower bound increases at least $\frac{1}{2}$ in both cases. Therefore, the depth of the search tree is bounded by $2k'$, which leads to the running time of $O^*(2^{2k'}) = O^*(4^{k'})$.

3.1.3 Reduction from Odd Cycle Transversal

Lokshtanov et al. [71] showed that many problems including ODD CYCLE TRANSVERSAL and ALMOST 2-SAT can be reduced to VERTEX COVER ABOVE LP, and therefore, the above algorithm can solve these problems in $O^*(4^k)$ time, where the parameter k is the solution size. Here, we review the (linear-time) reduction from ODD CYCLE TRANSVERSAL.

From the input graph $G = (V, E)$, we construct a graph $G' = (V', E')$ such that:

- $V' = \{v_1 \mid v \in V\} \cup \{v_2 \mid v \in V\}$ and
- $E' = \{u_1v_1 \mid uv \in E\} \cup \{u_2v_2 \mid uv \in E\} \cup \{v_1v_2 \mid v \in V\}$.

Let C be a minimum vertex cover of G'. Then, a minimum odd cycle transversal can be computed by taking vertices $v \in V$ such that both v_1 and v_2 are in C.

Let k be the size of the minimum odd cycle transversal of G. Since at least one of v_1 and v_2 must be contained in a vertex cover of G', the size of the minimum

[1] Using sophisticated reduction rules, they improved the running time to $O^*(2.3146^{k'})$. However, these reduction rules cannot be applied in linear-time. Thus, we use the simple version.

vertex cover of G' is exactly $k+|V|$. Because the all-half vector (i.e., the vector x such that $x_v = \frac{1}{2}$ for any v) is the optimal LP solution to the reduced instance, the optimal LP value is exactly $|V|$. Therefore, the difference k' between the solution size and the LP lower bound of the reduced instance is $(k + |V|) - |V| = k$. Thus, the $O^*(4^{k'})$-time algorithm for VERTEX COVER ABOVE LP can solve ODD CYCLE TRANSVERSAL in $O^*(4^k)$ time.

3.1.4 Solving the LP Relaxation by Network Flow

For computing the optimal LP solution efficiently, we can use the result by Nemhauser and Trotter [81]. They showed that the half-integral optimal solution of the LP relaxation of VERTEX COVER can be computed by solving a minimum s-t cut problem as follows. From the input graph $G = (V, E)$, we construct a network $(\bar{G} = (\bar{V}, \bar{E}), c)$ such that:

- $\bar{V} = \{s, t\} \cup \{l_v \mid v \in V\} \cup \{r_v \mid v \in V\}$,
- $\bar{E} = \{sl_v \mid v \in V\} \cup \{r_v t \mid v \in V\} \cup \{l_u r_v \mid uv \in E\} \cup \{l_v r_u \mid uv \in E\}$,
- $\forall v \in V, c(sl_v) = c(r_v t) = 1$, and $\forall uv \in E, c(l_u r_v) = c(l_v r_u) = \infty$.

Then, the capacity of the minimum s-t cut is exactly the twice of the optimal LP value. Moreover, there is the following correspondence between optimal LP solutions and minimum s-t cuts.

Lemma 3.1 (Nemhauser and Trotter [81])**.** *For any minimum s-t cut S, the following x^* is a half-integral optimal solution:*

$$x^*_v = \begin{cases} 0 & (l_v \in S, r_v \notin S), \\ 1 & (r_v \in S, l_v \notin S), \\ \frac{1}{2} & (\textit{otherwise}). \end{cases}$$

Conversely, for any half-integral optimal LP solution x^*, the following S is a minimum s-t cut:

$$S = \{s\} \cup \{l_v \mid v \in V, x^*_v = 0\} \cup \{r_v \mid v \in V, x^*_v = 1\}.$$

By using Hopcroft-Karp algorithm [50], we can compute the minimum s-t cut of the above network in $O(m\sqrt{n})$ time. Figure 3.1 illustrates an example of the correspondence between optimal LP solutions and minimum s-t cuts; from the graph shown in the left, we can construct the network shown in the right; from the minimum s-t cut S shown as the green-colored vertices, we can obtain the corresponding optimal LP solution $x^* = (\frac{1}{2}, \frac{1}{2}, 1, 0, 0)$.

3.2 Linear-time FPT Algorithm

3.2.1 Algorithm Overview

When trying to improve the $n^{O(1)}$ part of the $O^*(4^{k'})$-time FPT algorithm presented in Subsection 3.1.2 to linear time ($O(n + m)$), there are two obstacles. First, we need to compute the LP lower bound in linear time. Although by using the network-flow-based algorithm presented in Subsection 3.1.4, we can solve the LP relaxation in $O(m\sqrt{n})$ time, this is not enough to achieve linear time. Our solution is that, instead of computing a minimum cut from scratch, we keep and update a maximum flow in linear time. Although, we still need to compute the

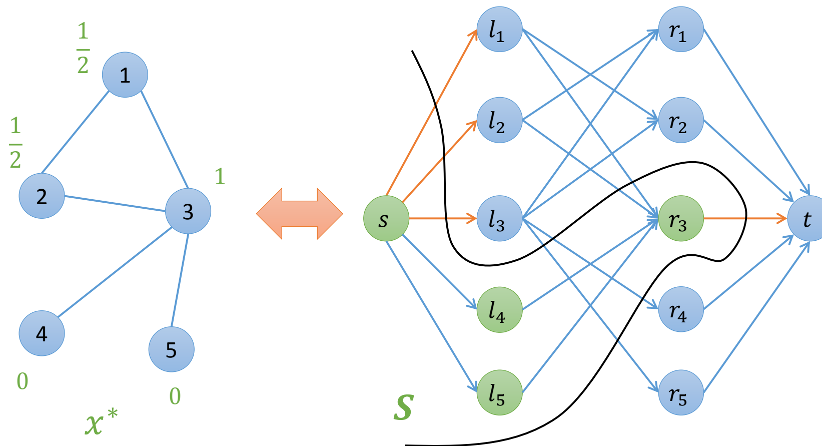

Figure 3.1: An example of the correspondence between optimal LP solutions and minimum s-t cuts

initial maximum flow, which cannot be computed in linear time in general, for many applications such as instances reduced from Odd Cycle Transversal and Almost 2-SAT, we can obtain it in linear time.

The second obstacle is that, in order to avoid the repeat of lines 5–8 of Algorithm 1, we need to find a vertex v such that the LP lower bound strictly increases in both cases of fixing x_v to zero and one in linear time (otherwise, on each branching step, we may need to check up to $O(|V|)$ vertices, which leads to $\Omega(nm)$ time). From the persistency property, we can fix the integral part of the optimal LP solution. However, this does not imply that the branching on a vertex v with $x^*_v = \frac{1}{2}$ always increases the LP lower bound. This is because the optimal LP solutions may not be unique and there might be another optimal LP solution such that x^*_v takes an integer value (e.g., in the example shown in Figure 3.1, the branching on the vertex 1 does not increase the LP lower bound because $(1, 0, 1, 0, 0)$ and $(0, 1, 1, 0, 0)$ are also optimal solutions). Our solution is that, instead of computing an arbitrary half-integral optimal solution, we compute a half-integral optimal solution whose half-integral part is minimal. We call such an optimal solution as *extreme* optimal solution. If we fix the integral part of an extreme optimal solution, the all-half vector becomes the unique optimal solution for the remaining graph. Thus, branching on any vertex always increases the LP lower bound.

An overview of our algorithm is described in Algorithm 2. During the recursion, we keep and update the network and a maximum flow. First, we compute an extreme optimal solution x^* in linear time using the maximum flow (line 2). Then, we fix the integral part of x^* and update the network and the maximum flow in linear time (line 3). Now, the all-half vector becomes the unique optimal solution for the remaining graph. If the LP lower bound (which is $\frac{|V|}{2}$) is larger than the size k of the solution we want to find, we prune the search (line 4). If the graph becomes empty, we finish the algorithm (line 5). Otherwise, we pick an arbitrary vertex v and branch into the two cases 1) including v to the vertex cover or 2) discarding v while including its neighbors to the vertex cover. In each case, we update the network and the maximum flow in linear time (line 7).

Algorithm 2 $O(4^{k'}(n+m))$-time algorithm for VERTEX COVER ABOVE LP

INPUT: a graph G, the corresponding network $\bar{G}$, a maximum flow f, and an upper bound k

OUTPUT: whether the graph contains a vertex cover of size at most k

1: **procedure** SOLVE($G, \bar{G}, f, k$)
2: $x^* \leftarrow$ EXTREMEOPT($\bar{G}, f$) ▷ Compute an extreme optimal solution
3: $(G, \bar{G}, f, k) \leftarrow$ REDUCE($G, \bar{G}, f, k, x^*$) ▷ Fix the integral part
4: **if** $\frac{|V|}{2} > k$ **then return** false
5: **if** $V = \emptyset$ **then return** true
6: pick an arbitrary vertex $v \in V$
7: **return** SOLVE($G - v, \bar{G}', f', k-1$) $\vee$ SOLVE($G - N[v], \bar{G}'', f'', k - d(v)$)

3.2.2 Updating the Maximum Flow

Now, we explain how to update the maximum flow in linear time. There are two cases: when fixing the integral part (line 3) and when branching into two cases (line 7). Here we note that the branching on a vertex v can also be implemented as fixing the variable x_v to zero and one.

From the correspondence between minimum cuts and optimal LP solutions (Lemma 3.1), fixing x_v to zero corresponds to identifying l_v as the source s and r_v as the sink t (or equivalently, adding edges sl_v and r_vt of the infinite capacity), and fixing x_v to one corresponds to identifying r_v as s and l_v as t (or equivalently, adding edges sr_v and l_vt of the infinite capacity). Thus the maximum flow remains a flow (which may not be the maximum) after the fixing.

Fixing the integral part (line 3) does not change the optimal LP solution. Thus, the flow after the fixing is still the maximum flow. On the other hand, when we branch into two cases (line 7), the optimal LP value increases. Thus, we need to update the flow to the maximum flow. This can be done by searching augmenting paths. Let d be the increase of the optimal LP value (when fixing x_v to one, the optimal LP value increases only by $\frac{1}{2}$ but when fixing x_v to zero, the optimal LP value may increase more than $\frac{1}{2}$). Then, we can update the flow by searching an augmenting path $2d$ times, which can be done in $O(d(n+m))$ time. Let $T(k')$ be the time complexity for solving the problem when the difference between the solution size and the LP lower bound is k'. Then, we obtain the recurrences $T(k') \leq T(k' - \frac{1}{2}) + T(k' - d) + O(d(n+m))$ for $\frac{1}{2} \leq d \leq k'$. Here, we note that d is upper bounded by k' because when we find more than $2k'$ augmenting paths, the LP lower bound exceeds the size of the solution we want to find and we can immediately prune the search without finishing the update of the maximum flow. The worst case is achieved when $d = \frac{1}{2}$ and we obtain the time complexity of $O(4^{k'}(n+m))$.

3.2.3 Computing the Extreme Optimal Solution

In order to describe the linear-time algorithm for computing an extreme optimal solution, we need several definitions. We say that a half-integral optimal solution x^* is an *extreme* optimal solution if there is no half-integral optimal solution y^* distinct from x^* such that for any $v \in V$ it holds that $x^*_v \neq \frac{1}{2} \Rightarrow x^*_v = y^*_v$. From the definition, after fixing the integral part of an extreme optimal solution, the all-half vector becomes the unique half-integral optimal solution. If an s-t cut contains at most one of l_v and r_v for each $v \in \bar{V}$, it is called *normalized*. As shown in Lemma 3.1, there is the one-to-one correspondence between the half-integral

Algorithm 3 Algorithm to compute an extreme minimum cut.

INPUT: the residual graph
OUTPUT: an extreme minimum cut

1: compute the strongly connected components
2: $S \leftarrow$ the vertices reachable from s
3: **while** $\exists$ unchecked scc T such that $N^+(T) \subseteq S$ **do**
4:     **if** $S \cup T$ is a normalized cut **then**
5:         $S \leftarrow (S \cup T)$
6: **return** S

optimal solutions and the normalized minimum cuts. A normalized minimum cut S is called an *extreme* minimum cut if there is no normalized minimum cut T such that $S \subset T$. We use the same adjective "*extreme*" for half-integral optimal solutions and normalized minimum cuts because there is the following one-to-one correspondence.

Lemma 3.2. *Let x^* be a half-integral optimal solution and S be the corresponding normalized minimum cut. Then, x^* is an extreme optimal solution if and only if S is an extreme minimum cut.*

Proof. ($\Rightarrow$) Assume that there exists a normalized minimum cut $T \supset S$. From T, we construct the corresponding half-integral optimal solution y^* using Lemma 3.1. For each v with $x_v^* \neq \frac{1}{2}$, S contains exactly one of l_v and r_v, and T can contain at most one of l_v and r_v. Thus, the same vertex must be contained in T for such v. Then, from the construction of y^*, it holds that $x_v^* \neq \frac{1}{2} \Rightarrow x_v^* = y_v^*$. Thus, x^* is not an extreme optimal solution, which is a contradiction.

($\Leftarrow$) Assume that there exists a half-integral optimal solution y^* distinct from x^* such that for any $v \in V$ it holds that $x_v^* \neq \frac{1}{2} \Rightarrow x_v^* = y_v^*$. From y^*, we construct the corresponding normalized minimum cut T using Lemma 3.1. From the construction, it holds that $S \subset T$. Thus, S is not an extreme minimum cut, which is a contradiction. $\square$

From the above correspondence, in order to compute an extreme optimal solution, it suffices to compute an extreme minimum cut. In order to compute an extreme minimum cut, we introduce the following one-to-one correspondence between the minimum s-t cut and the closed vertex set of the residual graph.

Lemma 3.3 (Picard and Queyranne [85])**.** *For any network, its two vertices s and t, and its maximum s-t flow f, an s-t cut S is a minimum cut if and only if S is a closed set in the residual graph with respect to f.*

Note that a maximum s-t flow in the lemma is arbitrary. This lemma reveals a nice structure of the all minimum cuts: although there exist exponentially many minimum cuts in a network, we can find an extreme one in linear-time as described in Algorithm 3.

First, we compute the strongly connected components of the residual graph G_f. From Lemma 3.3, for each strongly connected component T, any minimum cut must contain all of T or none of T. Then we compute the vertex set S reachable from s in G_f. Since this is a closed set containing s, it is a minimum cut. Suppose that S is not a normalized cut. From Lemma 3.3, any minimum cut must completely contain S. Therefore, there exists no normalized minimum cut, which is a contradiction. Thus, S is a normalized minimum cut. From now on, we modify S to be an extreme minimum cut by expanding it. Let $T \subseteq V \setminus S$ be a strongly connected component that satisfies the following two conditions:

1. All the outgoing edges from T are coming into S.
2. The cut $S \cup T$ is normalized.

If there exists a strongly connected component T that satisfies the first condition, the cut $S \cup T$ also becomes a closed set. Thus it is a minimum cut. If there exists T that satisfies both of the conditions, we can obtain a new normalized minimum cut by expanding S to $S \cup T$. If there are no such T, S is an extreme cut. This is because any minimum cut $S' \supset S$ must contain at least one of the strongly connected components that satisfy the condition 1, but including any of them does not lead to a normalized cut.

Figures 3.2 and 3.3 illustrate an example of the execution of the algorithm. From the graph shown in the left, we construct the network shown in the right. A maximum flow of the network is shown as red-colored edges. The residual graph consists of the following five sccs: $\{s\}$, $\{l_6, l_7, r_4, r_5\}$, $\{l_1, l_2, l_3, r_1, r_2, r_3\}$, $\{l_4, l_5, r_6, r_7\}$, and $\{t\}$. The initial cut (the vertices reachable from s) is the set $S_0 = \{s\}$, and the corresponding optimal solution is the all-half vector (Figure 3.2). Then, the algorithm picks the scc $T_1 = \{l_6, l_7, r_4, r_5\}$ satisfying $N^+(T_1) \subseteq S_0$. Since $S_0 \cup T_1$ is a normalized cut, we expand the cut to $S_1 = S_0 \cup T_1 = \{s, l_6, l_7, r_4, r_5\}$, and the corresponding optimal solution changes to $(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, 1, 1, 0, 0)$ (Figure 3.3). Then, the algorithm picks the scc $T_2 = \{l_1, l_2, l_3, r_1, r_2, r_3\}$ satisfying $N^+(T_2) \subseteq S_1$. Since $S_1 \cup T_2$ is not a normalized cut (because it contains both of l_1 and r_1), we skip it. There are no other sccs satisfying the condition $N^+(T) \subseteq S_1$. Thus, the algorithm finishes. The vector $(\frac{1}{2}, \frac{1}{2}, \frac{1}{2})$ is actually the unique optimal LP solution for the subgraph on the vertices $\{1, 2, 3\}$.

Finally, we analyze the running time of the algorithm. We can compute the strongly connected components in $O(|V| + |E|)$ time. In order to efficiently find a strongly connected component that satisfies the condition 1, for each strongly connected component T, we keep track of the number of edges outgoing from T to the vertices outside S. If this number is zero, it satisfies the condition 1. When updating S to $S \cup T$, for each edge $uv \in \delta^-(T)$, we decrement the number for the strongly connected component that contains u. This takes only $O(|\delta^-(T)|)$ time for each T. Thus it takes only $O(|E|)$ time in total. If a strongly connected component T does not satisfy the condition 2 for some S, it will never satisfy the condition for any $S' \supset S$. Therefore, we don't have to check the same strongly connected component multiple times. Thus the total running time is $O(|V|+|E|)$.

3.2.4 Computing the Initial Maximum Flow

Algorithm 2 requires the initial maximum flow. In general, we cannot compute it in linear time and it takes $O(m\sqrt{n})$ time by using Hopcroft-Karp algorithm [50]. However, for many applications such as instances reduced from Odd Cycle Transversal and Almost 2-SAT, we can obtain it in linear time. In this subsection, we describe how to obtain the initial maximum flow for Odd Cycle Transversal in linear time. We omit the case for Almost 2-SAT here, but by using more general result described in Chapter 5, we can solve it in $O(4^k(n+m))$ time without using a reduction to Vertex Cover above LP.

Let $G = (V, E)$ be an instance of Odd Cycle Transversal. We reduce it to Vertex Cover as described in Subsection 3.1.3 and obtain the graph $G' = (V', E')$. A maximum flow f for the network $\bar{G}'$ can be constructed as follows:

- $f(sl_{v_1}) = f(l_{v_1} r_{v_2}) = f(r_{v_2} t) = 1$ for any $v \in V$, and

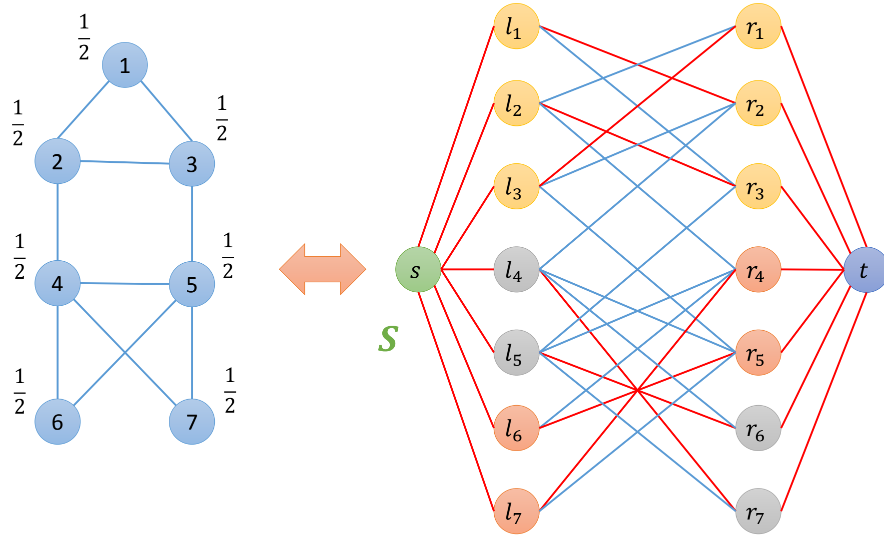

Figure 3.2: The optimal LP solution corresponding to the cut $\{s\}$

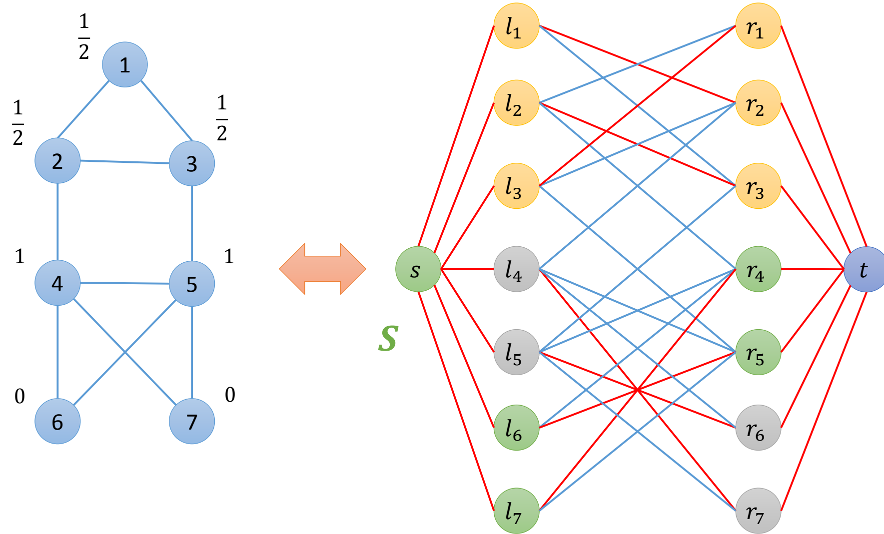

Figure 3.3: The optimal LP solution corresponding to the cut $\{s, l_6, l_7, r_4, r_5\}$

- $f(sl_{v_2}) = f(l_{v_2}r_{v_1}) = f(r_{v_1}t) = 1$ for any $v \in V$.

Since the amount of the flow f is $2|V|$ and the optimal LP value for G' is $|V|$, the flow f is the maximum flow.

Chapter 4

Branch-and-Reduce Algorithms in Practice

In this chapter, for showing a practical impact of theoretical research on branching algorithms, we propose an algorithm for Vertex Cover which involves many techniques from theoretical studies and give an experimental evaluation. First, in Section 4.1, we describe the overview of our algorithm. In Sections 4.2, 4.3, and 4.4, we give a list of the branching rules, the reduction rules, and the lower bounds used in our algorithms, respectively. We investigate the parameterized complexity above these lower bounds in Section 4.5. Finally, we explain our experimental results in Section 4.6.

4.1 Algorithm Overview

The overview of our algorithm is described in Algorithm 4. For ease of presentation, the described algorithm only addresses the size of the minimum vertex cover. However, obtaining the minimum vertex cover itself is not difficult. Indeed, in our experiments, the minimum vertex cover is also computed, and the time consumption to accomplish this is also accumulated. The *packing constraints* in the algorithm are created by our new branching and reduction rules. They are not used to strengthen the LP relaxation, as in the branch-and-cut methods, but are used for the pruning and the reduction. We describe the details in Subsections 4.2.4 and 4.3.4. We start the algorithm by setting $\mathcal{P} = \emptyset$, $c = 0$, and $k = |V|$. On each branching node, we first apply a list of reduction rules. Then, we prune the search if the packing constraints are not satisfied or if the lower bound is at least the size of the best solution we have. If the graph is empty, we update the best solution. If the graph is not connected, we separately solve each connected component. Otherwise, we branch into two cases by applying the branching rule. In our implementation, for time and space efficiency, we do not create new graphs after the branching but dynamically modify a single graph.

4.2 Branching Rules

4.2.1 Vertex Selection

In our main implementation, we completely use the same strategy as the one used in the theoretical exact exponential algorithm by Fomin et al. [40] for selecting a vertex to branch on. Basically, a vertex of the maximum degree is selected. If there are multiple possibilities, we choose the vertex v that minimizes the number of edges among $N(v)$. In our experiments (Subsection 4.6.3), we compare this strategy to the random selection strategy and the minimum degree selection strategy.

Algorithm 4 The branch-and-reduce algorithm for VERTEX COVER

INPUT: a graph G, packing constraints $\mathcal{P}$, a current solution size c, and an upper bound k

1: **procedure** SOLVE($G, \mathcal{P}, C, k$)
2: $(G, \mathcal{P}, c) \leftarrow$ REDUCE($G, \mathcal{P}, c$)
3: **if** UNSATISFIED($\mathcal{P}$) **then return** k
4: **if** $c +$ LOWERBOUND(G) $\geq k$ **then return** k
5: **if** G is empty **then return** c
6: **if** G is not connected **then**
7: **for all** $(G_i, \mathcal{P}_i) \in$ COMPONENTS($G, \mathcal{P}$) **do**
8: $c \leftarrow c +$ SOLVE($G_i, \mathcal{P}_i, 0, k - c$)
9: **return** $\min(k, c)$
10: $((G_1, \mathcal{P}_1, c_1), (G_2, \mathcal{P}_2, c_2)) \leftarrow$ BRANCH($G, \mathcal{P}, c$)
11: $k \leftarrow$ SOLVE($G_1, \mathcal{P}_1, c_1, k$) ▷ Updating the upper bound k by trying the first case.
12: $k \leftarrow$ SOLVE($G_2, \mathcal{P}_2, c_2, k$) ▷ The updated k is used to bound the search space for the second case.
13: **return** k ▷ k has been updated by the minimum of the two cases.

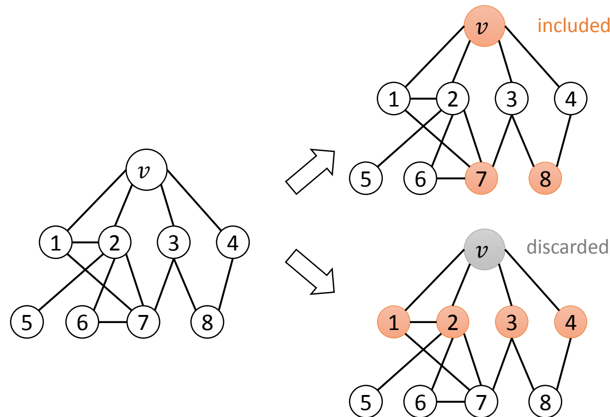

Figure 4.1: An example of the mirror branching

4.2.2 Mirror Branching

For a vertex v, a vertex $u \in N^2(v)$ is called a *mirror* of v if $N(v) \setminus N(u)$ induces a clique or is an empty set. We denote the set of mirrors for v by $\mathcal{M}(v)$ and use the notation of $\mathcal{M}[v] = \mathcal{M}(v) \cup \{v\}$. For the *mirror branching* rule by Fomin et al. [40], we branch into two cases: 1) including $\mathcal{M}[v]$ to the vertex cover or 2) discarding v while including $N(v)$ to the vertex cover. In our implementation, we use this branching rule when the selected vertex v has mirrors. Figure 4.1 illustrates an example of the mirror branching, where the mirrors of the vertex v are $\mathcal{M}(v) = \{7, 8\}$.

4.2.3 Satellite Branching

For a vertex v, a vertex $u \in N^2(v)$ is called a *satellite* of v if there exists a vertex $w \in N(v)$ such that $N(w) \setminus N[v] = \{u\}$. We denote the set of satellites for v by $\mathcal{S}(v)$ and use the notation of $\mathcal{S}[v] = \mathcal{S}(v) \cup \{v\}$. Kneis et al. [64] introduced the following *satellite branching* rule for the case in which there are no mirrors: 1)

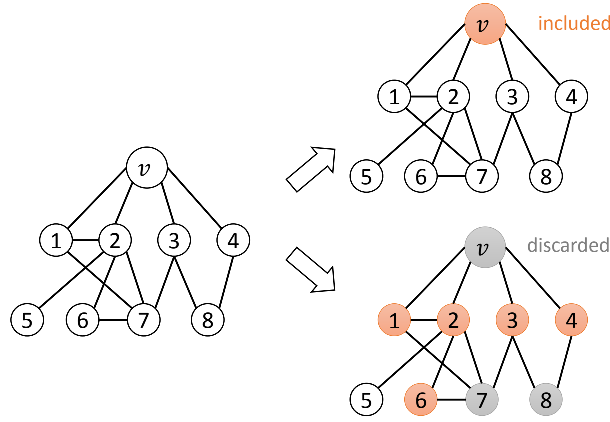

Figure 4.2: An example of the satellite branching

including v to the vertex cover or 2) discarding $\mathcal{S}(v)$ while including $N(\mathcal{S}(v))$ to the vertex cover. In our implementation, instead of using this branching rule, we use a more powerful branching rule introduced in the next subsection. Figure 4.2 illustrates an example of the satellite branching, where the satellites of the vertex v are $\mathcal{S}(v) = \{7, 8\}$.

4.2.4 Packing Branching

Let v be the selected vertex to branch on. The proof outline for the correctness of the satellite branching is as follows. If there exists a minimum vertex cover of G that contains the vertex v, we can find it by searching for a minimum vertex cover of $G - v$. Otherwise, we can assume that no minimum vertex covers contain the vertex v. If there exists a minimum vertex cover C of G that does not contain the vertex v but contains a satellite $u \in \mathcal{S}(v)$, then by discarding the vertex $w \in N(v)$ that satisfies $N(w) \setminus N[v] = \{u\}$ from C and including v to C, we obtain a vertex cover that contains the vertex v of the same size, which is a contradiction.

The key idea of satellite branching is that during the search for a minimum vertex cover that does not contain the vertex v, we can assume that there are no minimum vertex covers that contain the vertex v. To avoid the search of vertex covers from which we can confirm the existence of a vertex cover of the same size containing v, we exploit this idea by explicitly creating constraints as follows.

For a vertex $w \in N(v)$, let $N^+(w) = N(w) \setminus N[v]$. During the search for a minimum vertex cover that does not contain v, if we include all the vertices of $N^+(w)$ to the vertex cover, by discarding the vertex w and including the vertex v, we can obtain a vertex cover of the same size. Thus, in the search for a minimum vertex cover that does not contain v, for each vertex $w \in N(v)$, we can introduce a constraint of $\sum_{u \in N^+(w)} x_u \leq |N^+(w)| - 1$, where x_u is a variable that indicates whether the vertex u is in the vertex cover (1) or not (0). We call these constraints *packing constraints*. We keep and manage the constraints during the search; when we include a vertex v to the vertex cover, for each constraint that contains the variable x_v, we delete the variable and decrease the right-hand side by one, and when we delete a vertex v from the graph without including it to the vertex cover, for each constraint that contains the variable x_v, we delete the variable while keeping its right-hand side. When some constraint is not satisfied at some node, i.e., the right-hand side of the constraint becomes negative, we prune the subsequent search from the node. We note that without packing constraints, we can prune the search only when the graph becomes empty or when the lower

bound exceeds the best solution we have found so far.

We can also introduce a packing constraint when we search for a minimum vertex cover that contains a vertex v. If all the neighbors of v are contained in the vertex cover, by discarding v, we can obtain a vertex cover of smaller size. Thus, in the search for a minimum vertex cover that contains v, we can introduce a constraint of $\sum_{u \in N(v)} x_u \leq |N(v)| - 1$.

Moreover, we can also use packing constraints for reductions. If the right-hand side of a constraint becomes zero but the left-hand side contains a variable x_u, we can delete the vertex u from the graph while including its neighbors $N(u)$ to the vertex cover. The satellite branching corresponds to the case that $N^+(w)$ is a single vertex set. In Subsection 4.3.4, we introduce more sophisticated reduction rules to exploit packing constraints.

We note that the total size of packing constraints scales at most linearly with the graph size because we create at most one constraint for each vertex w and the size of each constraint is at most the degree of the corresponding vertex. Thus explicitly keeping all the constraints does not seriously affect the computation time. We also note that packing constraints are auxiliary; i.e., our objective is not to search for a minimum vertex cover under the constraints but to search for a minimum vertex cover or conclude that there exists a minimum vertex cover not satisfying the constraints (which can be found in another case of the branching).

4.3 Reduction Rules

We use various reduction rules from theoretical branching algorithms. We denote the set of all minimum vertex covers of G by $\mathrm{vc}(G)$.

4.3.1 Reductions from Exponential Algorithms

First, we introduce four reduction rules from the exact exponential algorithm by Fomin et al. [40]. Three of them are quite simple. The first one is the *components* rule. When a graph is not connected, we can solve for each component separately. The second one is the *degree-1* rule. If a graph contains a vertex of degree at most one, there always exists a minimum vertex cover that does not contain the vertex. Therefore, we can delete it and include its neighbors to the vertex cover. The third one is the *dominance* rule. We say a vertex v *dominates* a vertex u if $N[u] \subseteq N[v]$. If a vertex v dominates some vertex, there always exists a minimum vertex cover that contains v. Therefore, we can include it to the vertex cover. We note that the degree-1 rule is completely contained in the components rule and the dominance rule. However, it is still useful because its computational cost is smaller in practice. The final rule, *degree-2 folding*, is somewhat tricky. It removes a vertex of degree two and its neighbors while introducing a new vertex as in the following lemma (see Figure 4.3 for an example).

Lemma 4.1 (Degree-2 Folding [40]). *Let v be a vertex of degree two whose two neighbors are not adjacent, and let G' be a graph obtained from G by removing $N[v]$, introducing a new vertex w which is connected to $N^2(v)$. Then, for any $C' \in \mathrm{vc}(G')$, the following C is in $\mathrm{vc}(G)$:*

$$C = \begin{cases} C' \cup \{v\} & (w \notin C'), \\ (C' \setminus \{w\}) \cup N(v) & (w \in C'). \end{cases}$$

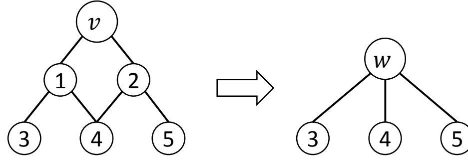

Figure 4.3: An example of the degree-2 folding

4.3.2 Reductions from FPT Algorithms

We use the linear-time LP-based reduction rule (fixing the integral part of an extreme minimum solution) described in Section 3.2. In our implementation, we compute the initial maximum flow by using the Hopcroft-Karp algorithm [50], which runs in $O(|E|\sqrt{|V|})$ time. As we described in Section 3.2, when the graph is changed by reductions or branchings, we do not recompute the maximum flow from scratch but modify the current non-maximum flow to the maximum one by searching for augmenting paths in the residual graph.

4.3.3 Reductions from Exponential Algorithms for Sparse Graphs

Now, we introduce the four reduction rules that appeared in the exact exponential algorithm for sparse graphs by Xiao and Nagamochi [105]. These rules are very complicated, but as we see in Section 4.6, they are quite useful in practice.

The first rule, *unconfined*, is a generalization of the dominance and the satellite rules by Kneis et al. [64]. A vertex v is called unconfined if the following procedure returns yes:

1. Let $S = \{v\}$.
2. Find $u \in N(S)$ such that $|N(u) \cap S| = 1$ and $|N(u) \setminus N[S]|$ is minimized.
3. If there is no such vertex, return no.
4. If $N(u) \setminus N[S] = \emptyset$, return yes.
5. If $N(u) \setminus N[S]$ is a single vertex w, go back to line 2 by adding w to S.
6. Return no.

For any unconfined vertex v, there always exists a minimum vertex cover that contains v. Thus, we can include it to the vertex cover.

The second rule, *twin*, is similar to the degree-2 folding rule. Two vertices u and v are called a twin if $N(u) = N(v)$ and $d(u) = d(v) = 3$. If there is a twin, we can make the graph smaller, as in the following lemma (see Figure 4.4 for an example).

Lemma 4.2 (Twin [105]). *Let u and v be a twin. If there exists an edge among $N(u)$, for any $C' \in \mathrm{vc}(G - N[\{u, v\}])$, $C' \cup N(u) \in \mathrm{vc}(G)$. Otherwise, let G' be a graph obtained from G by removing $N[\{u, v\}]$, introducing a new vertex w connected to $N^2(u) \setminus \{v\}$. Then, for any $C' \in \mathrm{vc}(G')$, the following C is in $\mathrm{vc}(G)$:*

$$C = \begin{cases} C' \cup \{u, v\} & (w \notin C'), \\ (C' \setminus \{w\}) \cup N(u) & (w \in C'). \end{cases}$$

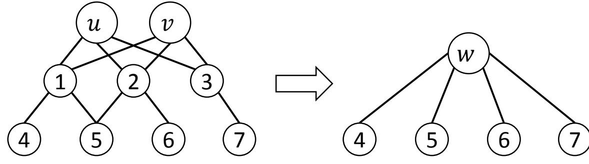

Figure 4.4: An example of the twin

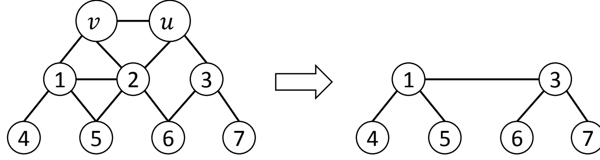

Figure 4.5: An example of the funnel

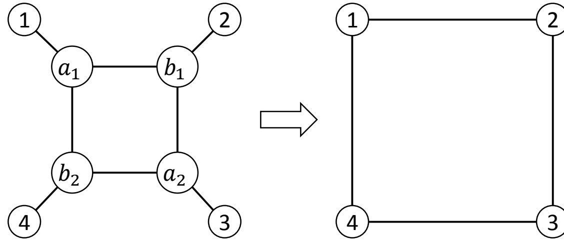

Figure 4.6: An example of the desk

Now, we introduce the notion of *alternative*. Two subsets of vertices A and B are called alternatives if $|A| = |B| \geq 1$ and there exists a minimum vertex cover C that satisfies $C \cap (A \cup B) = A$ or B. The third and fourth rules are special cases of the alternative. Let u, v be adjacent vertices such that $N(v) \setminus \{u\}$ induces a complete graph. Then, $\{u\}$ and $\{v\}$ are alternative sets (called a *funnel*). Let $a_1 b_1 a_2 b_2$ be a chordless 4-cycle such that the degree of each vertex is at least three. Let $A = \{a_1, a_2\}$ and $B = \{b_1, b_2\}$. If it holds that $N(A) \cap N(B) = \emptyset$, $|N(A) \setminus B| \leq 2$, and $|N(B) \setminus A| \leq 2$, then A and B are alternatives (called a *desk*). If there is a funnel or a desk, we can remove it by the following lemma (see Figures 4.5 and 4.6 for examples).

Lemma 4.3 (Alternative [105]). *Let A, B be alternative subsets of vertices, and G' be a graph obtained from G by removing $(N(A) \cap N(B)) \cup A \cup B$ and introducing an edge between every two nonadjacent vertices $u \in N(A) \setminus N[B]$ and $v \in N(B) \setminus N[A]$. Then, for any $C' \in \mathrm{vc}(G')$, the following C is in $\mathrm{vc}(G)$:*

$$C = \begin{cases} C' \cup (N(A) \cap N(B)) \cup A & (N(B) \setminus N[A] \subseteq C'), \\ C' \cup (N(A) \cap N(B)) \cup B & (N(A) \setminus N[B] \subseteq C'). \end{cases}$$

4.3.4 Packing Reductions

In Subsection 4.2.4, we introduced the branching rule that creates auxiliary constraints, called packing constraints, and introduced the simple reduction rule on the basis of these constraints. In this subsection, we introduce more sophisticated

reduction rules to exploit packing constraints. Let $\sum_{v \in S} x_v \leq k$ be a packing constraint such that S is nonempty.

The first rule is for the case in which k is zero. To satisfy the constraint, we cannot include any vertices in S to the vertex cover. Thus, if there is an edge among S, we can prune the subsequent search. Otherwise, we can delete S from the graph while including $N(S)$ to the vertex cover. Here, we can introduce additional packing constraints. Let u be a vertex such that $N(u) \cap S$ is a single vertex w, and let $N^+(u) = N(u) \setminus N[S]$. If we include all the vertices of $N^+(u)$ to the vertex cover, by discarding the vertex u and including w, we can obtain a vertex cover of the same size that does not satisfy the constraint of $\sum_{v \in S} x_v \leq 0$. Thus, we can introduce a new constraint of $\sum_{v \in N^+(u)} x_v \leq |N^+(u)| - 1$.

The second rule is for the case in which k is positive. Let $u \notin S$ be a vertex such that $|S \cap N(u)| > k$. If we do not include u to the vertex cover, all the vertices of $N(u)$ must be contained in the vertex cover. Thus, the constraint is not satisfied. Therefore, we can include u to the vertex cover. Moreover, if at least $|N(u)| - 1$ vertices of $N(u)$ are included to the vertex cover, by discarding u and including the remaining vertex of $N(u)$, we can obtain a vertex cover of the same size that does not satisfy the constraint. Thus, we can introduce a new constraint of $\sum_{v \in N(u)} x_v \leq |N(u)| - 2$.

When we also use reduction rules such as the degree-2 folding, which modifies the graph by deleting some vertices and creating new vertices, the deleted vertices might be included to the vertex cover later on. In that case, we revert the modification until all the vertices in the constraint are recovered and then check the constraint.

4.4 Lower Bounds

We introduce several lower bounds that can be easily computed. In our main implementation, we take the maximum of them as a lower bound.

4.4.1 Clique Cover

A set of disjoint cliques $C_1, \ldots, C_k$ is called *clique cover* if it covers all the vertices. For a clique cover $C_1, \ldots, C_k$, the value $\sum_{i=1}^{k}(|C_i| - 1) = |V| - k$ gives a lower bound for the size of the minimum vertex cover.

In our implementation, we compute a clique cover greedily as follows. First, we sort the vertices by ascending order of their degrees and initiate a set of cliques $\mathcal{C}$ to be an empty set. Then, for each vertex v, we search for a clique $C \in \mathcal{C}$ to which v can be added. If there are multiple possible cliques, we choose the one with maximum size. If there are no such cliques, we add a clique of the single vertex v to $\mathcal{C}$. Since it takes only $O(d(v))$ time for each vertex v, the algorithm runs in linear time in total.

This lower bound is also used in the state-of-the-art branch-and-bound algorithm *MCS* [99]. MCS computes a clique cover using a more sophisticated strategy to obtain a better lower bound. However, it does not scale for large graphs.

4.4.2 LP Relaxation

The optimal value of the LP relaxation gives a lower bound for the size of the minimum vertex cover. After the LP-based reduction, the remaining graph admits a half-integral optimal solution of value $\frac{|V|}{2}$. This lower bound has been used

in FPT algorithms parameterized by the difference between LP lower bounds and the IP optimum [71, 58].

4.4.3 Cycle Cover

A set of disjoint cycles $C_1, \ldots, C_k$ is called *cycle cover* if it covers all the vertices. Here, two adjacent vertices are considered as a cycle of length two, but a single vertex does not form a cycle of length one. For a cycle cover $C_1, \ldots, C_k$, the value $\sum_{i=1}^{k} \left\lceil \frac{|C_i|}{2} \right\rceil$ gives a lower bound for the size of the minimum vertex cover.

We do not have to compute a cycle cover from scratch. After the LP-based reduction of Section 3.2, the maximum flow forms a perfect matching of the bipartite graph $\bar{G} - \{s, t\}$. Thus, by taking an edge uv for each edge $l_u r_v$ in the perfect matching, we can obtain a cycle cover of the graph G in $O(|V|)$ time. Since the optimal value of the LP relaxation is $\frac{|V|}{2} = \sum_{i=1}^{k} \frac{|C_i|}{2}$, the lower bound given by this cycle cover is never worse than the LP optimum. Let $v_1, ..., v_n$ be vertices forming a cycle. If there are four vertices $\{v_i, v_{i+1}, v_j, v_{j+1}\}$ with edges $v_i v_{j+1}$ and $v_j v_{i+1}$, we can split the cycle into two smaller cycles. In our implementation, if it is possible to split a cycle of even length into two smaller cycles of odd length, we split it to improve the lower bound.

4.5 Parameterized Complexity of Vertex Cover above Lower Bounds

The previous theoretical research has shown that if the LP relaxation gives a lower bound that is close to the optimal value, VERTEX COVER can be efficiently solved in the context of parameterized complexity [71]. In our algorithm, we used two different lower bounds, clique cover and cycle cover, which can give a better lower bound than LP relaxation. In this section, we investigate the parameterized complexity of VERTEX COVER above these lower bounds and show that even if these lower bounds are very close to the optimal value, the problem can become very difficult.

4.5.1 Vertex Cover above Clique Cover

Let us define a parameterized problem, VERTEX COVER ABOVE CLIQUE COVER. In this problem, we are given a graph G, a clique cover $\mathcal{C}$ of G, and a parameter k'; our objective is to find a vertex cover of size at most $|V| - |\mathcal{C}| + k'$. Here, $|V| - |\mathcal{C}|$ is the lower bound of the optimal solution size obtained from the given clique cover. In contrast to LP lower bound, we prove that this parameterized problem remains NP-hard even for constant parameter values (i.e., even when the difference between the lower bound obtained from the clique cover and the optimal value is a constant).

Theorem 4.1. VERTEX COVER ABOVE CLIQUE COVER *is NP-hard even when the parameter value is fixed to zero.*

Proof. We prove the theorem by a reduction from 3-SAT. Let $(X, \mathcal{F})$ be an instance of 3-SAT, where $X = \{x_1, x_2, \ldots, x_n\}$ is a set of variables and $\mathcal{F} = \{F_1, F_2, \ldots, F_m\}$ is a set of 3-clauses on X. We write each clause F_i as $F_i = (l_{i,1} \vee l_{i,2} \vee l_{i,3})$, where $l_{i,j}$ is a literal of X, i.e., $l_{i,j} = x$ or $\bar{x}$ for some $x \in X$.

We reduce the instance of 3-SAT to an instance of VERTEX COVER ABOVE CLIQUE COVER with a parameter $k' = 0$ as follows. For each variable $x_i \in X$, we create two vertices v_i and $\bar{v}_i$ and connect them by an edge. Let f be a function

that maps a literal x_i to the vertex v_i and a literal $\bar{x}_i$ to the vertex $\bar{v}_i$. For each clause $F_i = (l_{i,1} \vee l_{i,2} \vee l_{i,3}) \in \mathcal{F}$, we create three vertices $u_{i,1}$, $u_{i,2}$, and $u_{i,3}$, and connect them to form a triangle. Then, for each $j = 1, 2, 3$, we connect $u_{i,j}$ to $f(l_{i,j})$. Finally, we construct a clique cover $\mathcal{C}$ by taking a clique $\{v_i, \bar{v}_i\}$ of size two from each variable $x_i \in X$, and a clique $\{u_{i,1}, u_{i,2}, u_{i,3}\}$ of size three from each clause $F_i \in \mathcal{F}$. The number of the vertices is $2n + 3m$, and the size of this clique cover is $n + m$. Thus, the lower bound obtained from the clique cover is $n + 2m$.

Finally, we prove that, if and only if the instance of 3-SAT is satisfiable, the reduced graph has a vertex cover of size $n + 2m$.

($\Rightarrow$) We construct a vertex cover C as follows. Let π be a truth assignment that satisfies all the clauses. For each variable $x_i \in X$, if $\pi(x_i)$ is true, we include v_i to C; otherwise, we include $\bar{v}_i$ to C. This covers an edge between v_i and $\bar{v}_i$. For each clause $F_i \in \mathcal{F}$, we choose a literal $l_{i,j}$ such that $\pi(l_{i,j})$ is true. Since π is a satisfying assignment, we can always choose such a literal. Then, we include the two vertices other than $u_{i,j}$ from the triangle $\{u_{i,1}, u_{i,2}, u_{i,3}\}$ to C. These cover the edges on the triangle. Moreover, for each $j = 1, 2, 3$, if $\pi(l_{i,j})$ is true, $f(l_{i,j})$ is in C; otherwise, $u_{i,j}$ is in C. Therefore, the edge between $u_{i,j}$ and $f(l_{i,j})$ is also covered. Thus, all the edges are covered by C; i.e., C is a vertex cover. Apparently, the size of C is $n + 2m$.

($\Leftarrow$) We construct a satisfying assignment π as follows. Let C be a vertex cover of size $n + 2m$. Since the lower bound obtained from the clique cover $\mathcal{C}$ is also $n + 2m$, this implies that for each clique $C_i \in \mathcal{C}$, C contains exactly $|C_i| - 1$ vertices from C_i. Therefore, for each variable $x_i \in X$, C contains exactly one of v_i and $\bar{v}_i$. If v_i is contained in C, we assign $\pi(x_i)$ to true; otherwise, we assign $\pi(x_i)$ to false. Now, we show that this assignment π satisfies all the clauses. For each clause $F_i \in \mathcal{F}$, since C contains exactly two vertices from the triangle $\{u_{i,1}, u_{i,2}, u_{i,3}\}$, exactly one vertex $u_{i,j}$ of them is not contained in C. Since C is a vertex cover, its adjacent vertex $f(l_{i,j})$ is contained in C. Thus, $\pi(l_{i,j})$ is true, and therefore, F_i is satisfied by π. □

4.5.2 Vertex Cover above Cycle Cover

Let us define another parameterized problem Vertex Cover above Cycle Cover. Similar to the previous problem, we are given a graph G, a cycle cover $\mathcal{C}$ of G, and a parameter k', and our objective is to find a vertex cover of size at most $\sum_{C \in \mathcal{C}} \left\lceil \frac{|C|}{2} \right\rceil + k'$. Here, $\sum_{C \in \mathcal{C}} \left\lceil \frac{|C|}{2} \right\rceil$ is the lower bound of the optimal solution size obtained from the given cycle cover. Similar to Vertex Cover above Clique Cover, Vertex Cover above Cycle Cover also remains NP-hard even for constant parameter values.

Theorem 4.2. Vertex Cover above Cycle Cover *is NP-hard even when the parameter value is fixed to zero.*

Proof. The proof is almost the same. The size of each clique C in the clique cover $\mathcal{C}$ used in the proof of Theorem 4.1 is two or three. Thus, the clique cover $\mathcal{C}$ is also a cycle cover. Moreover, when $|C| = 2$ or 3, $|C| - 1$ equals to $\left\lceil \frac{|C|}{2} \right\rceil$. Therefore, the lower bound obtained by considering $\mathcal{C}$ as a cycle cover exactly matches the lower bound obtained by considering $\mathcal{C}$ as a clique cover. Thus, we can use the same argument. □

4.6 Experiments

4.6.1 Setup

Experiments were conducted on a machine with Intel Xeon X5670 (2.93 GHz) and 48GB of main memory running Linux 2.6.18. C++ programs were compiled using gcc 4.8.2 with `-O3` option. Java programs were executed with JRE 1.8.0. All the timing results were sequential. We set the time limit for each execution as 24 hours. Timeouts are denoted as '-' in tables.

Instances

As for problem instances, we focused on real large sparse networks. Computing small vertex covers on these networks is important for graph indexing methods [28, 43]. We also used instances from DIMACS Implementation Challenge and the Odd Cycle Transversal problem. Directions of edges are ignored and self-loops were removed beforehand. The detailed description of the three sets of graphs are as follows.

Real Sparse Networks: We focused on real large sparse networks such as social networks, web graphs, computer networks and road networks. They were obtained from the Stanford Large Network Dataset Collection[1], Koblenz Network Collection[2], and Laboratory for Web Algorithmics[3] [24, 23].

DIMACS Instances: DIMACS Instances are those from DIMACS Implementation Challenge on the maximum clique problem [61]. They consist of artificial synthetic graphs and problems reduced from other problems. We used complement graphs of them as Vertex Cover instances. Since they are originally dense graphs and have at most thousands of vertices, explicitly considering complement graphs is feasible and has been often done for benchmarking algorithms for Vertex Cover and Minimum Independent Set. Indeed, these complement graphs are also available online for these problems, and we downloaded them[4].

Instances from Odd Cycle Transversal: The theoretical research on parameterized algorithms suggests that the fastest way to solve Odd Cycle Transversal is to reduce them into Vertex Cover [71]. Therefore, we conduct experiments on the graph reduced from an instance of Odd Cycle Transversal. We used real Odd Cycle Transversal instances from bioinformatics, which formulates the Minimum Site Removal problem[5] [52]. The reduction from Odd Cycle Transversal to Vertex Cover can be found in Subsection 3.1.3.

Methods

We generally compare the three algorithms for Vertex Cover based on different approaches: B&R, CPLEX and MCS [99]. For instances from Odd Cycle Transversal, we also include the results of the algorithm for directly solving Odd Cycle Transversal by Hüffner [52].

[1] http://snap.stanford.edu/data/
[2] http://konect.uni-koblenz.de/
[3] http://law.di.unimi.it/datasets.php
[4] http://www.cs.hbg.psu.edu/txn131/vertex_cover.html
[5] http://www.user.tu-berlin.de/hueffner/occ/

B&R: *B&R* is the branch-and-reduce algorithm stated above, which is implemented in Java. Unless mentioned otherwise, all the branching rules (Section 4.2), all the reduction rules (Section 4.3), and all the lower bounds (Section 4.4) were used. Our implementation is available online[6].

CPLEX: *IBM ILOG CPLEX Optimization Studio (CPLEX)* is a state-of-the-art commercial optimization software package. We used version 12.6 and formulated VERTEX COVER through integer programming. To exactly obtain the minimum vertex cover, we set `mip tolerances mipgap` and `mip tolerances absmipgap` as zero and switched on `emphasis numerical`. Nevertheless, CPLEX did not produce truly optimal solutions for some instances, probably because of numerical precision issues[7]. These results are presented in our tables in parentheses.

MCS: *MCS* [99] is a state-of-the-art branch-and-bound algorithm for the MAXIMUM CLIQUE problem. We used this algorithm for computing minimum vertex cover by virtually considering complement graphs. The algorithm is tailored to DIMACS instances and uses the *greedy coloring technique* to obtain good lower bounds. The algorithm never applies any reductions. It was implemented by the authors in C++.

Hüffner: This is the state-of-the-art algorithm by Hüffner for directly solving ODD CYCLE TRANSVERSAL [52]. This algorithm is based on an FPT algorithm by Reed, Smith and Vetta [91] using the *iterative compression technique*.

4.6.2 Algorithm Comparison

The experimental results on real sparse networks and DIMACS instances are shown in Tables 4.1, 4.2, 4.3, 4.4, and 4.5. For each instance, the table lists the number of vertices ($|V|$), the number of edges ($|E|$), the initial LP lower bound before applying reduction rules (LP), the size of the minimum vertex cover (VC), and results of the three methods. For each method, besides time consumption in seconds (T), the number of branches (#B) are described. For CPLEX, the number of introduced cuts (#C) is also denoted.

We first observe that B&R and CPLEX clearly outperform MCS on real sparse networks. Also, except for road networks, B&R is generally comparable with CPLEX. B&R solves several cases that CPLEX fails to solve within the time limit, such as libimseti, hollywood-2009, and hollywood-2011. Moreover, for some of the other instances, such as petster-cat, soc-LiveJournal1, web-Google, and in-2004, B&R is orders of magnitude faster than CPLEX. On the other hand, for a few web graph such as cnr-2000 and eu-2005, only CPLEX gave an answer within the time limit.

In contrast, on DIMACS instances, as it is tailored to these instances, MCS generally works better. The performances of B&R and CPLEX are comparable. For example, B&R solved some of the *p_hat* instances and *sanr* instances that

[6] `https://github.com/wata-orz/vertex_cover`

[7] We also tested modification of the feasibility tolerance parameter in its simplex routine (`simplex tolerances feasibility`). Indeed, the results were quite sensitive to this parameter, which implies numerical precision issues. We observed that the best results were produced by the default value 10^{-6} in almost all the cases, and thus, the default value was used for this parameter. Consequently, we switched on the numerical precision emphasis parameter (`emphasis numerical`). While it improved the results to some extent, in some instances the results were larger than ours, though we confirmed that our smaller solutions were, indeed, vertex covers.

CPLEX could not solve, but CPLEX solved some of the *gen* instances that B&R could not solve.

Tables 4.6 and 4.7 lists the results on instances from the Odd Cycle Transversal problem. For Hüffner's algorithm, we describe the number of augmentations (#A) instead of the number of branches. We observe that B&R, CPLEX and MCS strongly outperform Hüffner's algorithm.

4.6.3 Observations

Finally, we examine the effect of branching strategies, reduction rules and lower bounds.

Branching Rules

We compared the following three branching strategies. *B0* selects a vertex to branch on in a uniformly random manner, *B1* branches on a vertex with the minimum degree and *B2* chooses a vertex with the maximum degree.

Table 4.8 lists the results, which show that selecting a vertex with the maximum degree (B2) is significantly better than other strategies. This matches the results of theoretical research. Another interesting finding here is that the minimum degree strategy (B1) performs better than the random strategy (B0). This is because our algorithm incorporates mirror branching (Subsection 4.2.2), which occurs more often when branching on vertices with small degrees.

Reduction Rules

To examine the effects of reduction rules, we compare algorithms *R0–R4*, which use different sets of reduction rules. *R0* does not use any reduction rules other than connected component decomposition. *R1* uses the first three reduction rules: degree-1, dominance, and degree-2 folding (Subsection 4.3.1). In addition to the first three reduction rules, *R2* uses the LP-based reduction rule (Subsection 4.3.2). *R3* also adopts unconfined, twin, funnel, and desk (Subsection 4.3.3). *R4* uses all the reduction rules, including the packing rule (Subsection 4.3.4), which is newly introduced in this thesis.

Results are listed in Table 4.9. We can observe the significant effect of reduction rules on the search space. Indeed, without reduction rules, R0 cannot solve any problems. On the other hand, we confirm that search space gets smaller and smaller by introducing reduction rules on instances such as web-Google, web-NotreDame and as-Skitter. We can also see that the number of problems that can be solved within the time limit increases by adopting reduction rules.

Lower Bounds

Finally, we compare algorithms *L0–L4* using different lower bounds. *L0* only uses the number vertices currently included to the vertex cover. *L1*, *L2*, and *L3* use the clique cover (Subsection 4.4.1), LP relaxation (Subsection 4.4.2), and cycle cover (Subsection 4.4.3), respectively. *L4* combines all these lower bounds.

Table 4.10 describes the results. It shows that the difference of lower bounds does not drastically affect the results in comparison to the branching rules and the reduction rules. As expected, the search space of L4 is the smallest among the five methods in all the instances. Since L3 is an extension of L2, it works better than L2 in all the instances. Although L1 works better than L3 in some instances, L3 works better in the other instances.

Table 4.1: Performance comparison of algorithms for Vertex Cover on real sparse graphs (1/2); T denotes running time (in seconds), #B denotes the number of branches, and #C denotes the number of introduced cuts. Timeouts are denoted as '-'. Numbers in parentheses are explained in Subsection 4.6.1.

Instance			LP	VC	B&R		CPLEX			MCS [99]	
Name	$\lvert V\rvert$	$\lvert E\rvert$	**LP**	**VC**	**T**	**#B**	**T**	**#B**	**#C**	**T**	**#B**
Social Networks:											
ca-GrQc	5,242	14,484	2,412.5	2,783	0.1	0	0.1	0	0	61.5	4,351
ca-HepTh	9,877	25,973	4,553.0	4,981	0.2	0	0.2	0	0	2,181.3	33,183
ca-CondMat	23,133	93,439	11,156.5	13,521	0.1	0	0.8	0	0	36,540.5	107,891
wiki-Vote	7,115	100,762	2,249.0	2,249	0.0	0	0.3	0	0	–	–
ca-HepPh	12,008	118,489	5,722.5	7,012	0.1	0	0.7	0	0	1,803.7	26,231
email-Enron	36,692	183,831	12,559.5	14,437	0.6	0	1.4	0	0	–	–
ca-AstroPh	18,772	198,050	9,218.5	12,012	0.1	0	1.6	0	0	–	–
email-EuAll	265,214	364,481	18,315.5	18,316	0.1	0	2.0	0	0	–	–
soc-Epinions1	75,879	405,740	22,080.0	22,280	1.1	0	1.5	0	0	–	–
soc-Slashdot0811	77,360	469,180	23,936.0	24,046	0.2	0	1.9	0	0	–	–
soc-Slashdot0902	82,168	504,230	25,657.5	25,770	0.2	0	2.6	0	0	–	–
dblp-2010	300,647	807,700	138,634.5	166,234	2.0	0	13.0	0	0	–	–
youtube-links	1,138,499	2,990,443	276,684.0	278,125	1.0	0	14.8	0	0	–	–
dblp-2011	933,258	3,353,618	427,093.5	498,969	4.8	0	89.7	0	0	–	–
wiki-Talk	2,394,385	4,659,565	56,111.5	56,163	1.3	0	64.6	0	0	–	–
petster-friendships-cat	149,700	5,448,197	40,975.0	41,103	3.0	0	1,713.4	0	3,042	–	–
petster-friendships-dog	426,820	8,543,549	149,419.5	150,117	5.8	3	1,702.6	0	9,522	–	–
youtube-u-growth	3,223,589	9,376,594	834,673.5	840,060	2.6	0	155.1	0	0	–	–
flickr-links	1,715,255	15,555,041	467,585.5	474,637	2.1	0	121.6	0	132	–	–
petster-carnivore	623,766	15,695,166	290,474.0	371,189	3.6	0	6,240.6	0	67,378	–	–
libimseti	220,970	17,233,144	93,378.0	93,676	1,642.8	472	–	–	–	–	–
soc-pokec	1,632,803	22,301,964	781,066.0	–	–	–	–	–	–	–	–
flickr-growth	2,302,925	22,838,276	632,011.0	640,921	2.8	0	239.6	0	323	–	–
soc-LiveJournal1	4,847,571	42,851,237	2,110,273.5	2,215,668	11.5	36	23,876.5	0	114,581	–	–
hollywood-2009	1,107,243	56,375,711	553,223.0	890,039	26.8	0	–	–	–	–	–
hollywood-2011	1,985,306	114,492,816	992,238.0	1,657,357	50.5	0	–	–	–	–	–
orkut-links	3,072,441	117,185,083	1,519,101.5	–	–	–	–	–	–	–	–

Table 4.2: Performance comparison of algorithms for VERTEX COVER on real sparse graphs (1/2); T denotes running time (in seconds), #B denotes the number of branches, and #C denotes the number of introduced cuts. Timeouts are denoted as '-'. Numbers in parentheses are explained in Subsection 4.6.1.

	Instance				B&R		CPLEX			MCS [99]	
Name	$\|V\|$	$\|E\|$	LP	VC	T	#B	T	#B	#C	T	#B
Web Graphs:											
web-NotreDame	325,729	1,090,108	66,603.5	73,878	13.4	4,266	(181.2)	478	539	–	–
web-Stanford	281,903	1,992,636	108,845.5	118,513	67,270.3	55,865,269	(1,836.1)	1,006	36,237	–	–
baidu-relatedpages	415,641	2,374,044	129,579.5	142,667	2.1	8	979.4	0	31,744	–	–
cnr-2000	325,557	2,738,969	86,869.0	95,577	–	–	4,124.4	503	18,579	–	–
web-Google	875,713	4,322,051	304,486.0	346,575	1.4	10	113.3	0	332	–	–
web-BerkStan	685,230	6,649,470	247,158.5	276,748	142.3	42,270	6,777.4	970	66,277	–	–
in-2004	1,382,869	13,591,473	419,758.0	486,146	3.5	668	9,445.6	484	58,954	–	–
hudong-internallink	1,984,484	14,428,382	499,455.5	502,742	1.9	5	141.7	0	21	–	–
eu-2005	862,664	16,138,468	319,575.5	410,318	–	–	39,847.5	484	56,496	–	–
baidu-internallink	2,141,300	17,014,946	633,556.5	636,967	2.4	0	161.7	0	63	–	–
indochina-2004	7,414,768	150,984,819	2,233,170.5	–	–	–	–	–	–	–	–
uk-2002	18,484,117	261,787,258	5,714,023.5	–	–	–	–	–	–	–	–
Computer Networks:											
p2p-Gnutella08	6,301	20,777	2,054.0	2,054	0.2	0	0.1	0	0	412.7	7,882
p2p-Gnutella09	8,114	26,013	2,574.0	2,574	0.1	0	0.1	0	0	831.2	5,540
p2p-Gnutella06	8,717	31,525	3,405.0	3,405	0.1	0	0.1	0	0	–	–
p2p-Gnutella05	8,846	31,839	3,428.0	3,428	0.0	0	0.1	0	0	–	–
p2p-Gnutella04	10,876	39,994	4,348.0	4,348	0.0	0	0.1	0	0	–	–
p2p-Gnutella25	22,687	54,705	6,017.0	6,017	0.2	0	0.2	0	0	–	–
p2p-Gnutella24	26,518	65,369	7,208.5	7,209	0.0	0	0.2	0	0	–	–
p2p-Gnutella30	36,682	88,328	9,268.0	9,268	0.1	0	0.3	0	0	–	–
p2p-Gnutella31	62,586	147,892	15,693.0	15,693	0.1	0	0.6	0	0	–	–
as-Skitter	1,696,415	11,095,298	517,760.5	525,835	2,769.8	2,123,545	(1,343.0)	522	1,812	–	–
Road Networks:											
roadNet-CA	1,965,206	2,766,607	956,794.5	1,003,354	–	–	3,043.9	495	91,847	–	–
roadNet-PA	1,088,092	1,541,898	529,699.0	554,464	–	–	1,699.0	1,028	51,819	–	–
roadNet-TX	1,379,917	1,921,660	671,220.0	701,625	–	–	2,191.6	990	58,368	–	–

Table 4.3: Performance comparison of algorithms for Vertex Cover on the DIMACS instances (1/3); T denotes running time (in seconds), #B denotes the number of branches, and #C denotes the number of introduced cuts. Timeouts are denoted as '-'. Numbers in parentheses are explained in Subsection 4.6.1.

Instance					B&R		CPLEX			MCS [99]	
Name	$\|V\|$	$\|E\|$	LP	VC	T	#B	T	#B	#C	T	#B
C1000.9	1,000	49,421	500.0	–	–	–	–	–	–	–	–
C125.9	125	787	62.5	91	3.2	3,310	2.8	2,334	24	0.1	10,442
C2000.5	2,000	999,164	1,000.0	1,984	–	–	–	–	–	73,132.7	11,749,950,425
C2000.9	2,000	199,468	1,000.0	–	–	–	–	–	–	–	–
C250.9	250	3,141	125.0	206	53,125.4	98,016,830	–	–	–	2,774.5	223,414,645
C4000.5	4,000	3,997,732	2,000.0	–	–	–	–	–	–	–	–
C500.9	500	12,418	250.0	–	–	–	–	–	–	–	–
MANN_a27	378	702	189.0	252	2.3	1,396	4.4	2,981	25	0.6	9,164
MANN_a45	1,035	1,980	517.5	690	77.7	123,907	15.9	6,179	41	122.3	249,186
MANN_a81	3,321	6,480	1,660.5	–	–	–	–	–	–	–	–
MANN_a9	45	72	22.5	29	0.2	4	0.0	0	8	0.0	43
brock200_1	200	5,066	100.0	179	133.2	313,662	1,125.9	252,187	51	1.0	149,327
brock200_2	200	10,024	100.0	188	5.0	5,279	71.6	6,095	168	0.0	4,183
brock200_3	200	7,852	100.0	185	21.4	42,385	145.4	23,946	103	0.0	8,468
brock200_4	200	6,811	100.0	183	27.4	43,951	276.1	38,764	89	0.1	28,714
brock400_1	400	20,077	200.0	373	–	–	–	–	–	807.8	99,825,943
brock400_2	400	20,014	200.0	371	–	–	–	–	–	453.3	52,750,157
brock400_3	400	20,119	200.0	369	–	–	–	–	–	234.0	26,649,253
brock400_4	400	20,035	200.0	367	–	–	–	–	–	132.9	14,080,352
brock800_1	800	112,095	400.0	777	–	–	–	–	–	10,480.3	1,273,480,056
brock800_2	800	111,434	400.0	776	–	–	–	–	–	6,758.4	708,763,126
brock800_3	800	112,267	400.0	775	–	–	–	–	–	8,571.2	1,104,635,160
brock800_4	800	111,957	400.0	774	–	–	–	–	–	5,292.4	580,958,322
c-fat200-1	200	18,366	100.0	188	1.0	1	11.3	0	141	0.0	3
c-fat200-2	200	16,665	100.0	176	0.4	1	8.5	0	3,422	0.0	1
c-fat200-5	200	11,427	100.0	142	0.2	1	6.2	0	48	0.0	27
c-fat500-1	500	120,291	250.0	486	3.8	13	43.7	0	4,991	0.0	1
c-fat500-10	500	78,123	250.0	374	0.8	0	39.1	0	5,291	0.0	1
c-fat500-2	500	115,611	250.0	474	2.5	0	130.2	0	497	0.0	1
c-fat500-5	500	101,559	250.0	436	1.0	0	77.6	0	5,185	0.0	1

Table 4.4: Performance comparison of algorithms for VERTEX COVER on the DIMACS instances (2/3); T denotes running time (in seconds), #B denotes the number of branches, and #C denotes the number of introduced cuts. Timeouts are denoted as '-'. Numbers in parentheses are explained in Subsection 4.6.1.

Instance			LP	VC	B&R		CPLEX			MCS [99]	
Name	$\|V\|$	$\|E\|$	LP	VC	T	#B	T	#B	#C	T	#B
gen200_p0.9_44	200	1,990	100.0	156	154.5	310,740	7.5	1,266	6	3.5	310,189
gen200_p0.9_55	200	1,990	100.0	145	75.6	153,928	0.6	0	26	1.8	170,075
gen400_p0.9_55	400	7,980	200.0	345	–	–	7,049.4	782,289	17	21,091.5	981,661,757
gen400_p0.9_65	400	7,980	200.0	335	–	–	36.7	583	15	101,264.3	707,230,241
gen400_p0.9_75	400	7,980	200.0	325	–	–	9.0	3	22	23,466.2	1,566,029,572
hamming10-2	1,024	5,120	512.0	512	0.2	0	0.1	0	0	0.1	512
hamming10-4	1,024	89,600	512.0	–	–	–	–	–	–	–	–
hamming6-2	64	192	32.0	32	0.0	0	0.0	0	0	0.0	32
hamming6-4	64	1,312	32.0	60	0.2	71	0.2	35	45	0.0	70
hamming8-2	256	1,024	128.0	128	0.2	0	0.0	0	0	0.0	128
hamming8-4	256	11,776	128.0	240	16.9	14,690	2.3	0	119	0.1	19,567
johnson16-2-4	120	1,680	60.0	112	44.2	306,320	0.0	0	0	0.2	237,952
johnson32-2-4	496	14,880	248.0	480	–	–	0.0	0	0	–	–
johnson8-2-4	28	168	14.0	24	0.4	26	0.0	0	0	0.0	26
johnson8-4-4	70	560	35.0	56	0.7	385	0.0	0	0	0.0	190
keller4	171	5,100	85.5	160	4.2	4,201	24.5	5,822	60	0.0	6,800
keller5	776	74,710	388.0	749	–	–	–	–	–	137,802.4	1,194,276,461
keller6	3,361	1,026,582	1,680.5	–	–	–	–	–	–	–	–

Table 4.5: Performance comparison of algorithms for VERTEX COVER on the DIMACS instances (3/3); T denotes running time (in seconds), #B denotes the number of branches, and #C denotes the number of introduced cuts. Timeouts are denoted as '-'. Numbers in parentheses are explained in Subsection 4.6.1.

Instance					B&R		CPLEX			MCS [99]	
Name	$\|V\|$	$\|E\|$	LP	VC	T	#B	T	#B	#C	T	#B
p_hat1000-1	1,000	377,247	500.0	990	781.0	203,758	–	–	–	0.6	122,866
p_hat1000-2	1,000	254,701	500.0	954	27,819.4	16,035,941	–	–	–	266.6	12,300,564
p_hat1000-3	1,000	127,754	500.0	–	–	–	–	–	–	–	–
p_hat1500-1	1,500	839,327	750.0	1,488	9,959.7	2,890,004	–	–	–	4.5	834,181
p_hat1500-2	1,500	555,290	750.0	1,435	–	–	–	–	–	16,727.4	493,777,410
p_hat1500-3	1,500	277,006	750.0	–	–	–	–	–	–	–	–
p_hat300-1	300	33,917	150.0	292	6.0	2,838	329.5	5,777	892	0.0	1,211
p_hat300-2	300	22,922	150.0	275	6.7	4,104	224.4	5,794	675	0.0	1,962
p_hat300-3	300	11,460	150.0	264	159.2	173,032	8,004.6	515,111	192	2.3	191,209
p_hat500-1	500	93,181	250.0	491	44.1	18,517	8,250.7	59,203	2,530	0.0	7,395
p_hat500-2	500	61,804	250.0	464	80.3	46,461	18,872.3	121,888	1,842	0.6	34,258
p_hat500-3	500	30,950	250.0	450	12,228.6	9,736,947	–	–	–	199.8	10,120,545
p_hat700-1	700	183,651	350.0	689	169.6	35,442	59,953.0	155,180	6,050	0.1	19,093
p_hat700-2	700	122,922	350.0	656	701.2	343,613	–	–	–	5.9	275,676
p_hat700-3	700	61,640	350.0	638	–	–	–	–	–	2,506.1	88,791,027
san1000	1,000	249,000	500.0	985	610.3	146,929	6,787.9	57,314	235	2.5	64,944
san200_0.7_1	200	5,970	100.0	170	9.9	14,066	0.1	0	0	0.0	606
san200_0.7_2	200	5,970	100.0	182	1.2	227	4.4	445	70	0.0	307
san200_0.9_1	200	1,990	100.0	130	15.2	17,643	0.0	0	0	0.0	711
san200_0.9_2	200	1,990	100.0	140	38.4	71,547	0.0	0	0	0.1	6,340
san200_0.9_3	200	1,990	100.0	156	1,077.7	2,614,156	0.4	0	9	0.1	6,621
san400_0.5_1	400	39,900	200.0	387	8.9	3,074	9.0	10	77	0.0	1,190
san400_0.7_1	400	23,940	200.0	360	710.5	552,711	87.0	1,021	107	2.5	130,667
san400_0.7_2	400	23,940	200.0	370	1,995.4	2,208,212	12.3	0	33	7.0	375,023
san400_0.7_3	400	23,940	200.0	378	3,155.2	4,683,473	437.7	31,599	31	8.6	675,084
san400_0.9_1	400	7,980	200.0	300	–	–	5.0	0	26	69,041.6	3,762,277,624
sanr200_0.7	200	6,032	100.0	182	46.2	87,537	468.0	85,363	45	0.4	65,322
sanr200_0.9	200	2,037	100.0	158	702.7	1,690,472	1,523.7	603,714	37	41.5	3,369,435
sanr400_0.5	400	39,816	200.0	387	270.2	348,942	26,583.6	878,489	356	0.7	143,629
sanr400_0.7	400	23,931	200.0	379	24,365.0	41,203,755	–	–	–	198.8	30,154,732

Table 4.6: Performance comparison of algorithms for ODD CYCLE TRANSVERSAL (1/2); T denotes running time (in seconds), #B denotes the number of branches, #C denotes the number of introduced cuts, and #A denotes the number of augmentations. B&R and CPLEX are given the reduced graphs $\hat{G}$, while Hüffner is given the original OCT instances. $|V|$ and $|E|$ denote the sizes of the reduced graphs.

	Instance				B&R		CPLEX			Hüffner [52]	
Name	$\lvert V\rvert$	$\lvert E\rvert$	**LP**	**VC**	**T**	**#B**	**T**	**#B**	**#C**	**T**	**#A**
afro-16	26	43	13	13	0.0	0	0.0	0	0	0.0	0
afro-25	28	44	14	15	0.0	0	0.0	0	4	0.0	0
afro-31	60	132	30	32	0.0	0	0.0	0	0	0.0	5
afro-49	52	150	26	31	0.0	0	0.0	0	0	0.0	25
afro-30	78	181	39	43	0.0	6	0.0	0	10	0.0	38
afro-33	386	1,179	193	197	0.1	15	0.0	0	0	0.0	39
afro-37	144	412	72	77	0.1	4	0.0	0	13	0.0	92
afro-10	138	451	69	75	0.1	16	0.0	0	9	0.0	97
afro-21	56	208	28	37	0.0	0	0.0	0	0	0.0	245
afro-15	132	424	66	73	0.1	12	0.0	0	9	0.0	339
afro-51	156	626	78	89	0.1	15	0.0	0	16	0.0	1,112
afro-36	222	743	111	118	0.1	11	0.1	0	21	0.0	1,392
afro-11	204	716	102	113	0.2	20	0.2	0	24	0.0	1,945
afro-18	174	849	87	101	0.2	27	0.0	0	9	0.0	2,319
afro-13	258	895	129	141	0.1	29	0.1	0	24	0.0	3,081
afro-35	164	620	82	92	0.1	21	0.0	0	0	0.0	5,276
afro-27	236	780	118	129	0.5	20	0.5	0	36	0.0	5,796
afro-44	118	385	59	69	0.1	19	0.1	0	10	0.0	6,402
afro-26	184	660	92	105	0.1	29	0.3	0	16	0.0	8,481
afro-52	130	527	65	79	0.1	34	0.1	0	10	0.0	11,195
afro-47	124	520	62	76	0.1	20	0.1	0	14	0.0	13,793
afro-46	322	1,219	161	174	0.3	86	0.1	0	18	0.0	14,120
afro-34	266	1,035	133	146	0.2	22	0.0	0	0	0.0	16,413
afro-54	178	555	89	101	0.2	54	0.4	5	42	0.0	20,385
afro-53	176	552	88	100	0.1	20	0.2	27	20	0.0	21,728
afro-22	334	1,449	167	183	0.2	36	0.2	0	88	0.0	22,607
afro-23	278	1,155	139	157	0.2	31	0.5	0	34	0.0	23,322
afro-50	226	1,049	113	131	0.2	33	0.3	4	12	0.0	26,711
afro-20	448	1,756	224	243	0.3	41	1.0	3	41	0.1	32,049
afro-48	178	775	89	106	0.2	31	0.5	0	24	0.0	41,498
afro-29	552	2,392	276	297	0.4	66	0.3	0	55	0.1	56,095
afro-45	160	852	80	100	0.2	55	0.4	3	20	0.1	99,765
afro-43	126	679	63	81	0.2	32	0.3	41	25	0.1	102,609
afro-39	288	1,528	144	167	0.3	58	1.3	0	34	0.4	281,403

Table 4.7: Performance comparison of algorithms for Odd Cycle Transversal (2/2); T denotes running time (in seconds), #B denotes the number of branches, #C denotes the number of introduced cuts, and #A denotes the number of augmentations. B&R and CPLEX are given the reduced graphs $\hat{G}$, while Hüffner is given the original OCT instances. $|V|$ and $|E|$ denote the sizes of the reduced graphs.

	Instance				**B&R**		**CPLEX**			**Hüffner [52]**	
Name	$\lvert V\rvert$	$\lvert E\rvert$	**LP**	**VC**	**T**	**#B**	**T**	**#B**	**#C**	**T**	**#A**
afro-40	272	1,376	136	158	0.4	80	0.2	0	47	0.5	333,793
afro-28	334	1,875	167	194	0.5	72	0.9	22	51	0.6	464,272
afro-38	342	1,895	171	197	0.3	51	0.2	0	80	0.9	631,053
afro-14	250	1,175	125	144	0.2	50	1.0	9	40	2.2	1,707,228
afro-19	382	1,481	191	210	0.3	53	1.0	0	34	1.9	1,803,293
afro-24	516	2,474	258	279	0.3	42	0.3	0	69	5.3	1,998,636
afro-17	302	1,417	151	176	0.9	197	1.1	87	62	3.1	2,342,879
afro-42	472	2,456	236	266	0.6	114	2.0	45	76	41.4	22,588,100
afro-32	286	1,643	143	173	0.8	469	1.6	133	56	49.1	29,512,013
afro-41	592	3,536	296	336	1.0	294	2.6	37	78	128.2	55,758,998
jap-15	88	154	44	45	0.1	0	0.0	0	0	0.0	0
jap-16	18	29	9	9	0.0	0	0.0	0	0	0.0	0
jap-25	28	42	14	14	0.1	0	0.0	0	0	0.0	0
jap-20	482	1,521	241	242	0.1	1	0.0	0	0	0.0	2
jap-10	110	289	55	58	0.1	4	0.0	0	0	0.0	8
jap-19	168	428	84	87	0.0	3	0.0	0	0	0.0	10
jap-14	120	274	60	64	0.0	3	0.0	0	0	0.0	16
jap-24	284	916	142	146	0.1	3	0.0	0	0	0.0	31
jap-11	102	475	51	56	0.1	5	0.0	0	0	0.0	79
jap-13	156	498	78	84	0.1	3	0.0	0	0	0.0	112
jap-26	126	375	63	69	0.1	9	0.0	0	14	0.0	216
jap-21	66	237	33	42	0.0	1	0.0	0	0	0.0	384
jap-18	142	663	71	80	0.1	14	0.0	0	0	0.0	568
jap-17	158	723	79	89	0.1	13	0.0	0	4	0.0	1,591
jap-22	150	857	75	84	0.1	7	0.0	0	0	0.0	1,921
jap-28	180	1,224	90	103	0.2	24	0.0	0	0	0.0	17,886
jap-23	152	814	76	95	0.2	34	0.1	0	15	0.2	229,627

Table 4.8: Comparison of branching rules. T denotes running time (in seconds) and #B denotes the number of branches.

Instance	B0		B1		B2	
	T	#B	T	#B	T	#B
petster-dog	6.0	3	5.8	14	5.8	3
hudong-internal	2.3	19	1.8	9	1.9	5
baidu-related	–	–	–	–	2.1	8
web-Google	1.3	5	1.5	17	1.4	10
soc-LiveJournal1	11.3	314	10.8	195	11.5	36
libimseti	–	–	–	–	1,642.8	472
in-2004	37.2	30,344	28.4	21,377	3.5	668
web-NotreDame	–	–	687.8	356,138	13.4	4,266
web-BerkStan	–	–	–	–	142.3	42,270
as-Skitter	–	–	–	–	2,769.8	2,123,545
web-Stanford	–	–	–	–	67,270.3	55,865,269

Table 4.9: Comparison of reduction rules. T denotes running time (in seconds) and #B denotes the number of branches.

Instance	R0		R1		R2		R3		R4	
	T	#B	T	#B	T	#B	T	#B	T	#B
petster-dog	–	–	–	–	4,724.6	5,557,005	8.5	4	5.8	3
hudong-internal	–	–	24.5	185	2.0	8	2.0	5	1.9	5
baidu-related	–	–	–	–	1.9	152	1.9	8	2.1	8
web-Google	–	–	1.5	602	1.5	165	1.0	10	1.4	10
soc-LiveJournal1	–	–	–	–	45.0	7,500	9.6	33	11.5	36
libimseti	–	–	–	–	837.0	476	1,371.5	472	1,642.8	472
in-2004	–	–	–	–	28.0	30,824	4.6	1,504	3.5	668
web-NotreDame	–	–	–	–	747.7	1,088,096	29.3	16,563	13.4	4,266
web-BerkStan	–	–	–	–	–	–	7,986.2	3,898,313	142.3	42,270
as-Skitter	–	–	–	–	10,507.3	16,422,252	7,768.1	6,262,544	2,769.8	2,123,545
web-Stanford	–	–	–	–	–	–	–	–	67,270.3	55,865,269

Table 4.10: Comparison of lower bounds. T denotes running time (in seconds) and #B denotes the number of branches.

Instance	L0		L1		L2		L3		L4	
	T	#B	T	#B	T	#B	T	#B	T	#B
petster-dog	6.1	122	6.2	15	6.0	7	5.4	3	5.8	3
hudong-internal	2.0	16	1.9	6	2.0	6	1.9	5	1.9	5
baidu-related	74.7	27,047	1.9	8	2.7	77	2.6	68	2.1	8
web-Google	1.1	47	1.0	10	1.3	30	1.0	30	1.4	10
soc-LiveJournal1	12.6	180	10.6	37	12.0	122	11.0	117	11.5	36
libimseti	–	–	–	–	1,747.0	476	1,776.5	472	1,642.8	472
in-2004	4.4	878	4.1	683	5.2	781	4.4	757	3.5	668
web-NotreDame	17.9	13,740	14.2	5,678	13.7	5,702	13.7	4,447	13.4	4,266
web-BerkStan	261.2	206,209	145.3	42,503	199.6	113,184	176.0	62,877	142.3	42,270
as-Skitter	4,963.5	6,973,582	2,629.6	2,153,280	4,165.8	4,873,893	3,968.9	4,083,355	2,769.8	2,123,545
web-Stanford	–	–	–	–	–	–	64,757.6	55,912,396	67,270.3	55,865,269

Chapter 5

FPT Algorithms via Discrete Relaxations

In this chapter, we propose a new technique called *discrete relaxation* for widening the applicability of FPT branch-and-bound methods. First, in Section 5.1, we introduce powerful tools from the study of valued constraint satisfaction problems (VCSPs). Then, in Section 5.2, we formally define the discrete relaxations and present FPT branch-and-bound algorithms. In Section 5.3, we introduce *k-submodular* relaxations and give unified proofs for Vertex Cover and Edge Multiway Cut, and a new algorithm for Unique Label Cover. In Section 5.4, we improve the $n^{O(1)}$ part of the running time and obtain *linear-time* FPT algorithms by generalizing the algorithm in Chapter 3. Finally, in Section 5.5, we deal with vertex-deletion problems. We note that the results in Sections 5.2 and 5.3 are also contained in the extended abstract by Wahlström [101] based on which we wrote the joint version [59]. Although the symbol 'k' is usually used to denote the parameter, in this chapter, we use the symbol 'p' for avoiding the confusion.

5.1 Valued Constraint Satisfaction Problems

5.1.1 Definitions

Let D be a finite set called the *domain*. A *cost function* on D is a function $f : D^r \to \mathbb{R}_{\geq 0}$, where r is the *arity* of f. For a constraint R on D^r, let the *soft version* of R denote the cost function $f : D^r \to \{0, 1\}$ such that $f(\boldsymbol{a}) = 0$ if $R(\boldsymbol{a})$ is satisfied, and $f(\boldsymbol{a}) = 1$ otherwise.

A *valued constraint language* Γ is a set of cost functions. We denote by Γ_c the set of all functions obtained from functions in Γ by fixing a subset of the variables to domain values. For any function $f \in \Gamma_c$, a function $f_{i,a}$, which is defined by $f_{i,a}(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_r) = f(x_1, \ldots, x_{i-1}, a, x_{i+1}, \ldots, x_r)$, is also in Γ_c. We use this property for implementing branching. A language Γ is called *closed under substitution* if it holds that $\Gamma = \Gamma_c$. In our research, we are interested in languages with bounded fractionality. Let c be the smallest integer such that $c \cdot f$ becomes integral for any cost function $f \in \Gamma$, i.e., the codomain of f is $\frac{1}{c}\mathbb{N}$. Then, we call the language is *$\frac{1}{c}$-integral*. A $\frac{1}{2}$-integral language is called *half-integral*.

An application $f(x_1, \ldots, x_r)$ of a cost function $f : D^r \to \mathbb{R}_{\geq 0}$ to a tuple of variables $(x_1, \ldots, x_r)$ is called a *valued constraint*. For a fixed valued constraint language Γ, a problem VCSP(Γ) is defined as follows. An instance of VCSP(Γ) is a pair of a variable set $X = \{x_1, \ldots, x_n\}$ and a list of valued constraints $(f_1(x_{1,1}, \ldots, x_{1,r_1}), \ldots, f_m(x_{m,1}, \ldots, x_{m,r_m}))$, where $f_i \in \Gamma$ and $x_{i,j} \in X$ for each i and j. The *objective function* of the instance I is a function $f_I : D^X \to \mathbb{R}_{\geq 0}$ defined by $f_I(x_1, \ldots, x_n) = \sum_{i=1}^{m} f_i(x_{i,1}, \ldots, x_{i,r_i})$. The objective is to find an assignment $\phi : X \to D$ that minimizes $f_I(\phi) = f_I(\phi(x_1), \ldots, \phi(x_n))$. We denote the optimal value of an instance I by OPT(I).

A valued constraint language Γ is called *tractable* if, for any finite subset $\Gamma' \subseteq \Gamma$, VCSP(Γ') is in P. The complexity of VCSP(Γ) depends on the choice of the language Γ. By choosing appropriate Γ, we can define a variety of problems as VCSP(Γ). Let us see two examples.

Example 5.1. *Let $D = \{0, 1\}$ and $\Gamma = \{f_0, f_1, f_{\leq}\}$ which are defined as follows: $f_0(x) = x$, $f_1(x) = 1 - x$, $f_{\leq}(x, y) = 0$ if $x \leq y$, and $f_{\leq}(x, y) = 1$ if $x > y$. Then,* VCSP(Γ) *becomes equivalent to the (directed) minimum s-t cut problem by the following correspondence between valued constraints and edges: $f_0(v) \leftrightarrow sv$, $f_1(v) \leftrightarrow vt$, and $f_{\leq}(u, v) \leftrightarrow uv$. Therefore,* VCSP($\Gamma$) *is in* P.

Example 5.2. *Let $D = \{0, 1\}$ and $\Gamma = \{f_0, f_\vee\}$ which are defined as follows: $f_0(x) = x$, $f_\vee(0, 0) = 1$, and $f_\vee(x, y) = 0$ otherwise. Then,* VCSP(Γ) *can encode* Vertex Cover *by introducing the valued constraint $f_0(v)$ for each vertex v and the valued constraint $f_\vee(u, v)$ for each edge uv. Therefore,* VCSP(Γ) *is* NP*-hard.*

5.1.2 Tractable VCSPs

A series of work by Thapper and Živný and by Kolmogorov [97, 65, 98] gave a complete dichotomy theorem for tractable VCSPs; they showed that Γ is tractable if and only if it admits a *binary symmetric fractional polymorphism.* Moreover, they also showed that a simple LP relaxation (called the *basic LP relaxation*) can solve any tractable VCSPs exactly. Here, we note that the size of the basic LP relaxation is exponential in the arity of the valued constraints. If all cost functions in Γ are bounded-arity, we can solve VCSP(Γ) in polynomial time by using the basic LP relaxation. However, if Γ contains unbounded-arity cost functions (i.e., the arity of valued constraints can depend on the number of variables), the basic LP relaxation does not give a polynomial-time algorithm. Here, we assumed that each valued constraint is given as an oracle and thus the input size is polynomial in $|D|$, n, and m. We note that in this setting, the tractability of a language Γ containing unbounded-arity functions does not imply that VCSP(Γ) is in P[1] because the tractability of Γ is defined by a *finite* subset of Γ and any cost function in a finite language is bounded-arity.

In our research, we do not need the precise definition of the fractional polymorphism and the following simpler notion is enough. Let $f : D^r \to \mathbb{R}$ be a cost function. A *binary multimorphism* of f is a pair of operations $\langle h_1, h_2 \rangle : D^2 \to D$ such that for any $\boldsymbol{a}, \boldsymbol{b} \in D^r$, we have $f(\boldsymbol{a}) + f(\boldsymbol{b}) \geq f(h_1(\boldsymbol{a}, \boldsymbol{b})) + f(h_2(\boldsymbol{a}, \boldsymbol{b}))$. Here, the operations are applied componentwise, e.g., $h((a, b, c), (d, e, f)) = (h(a, d), h(b, e), h(c, f))$. Similarly, $\langle h_1, h_2 \rangle$ is a multimorphism of a valued constraint language Γ if it is a multimorphism of every $f \in \Gamma$. A multimorphism $\langle h_1, h_2 \rangle$ is called *symmetric* if it holds that $h_i(a, b) = h_i(b, a)$ for any $a, b \in D$ and $i \in \{1, 2\}$, and called *idempotent* if it holds that $h_i(a, a) = a$ for any $a \in D$ and $i \in \{1, 2\}$. Since the multimorphism is a special case of the fractional polymorphism, if a language Γ admits a binary symmetric multimorphism, it is tractable. We note that, from the definitions, any binary symmetric and idempotent multimorphism of Γ is also a multimorphism of Γ_c. Thus, if Γ admits a binary symmetric and idempotent multimorphism, Γ_c is also tractable.

Submodular functions would be the prime example of tractable functions. A set function $f : 2^X \to \mathbb{R}$ is called submodular if it satisfies the inequality $f(A) + f(B) \geq f(A \cap B) + f(A \cup B)$ for any $A, B \subseteq X$. We can observe that the

[1]If we assume that each valued constraint of arity r is given as a list of $|D|^r$ values, the size of the basic LP relaxation becomes polynomial in the input size and thus the basic LP relaxation can solve the problem in polynomial time.

pair of operations $\langle \cap, \cup \rangle$ is a binary symmetric and idempotent multimorphism. The submodular functions can be minimized in polynomial time, even when the function is given as an oracle [55, 94]. Various generalizations of submodular functions, including submodular on an arbitrary lattice and (weakly or strongly) tree-submodular functions [97], can be shown to be tractable by this framework.

In our research, we will mainly use a class of *k-submodular* functions introduced by Huber and Kolmogorov [51], which is a generalization of submodular functions and a special case of tree-submodular functions. Let $D = \{0, 1, \ldots, k\}$ be a domain and let $\sqcap, \sqcup$ be binary symmetric and idempotent operations defined as follows; $0 \sqcap a = 0$ for any $a \in D$; $0 \sqcup a = a$ for any $a \in D$; and $a \sqcap b = a \sqcup b = 0$ for any distinct $a, b \in D \setminus \{0\}$. A function $f : D^r \to \mathbb{R}$ is called k-submodular if it admits $\langle \sqcap, \sqcup \rangle$ as a multimorphism. The value $0 \in D$, which plays the special role in the definition, is called the *root*. The 1-submodular functions correspond to submodular functions and the 2-submodular functions are referred to as *bisubmodular functions*. Similarly to the submodular functions, bisubmodular functions can be minimized in polynomial time, even in a value oracle model [42]. On the other hand, polynomial time algorithms for minimizing a k-submodular function, which is given as an oracle, are not known yet. However, if a function can be written as a sum of bounded-arity k-submodular functions, we can minimize it in polynomial time by using the basic LP relaxation.

5.2 Discrete Relaxations

The existing FPT branch-and-bound algorithms for Vertex Cover [71] and Node Multiway Cut [34] rely on two properties of the LP relaxations: the half-integrality and the persistency. In our approach, instead of showing the half-integrality of an LP relaxation of a problem, we first define the problem as VCSP(Γ) on a domain D and then relax it to a tractable VCSP(Γ') by extending the domain to $D' \supset D$, which is called a *discrete relaxation* of VCSP(Γ). We can prove the tractability just by showing a binary symmetric multimorphism, and moreover, we can easily prove the persistency of the relaxation by using the multimorphism. In this section, we formally define the notion of the discrete relaxations and the persistency, and then present FPT branch-and-bound algorithms based on discrete relaxations. We use the same definitions as in the joint paper [59] which is based on the extended abstract by Wahlström [101].

For a cost function $f : D^r \to \mathbb{R}_{\geq 0}$, we say that a function $f' : (D')^r \to \mathbb{R}_{\geq 0}$ on an extended domain $D' \supset D$ is a *discrete relaxation* of f if it satisfies $f(\boldsymbol{a}) = f'(\boldsymbol{a})$ for every $\boldsymbol{a} \in D^r$. We call the original domain D as an *integral domain* and the extended domain D' as a *relaxed domain*. Similarly, for a valued constraint language Γ, we say that a language $\Gamma' = \{f' \mid f \in \Gamma\}$ is a discrete relaxation of Γ if each $f' \in \Gamma'$ is a discrete relaxation of $f \in \Gamma$. If Γ is integral and Γ' is $\frac{1}{c}$-integral, we say that Γ' is a $\frac{1}{c}$*-integral discrete relaxation* of Γ.

Let Γ be a language on a domain D and Γ' be a discrete relaxation of Γ. An instance I of VCSP(Γ) is relaxed to an instance I' of VCSP(Γ') by replacing every cost function $f \in \Gamma$ by the corresponding relaxation $f' \in \Gamma'$. We call the difference $\mathrm{OPT}(I) - \mathrm{OPT}(I')$ as the *relaxation gap* of I. We say that the discrete relaxation Γ' is *persistent* if for any instance I of VCSP(Γ) and for any optimal assignment $\phi^* : X \to D'$ of the relaxed instance I' of VCSP(Γ'), there is an optimal assignment $\phi : X \to D$ of the original instance I satisfying $\phi(x) = \phi^*(x)$ for any variable $x \in X$ with $\phi^*(x) \in D$.

As an example of the above definitions, let us see an application to Vertex Cover. As we saw in Example 5.2, the language $\Gamma = \{f_0, f_\vee\}$ defined on the

Algorithm 5 FPT branch-and-bound via discrete relaxation

1: **procedure** SOLVE(I')
2:     **if** $\mathrm{OPT}(I') > p$ **then return** false
3:     **if** the variable set X of I' is empty **then return** true
4:     pick a variable $x \in X$
5:     **for** $a \in D$ **do**
6:         **if** $\mathrm{OPT}(I'[x \leftarrow a]) = \mathrm{OPT}(I')$ **then**
7:             **return** SOLVE($I'[x \leftarrow a]$)
8:     **return** $\bigvee_{a \in D}$ SOLVE($I'[x \leftarrow a]$)

domain $D = \{0, 1\}$ can encode VERTEX COVER, and thus VCSP(Γ) is NP-hard. Let us define a discrete relaxation Γ' of Γ as follows. First, we relax the domain D to $D' = \{0, \frac{1}{2}, 1\}$ by adding the half-integral value $\frac{1}{2}$ to the original domain. Then, the cost functions can be relaxed to f'_0 and $f'_\vee$ by setting $f'_0(x) = x$ and $f'_\vee(x, y) = \max(0, 1-x-y)$. We can observe that this discrete relaxation actually corresponds to the classical half-integral LP relaxation of VERTEX COVER (see Subsection 3.1.1), and therefore, it is a persistent half-integral relaxation and can be solved in polynomial time. As we will see in Subsection 5.3.2, the relaxed functions become bisubmodular by taking $\frac{1}{2}$ as the root. Thus, without using the correspondence to the LP relaxation, we can directly show the tractability of the relaxed problem.

Using persistent relaxations, we can establish FPT branch-and-bound algorithms as following shows.

Lemma 5.1. *Let Γ be a valued constraint language on a domain D which is closed under substitution and admits a persistent $\frac{1}{c}$-integral discrete relaxation Γ' such that* VCSP(Γ') *is in* P*. Then, any instance I of* VCSP(Γ) *can be solved in $O^*(|D|^{cp'})$ time, where $p' = \mathrm{OPT}(I) - \mathrm{OPT}(I')$ is the relaxation gap.*

Proof. Let I be the input instance. First, we create the relaxed instance I' and compute the initial lower bound $\mathrm{OPT}(I')$. Our task is to find an assignment of value at most $p = \mathrm{OPT}(I') + p'$. We solve the problem by using the branch-and-bound algorithm described in Algorithm 5. In the recursive call of the algorithm, we first compute the lower bound $\mathrm{OPT}(I')$. If the lower bound is larger than p, we prune the search (line 2). If there are no remaining variables, we have found the assignment and finish the algorithm (line 3). Otherwise, we pick an arbitrary variable $x \in X$ and attempt to fix x to a for every $a \in D$ in turn. Let $I'[x \leftarrow a]$ denote the instance obtained by fixing x to a. Since this is an instance of VCSP(Γ'), we can compute the optimal value in polynomial time. If there is a value $a \in D$ such that fixing x to a does not increase the optimal value, we can safely fix it without branching because the relaxation is persistent (line 7). Otherwise, we branch into $|D|$ cases (line 8). Since the relaxation is $\frac{1}{c}$-integral, in each branch, the lower bound increases by at least $\frac{1}{c}$. Thus, the depth of the search tree is bounded by cp', which leads to the running time of $O^*(|D|^{cp'})$. □

For several cases, the factor $|D|$ in the base of the running time in the above lemma can be improved to a constant by changing the branching strategy (see the case of EDGE MULTIWAY CUT in Subsection 5.3.3). However, such an improvement would not possible in general (see the case of UNIQUE LABEL COVER in Subsection 5.3.3).

5.3 k-submodular Relaxations

We now investigate the power of k-submodular functions for discrete relaxation. Let Γ be a valued constraint language on a domain $D = \{1, 2, \ldots, k\}$ and Γ' be a discrete relaxation of Γ on a relaxed domain $D' = \{0, 1, 2, \ldots, k\}$. If Γ' is k-submodular (by considering 0 as the root), it is called a *k-submodular relaxation.* First, we show the persistency of k-submodular relaxations.

Lemma 5.2. *Any k-submodular relaxation is persistent.*

Proof. Let $(\circ)$ be an operation defined by $a \circ b = (a \sqcup b) \sqcup b$. We can observe that $a \circ 0 = a$ holds for any $a \in D'$ and $a \circ b = b$ holds for any $a, b \in D$. Let $f : D'^X \to \mathbb{R}$ be a k-submodular function and $\boldsymbol{b}^* \in D'^X$ be a (relaxed) optimum. Then, for any $\boldsymbol{a} \in D'^X$, we have:

$$\begin{aligned} f(\boldsymbol{a} \sqcup \boldsymbol{b}^*) &\le f(\boldsymbol{a} \sqcup \boldsymbol{b}^*) + f(\boldsymbol{a} \sqcap \boldsymbol{b}^*) - f(\boldsymbol{b}^*) && \text{(optimality)} \\ &\le f(\boldsymbol{a}) + f(\boldsymbol{b}^*) - f(\boldsymbol{b}^*) && \text{(k-submodularity)} \\ &= f(\boldsymbol{a}). \end{aligned}$$

Therefore, we have $f(\boldsymbol{a} \circ \boldsymbol{b}^*) = f((\boldsymbol{a} \sqcup \boldsymbol{b}^*) \sqcup \boldsymbol{b}^*) \le f(\boldsymbol{a} \sqcup \boldsymbol{b}^*) \le f(\boldsymbol{a})$. Thus, for any integral optimum $\boldsymbol{a}^* \in D^X$, $\boldsymbol{a}^* \circ \boldsymbol{b}^*$ also becomes an integral optimum, and moreover it agrees with $\boldsymbol{b}^*$ on the integral coordinates of $\boldsymbol{b}^*$. □

By applying Lemma 5.1, we get the following corollary.

Corollary 5.1. *Let Γ be a valued constraint language which is closed under substitution and admits a $\frac{1}{c}$-integral k-submodular relaxation Γ' such that* VCSP(Γ') *is in* P. *Then, any instance I of* VCSP(Γ) *can be solved in $O^*(k^{cp'})$ time, where $p' = \mathrm{OPT}(I) - \mathrm{OPT}(I')$ is the relaxation gap.*

As we noted in Subsection 5.1.2, for a k-submodular language Γ containing unbounded-arity functions, no polynomial-time algorithms for VCSP(Γ) are known yet. However, if every cost function in Γ is bounded-arity, we can solve VCSP(Γ) in polynomial time by using the basic LP relaxation. On the other hand, for a bisubmodular language Γ, we can solve VCSP(Γ) in polynomial time even if Γ contains unbounded-arity functions. Thus, we can obtain the following corollary.

Corollary 5.2. *Let Γ be a valued constraint language with a $\frac{1}{c}$-integral bisubmodular relaxation Γ'. Then, any instance I of* VCSP(Γ) *can be solved in $O^*(2^{cp'})$ time, where $p' = \mathrm{OPT}(I) - \mathrm{OPT}(I')$ is the relaxation gap.*

We note that we do not need the condition of closed under substitution because Γ_c also admits a $\frac{1}{c}$-integral bisubmodular relaxation $(\Gamma_c)'$ and VCSP$((\Gamma_c)')$ is also in P.

In the rest of this section, we first introduce useful functions with k-submodular relaxations (Subsection 5.3.1). Then, we apply bisubmodular relaxations for re-deriving the known FPT results for problems related to Vertex Cover and giving a new FPT algorithm for Submodular Vertex Cover (Subsection 5.3.2). Finally, we apply k-submodular relaxations for re-deriving the known FPT result for Edge Multiway Cut and giving a new FPT algorithm for edge-deletion Unique Label Cover (Subsection 5.3.3).

5.3.1 Basic k-submodular Functions

Now, let us introduce useful functions with k-submodular relaxations. For an integer $k \geq 2$, let $\mathcal{B}_k$ denote the language on a domain $D = \{1, 2, \ldots, k\}$ consisting of the following cost functions (called the *basic functions*).

1. Any integral unary function $f : D \to \mathbb{N}$;
2. the soft version of a constraint $(x = \pi(y))$ for any permutation π on D;
3. the soft version of a constraint $(x = a \vee y = b)$ for $a, b \in D$.

Since $\mathcal{B}_k$ contains all integral unary functions, it is closed under substitution. As we will see in the subsequent subsections, VCSP($\mathcal{B}_k$) can naturally encode many problems.

Let $D' = \{0, 1, 2, \ldots, k\}$ be the relaxed domain. We define the half-integral discrete relaxation $\mathcal{B}'_k$ as follows.

1. For an integral unary function f, we define $f'(0) = \frac{1}{2}(f(a_1) + f(a_2))$, where $a_1 = \arg\min_{a \in D} f(a)$ and $a_2 = \arg\min_{a \in D : a \neq a_1} f(a)$.
2. For the soft version f of a constraint $(x = \pi(y))$, we define $f'(0, 0) = 0$ and $f'(a, 0) = f'(0, a) = \frac{1}{2}$ for $a \in D$.
3. For the soft version f of a constraint $(x = a \vee y = b)$, we define $f'(a, 0) = f'(0, b) = f'(0, 0) = 0$, and $f'(a', 0) = f'(0, b') = \frac{1}{2}$ for $a' \in D \setminus \{a\}$ and $b' \in D \setminus \{b\}$.

The functions $\mathcal{B}'_k$ defined above are actually k-submodular, and thus we call them the *basic k-submodular functions*.

Lemma 5.3. *$\mathcal{B}'_k$ is a k-submodular relaxation of $\mathcal{B}_k$.*

Proof. Although we can prove the lemma by straight-forward case analysis, we only provide a proof for the first case of the basic functions (any integral unary function) here. For the other two cases, we provide a program code[2] checking the inequalities, whose correctness can be checked easily than a long boring case analysis. We note that since the arity of the functions is at most two, each inequality contains at most four variables, and therefore it suffices to check the inequalities for the case of $k = 4$. We also note that, by symmetry, it suffices to check the inequalities against identity bijection for the second case and a constraint $(x = 1 \vee y = 1)$ for the third case.

Let f be a unary function and f' be the relaxation of f defined by $f'(0) = \frac{1}{2}(f(a_1) + f(a_2))$, where $a_1 = \arg\min_{a \in D} f(a)$ and $a_2 = \arg\min_{a \in D : a \neq a_1} f(a)$. Consider two values $x, y \in D'$. If it holds that $x \neq y$, $x \neq 0$, and $y \neq 0$, we have $f'(x \sqcap y) + f'(x \sqcup y) = 2f(0) \leq f'(x) + f'(y)$. Otherwise, it holds that $\{x \sqcap y, x \sqcup y\} = \{x, y\}$. Thus, f' is k-submodular. □

Since all the cost functions in $\mathcal{B}'_k$ are bounded-arity, VCSP($\mathcal{B}'_k$) is in P. Thus, via Corollary 5.1, we obtain the following corollary.

Corollary 5.3. *Any instance I of* VCSP($\mathcal{B}_k$) *can be solved in $O^*(k^{2p'})$ time, where $p' = \mathrm{OPT}(I) - \mathrm{OPT}(I')$ is the relaxation gap.*

[2] `http://www-imai.is.s.u-tokyo.ac.jp/~y.iwata/CheckKSubmodular.java`

5.3.2 Bisubmodular Relaxations

First, we show the correspondence between the half-integral LP relaxation of Vertex Cover and the half-integral bisubmodular relaxation. Let $\Gamma = \{f_0, f_\vee\}$ be the language on a domain $D = \{0, 1\}$ such that VCSP(Γ) can encode Vertex Cover (see Example 5.2). Since f_0 is the unary function and $f_\vee$ be the soft version of a constraint ($x = 1 \vee y = 1$), we have $\Gamma \subseteq \mathcal{B}_2$. Therefore, from Corollary 5.3, Vertex Cover can be solved in $O^*(2^{2p'}) = O^*(4^{p'})$ time, where p' is the relaxation gap. By defining the relaxed domain as $D' = \{0, \frac{1}{2}, 1\}$, the relaxed functions can be written as $f'_0(x) = x$ and $f'_\vee(x, y) = \max(0, 1 - x - y)$. Thus, the bisubmodular relaxation corresponds to the classical LP relaxation of Vertex Cover (see Subsection 3.1.1), and therefore the relaxation gap p' coincides with the gap by the LP relaxation. In this way, we can re-derive the known FPT result for Vertex Cover above LP by Lokshtanov et al. [71]. Although they gave the FPT result for Almost 2-SAT by a reduction to Vertex Cover above LP, we can directly re-derive the result because VCSP($\mathcal{B}_2$) can naively encode Almost 2-SAT (by using the third type of the function).

Corollary 5.4 ([71])**.** Vertex Cover Above LP *and* Almost 2-SAT *are FPT with a running time of* $O^*(4^p)$.

Because we have a value oracle minimizer for bisubmodular functions, we can use arbitrary functions admitting bisubmodular relaxations in addition to the basic functions $\mathcal{B}_2$. One of such functions is a submodular function. For a submodular function $f : \{0, 1\}^X \to \mathbb{R}$, its *Lovász extension* [73] $\hat{f}$ evaluated on a domain $D' = \{0, \frac{1}{2}, 1\}$ is defined as follows; for $A \in D'^X$, let $A_1, A_h \subseteq X$ denote the sets of variables having the value 1 and the values at least $\frac{1}{2}$, respectively; then $\hat{f}$ is defined by $\hat{f}(A) = \frac{1}{2}(f(A_1) + f(A_h))$.

Lemma 5.4. *For a submodular function* $f : \{0, 1\}^X \to \mathbb{N}$*, its Lovász extension* $\hat{f} : \{0, \frac{1}{2}, 1\}^X \to \frac{1}{2}\mathbb{N}$ *is a half-integral ($\frac{1}{2}$-rooted) bisubmodular relaxation of* f*.*

Proof. From the definition, we have

$$\begin{aligned} 2\hat{f}(A \sqcap B) &= f((A \sqcap B)_1) + f((A \sqcap B)_h) \\ &= f(A_1 \cap B_1) + f(A_h \cup B_h), \text{ and} \\ 2\hat{f}(A \sqcup B) &= f((A \sqcup B)_1) + f((A \sqcup B)_h) \\ &= f((A_1 \cup B_1) \cap (A_h \cap B_h)) + f((A_1 \cup B_1) \cup (A_h \cap B_h)) \\ &\leq f(A_1 \cup B_1) + f(A_h \cap B_h). \end{aligned}$$

Therefore, it holds that

$$\begin{aligned} 2\hat{f}(A \sqcap B) + 2\hat{f}(A \sqcup B) &\leq f(A_1 \cap B_1) + f(A_h \cup B_h) + f(A_1 \cup B_1) + f(A_h \cap B_h) \\ &\leq f(A_1) + f(B_1) + f(A_h) + f(B_h) \\ &= 2\hat{f}(A) + 2\hat{f}(B). \end{aligned}$$

□

By using the above lemma, we can extend the objective function of Vertex Cover to submodular functions.

Submodular Vertex Cover **Parameter:** p'
Input: A graph $G = (V, E)$, a submodular function $f : 2^V \to \mathbb{N}$ (given as an oracle), and an integer p'.
Question: Is there a vertex cover $C \subseteq V$ of G such that $f(C)$ is at most the relaxed minimum $\hat{f}(C^*)$ plus p'?

This problem was previously shown to have a half-integral relaxation [56]. From Lemma 5.4, the problem is also FPT.

Corollary 5.5. Submodular Vertex Cover *is FPT, with a running time of* $O^*(4^{p'})$.

5.3.3 k-submodular Relaxations

First, we show the correspondence between the half-integral LP relaxation of Edge Multiway Cut and the half-integral k-submodular relaxation.

Edge Multiway Cut **Parameter:** p
Input: An undirected graph $G = (V, E)$, a set of terminals $T \subseteq V$, and an integer p.
Question: Is there a subset of edges $C \subseteq E$ of size at most p such that each terminal in T belongs to a different connected component in $G' = (V, E \setminus C)$?

The LP relaxation of Edge Multiway Cut is as follows:

$$\begin{aligned} &\text{minimize} \sum_{e \in E} x_e \\ &\text{s.t.} \quad \textstyle\sum_{e \in P} x_e \geq 1 \quad \forall \text{path } P \text{ connecting two distinct terminals,} \\ &\qquad\ \ x_e \geq 0 \qquad\quad\ \forall e \in E. \end{aligned}$$

Garg et al. [44] proved the half-integrality of the above LP relaxation. Note that for any half-integral solution x^*, the rounding $\lceil x^* \rceil$, defined by $\lceil x^* \rceil_i = \lceil x_i^* \rceil$, is an integral solution whose value is at most the twice of the original value, and therefore the gap by the LP relaxation is upper-bounded by the half of the optimal value. Although the above half-integral LP is defined on the domain $\{0, \frac{1}{2}, 1\}^E$, we view the problem from a different point of view as follows.

Let us assume $T = [k]$. We use the terminal set T as the domain. Let $f_= : T \times T \to \mathbb{N}$ be the soft version of the constraint $(x = y)$ and $f_t : T \to \mathbb{N}$ denote the soft version of the constraint $(x = t)$ for $t \in T$. Then, for a language $\Gamma = (\{f_=\} \cup \{f_t \mid t \in T\})$, VCSP($\Gamma$) can encode Edge Multiway Cut by introducing the valued constraint $f_=(u, v)$ for each edge $uv \in E$ and the valued constraint $f_t(t)$ of a sufficiently large weight for each terminal $t \in T$. Here, variables having a value $t \in T$ correspond to the vertices in the same connected component as the terminal t, and the edges for which the constraints are not satisfied correspond to the cut.

Since $\Gamma \subseteq \mathcal{B}_k$, it admits a half-integral k-submodular relaxation and thus VCSP(Γ) can be solved in $O^*(|T|^{2p'})$, where p' is the relaxation gap. Here, by the correspondence between the variables having the relaxed value 0 and the vertices that are at distance at least $\frac{1}{2}$ from any terminals (by using the LP solution x as the edge length), the k-submodular relaxation corresponds to the LP relaxation, and therefore the relaxation gap p' coincides with the gap by the LP relaxation. In this way, we can obtain an $O^*(|T|^{2p'}) = O^*(|T|^p)$-time FPT algorithm Edge Multiway Cut.

Although the above running time is worse than the $O^*(2^p)$ time obtained by Cygan et al. [34], we can improve it to $O^*(2^p)$ time by using their branching strategy described below. We pick an edge $uv \in E$ such that the value of u is integral $t \in T$ and the value of v is 0. If there are no such edges, we can fix all the variables of value 0 to an arbitrary integral value $i \in T$ without increasing the objective value. In order to satisfy the constraint $(u = v)$, we need to assign t to v. If the optimal value does not increase by this assignment, we can safely fix it. Otherwise, we branch into two cases: either (1) fixing the value of v to t

or (2) removing the edge uv while adding a constant 1 to the objective function. Since in each case of the branching, the optimal value increases by at least $\frac{1}{2}$ (because $f_{=}(t, 0) = \frac{1}{2}$), the depth of the search tree is bounded by $2p'$. Thus, the running time is bounded by $O^*(2^{2p'}) = O^*(2^p)$. Note that this improvement is not possible in general as we will see below.

Corollary 5.6 ([34])**.** Edge Multiway Cut *is FPT with a running time of* $O^*(2^p)$.

We now focus on the problem Unique Label Cover, which is defined as follows.

Unique Label Cover	**Parameter:** p
Input: A directed graph $G = (V, E)$, where each edge $uv \in E$ is associated with a permutation π_{uv} of a constant size label set Σ, and an integer p.	
Question: Is there a labelling $\phi : V \to \Sigma$ such that the number of edges for which the constraint $\phi(u) = \pi_{uv}(\phi(v))$ is not satisfied is at most p?	

Unique Label Cover is the defining problem of the Unique Games Conjecture [63], which is of central importance to the theory of approximation. Intuitively, the Unique Games Conjecture states that for any sufficiently small $\epsilon, \delta > 0$, there exists an integer k such that, over a label set of size k, distinguishing whether there exists a labelling satisfying $(1 - \delta)$-fraction of edges or any labelling can satisfy only ϵ-fraction of edges is NP-hard. The problem was previously considered from an FPT perspective by Chitnis et al. [29], who provided an FPT algorithm with a running time of $O^*(|\Sigma|^{O(p^2 \log p)})$, using highly advanced algorithmic methods. We observe that Unique Label Cover is equivalent to VCSP(Γ) where Γ contains the soft versions of all constraints $(x = \pi(y))$ for bijection π on the label set Σ. By setting $\Sigma = [k]$, we have $\Gamma \subseteq \mathcal{B}_k$. Therefore, from Corollary 5.3, Unique Label Cover can be solved in $O^*(|\Sigma|^{2p})$ time.

Corollary 5.7. Unique Label Cover *is FPT with a running time of* $O^*(|\Sigma|^{2p})$.

Our result implies that we can determine whether there is a labelling satisfying $|E| - O(\log |E|)$ edges in polynomial time. Note that this does not contradict the Unique Games Conjecture because when $p = \delta|E|$, the running time of the algorithm becomes exponential in $|E|$. Chitnis et al. [29] showed that the problem becomes W[1]-hard when considering that the size of the label set is not a constant but depends on an input. That is, we cannot remove the factor $|\Sigma|$ from the base of the above running time under the assumption of FPT $\neq$ W[1]. We note that this does not rule out the existence of algorithms running in time like $O^*(c^p |\Sigma|^{o(p)})$.

5.4 Linear-time FPT Algorithms

In the previous section, we have shown that VCSP($\mathcal{B}_k$) can be solved in FPT time (Corollary 5.3). In this section, we improve the $n^{O(1)}$ part of the running time and obtain *linear-time* FPT algorithms by generalizing the linear-time FPT algorithm for Vertex Cover above LP presented in Chapter 3. Our goal is to prove the following theorem.

Theorem 5.1. *Any instance I of* VCSP($\mathcal{B}_k$) *can be solved in $O(k^{2p+1}m)$ time, where m is the number of valued constraints and $p = \mathrm{OPT}(I)$ is the optimal value.*

Algorithm 6 Linear-time FPT branch-and-bound via discrete relaxation

```
1: procedure Solve(I')
2:     compute an extreme minimum solution φ* of I'
3:     if f_{I'}(φ*) > p then return false
4:     for x ∈ X with φ*(x) ∈ D do
5:         I' ← I'[x ← φ*(x)]
6:     if the variable set X of I' is empty then return true
7:     pick a variable x ∈ X
8:     return ⋁_{a∈D} Solve(I'[x ← a])
```

More precisely, we can obtain a running time of $O(k^{2p'+1}m + pkm)$, where $p' = \mathrm{OPT}(I) - \mathrm{OPT}(I')$ is the relaxation gap. The latter part (pkm) is needed for computing the initial optimal solution of I'. Since we can upper-bound the gap p' by the the optimal value p, the running time in the theorem follows. From the theorem, we can obtain the following corollaries.

Corollary 5.8. Almost 2-SAT *can be solved in* $O(4^p m)$ *time.*

Corollary 5.9. Unique Label Cover *can be solved in* $O(|\Sigma|^{2p} m)$ *time.*

Let $D = \{1, 2, \ldots, k\}$ be a domain and $D' = \{0\} \cup D$ be the relaxed domain. We say that a minimum solution $\phi \in D'^X$ of a function $f' : D'^X \to \mathbb{R}$ is *dominated* by a minimum solution $\psi \in D'^X$ if $\phi \neq \psi$ and for any $x \in X$ it holds that $\phi(x) \neq 0 \Rightarrow \phi(x) = \psi(x)$. If there are no such ψ, we say that ϕ is an *extreme* minimum solution. In what follows, we prove the following lemma.

Lemma 5.5. *For any instance* I' *of* $\mathrm{VCSP}(\mathcal{B}'_k)$*, an extreme minimum solution of* I' *can be computed in* $O(\mathrm{OPT}(I')km)$ *time.*

Using the above lemma, we can prove the Theorem 5.1.

Proof of Theorem 5.1. The improved algorithm is described in Algorithm 6. In the recursive call of the algorithm, we first compute the extreme minimum solution ϕ^* of I'. From the definition, for any variable $x \in X$ with $\phi^*(x) = 0$ and for any value $a \in D$, fixing x to a together with the integral part of ϕ^* strictly increases the optimal value. Thus, in each case of the branching, the lower bound increases by at least $\frac{1}{2}$, and therefore the depth of the search tree is bounded by $2p$. Since it takes only $O(pkm)$ time for each recursive call, the total running time is bounded by $O(k^{2p+1}pm)$. We can improve the running time to $O(k^{2p+1}m)$ time by applying the strategy described in Subsection 3.2.2 (reusing the previous minimum solution before a branching to recompute the new minimum solution after the branching by searching augmenting paths in the residual graph). Since this optimization is not important to achieve linear-time FPT[3], we omit the detail here. ☐

Let $f : D'^X \to \mathbb{R}_{\geq 0}$ be a function on a domain $D' = \{0, 1, 2, \ldots, k\}$. In order to prove Lemma 5.5, we aim to express f as cuts of a network. For a variable $v \in X$, we denote a vertex set $\{v_i \mid i \in D\}$ by X_v. An (X, k)*-network* is a network on vertices $V = \bigcup_{v \in X} X_v \cup \{s, t\}$. For an assignment $\phi : X \to D'$, we define the *s-t cut corresponding to* ϕ, which is denoted by S_ϕ, as the set of

[3] Note that, in Chapter 3, the strategy was important to achieve linear-time FPT because the problems were solved by reductions to Vertex Cover and therefore the optimal value of the reduced instance is not bounded by the optimal value p of the original instance.

vertices consisting of $v_{\phi(v)}$ for each variable $v \in X$ with $\phi(v) \neq 0$ together with s. That is, $S_\phi = \{s\} \cup \{v_{\phi(v)} \mid v \in X, \phi(v) \neq 0\}$. If an s-t cut contains at most one vertex from each X_v, it is called *normalized.* Note that S_ϕ is a normalized cut for any ϕ. For a normalized cut S, we define the *assignment corresponding to* S by $\phi_S(v) = i$ if $S \cap X_v = \{v_i\}$ and $\phi_S(v) = 0$ if $S \cap X_v = \emptyset$. A normalized minimum cut S is called *dominated* by a normalized minimum cut S' if it holds that $S \subset S'$. If there are no such S', we say that S is an *extreme* minimum cut.

We say that an (X, k)-network *represents* f if for any assignment $\phi : X \to D'$, the capacity of the corresponding cut S_ϕ is equal to the value of the function $f(\phi)$. We say that a function f is *representable* if there is an (X, k)-network that represents f. For an s-t cut $S \subseteq V$, we define the *normalized cut of* S, which is denoted by $\nu(S)$, as the set of vertices consisting of $S \cap X_v$ for each variable $v \in X$ with $|S \cap X_v| = 1$ together with s. That is, $\nu(S) = \{s\} \cup \{v_i \mid v \in X, S \cap X_v = \{v_i\}\}$. We say that an (X, k)-network is *k-submodular* if for any s-t cut S, it holds that $c(S) \geq c(\nu(S))$, where c is the capacity function of the network. If there exists a k-submodular (X, k)-network that represents a function f, we say that f is *k-submodular representable.*

If every cost function in a language Γ is k-submodular representable, VCSP(Γ) can be reduced to the minimum s-t cut problem as the following two Lemmas show.

Lemma 5.6. *Let Γ be a valued constraint language. If every cost function $f \in \Gamma$ is k-submodular representable, then for any instance I of* VCSP(Γ)*, the objective function f_I is also k-submodular representable. Moreover, the number of edges of the network for f_I is bounded by the sum of the number of edges of the network for each valued constraint of I.*

Proof. We can construct the k-submodular network for f_I by creating the k-submodular network $(G_i = (V, E_i), c_i)$ for each valued constraint and then taking the sum $(G = (V, \bigcup_{i=1}^m E_i), \sum_{i=1}^m c_i)$. □

Lemma 5.7. *If a function f is k-submodular representable, then f can be minimized by computing a minimum s-t cut of the network.*

Proof. Since the network represents f, for any assignment ϕ, it holds that $c(S_\phi) = f(\phi)$. Let ϕ be a minimizer of f and S be a minimum s-t cut of the network. Because the network is k-submodular, $\nu(S)$ is also a minimum s-t cut. Therefore, $f(\phi_{\nu(S)}) = c(\nu(S)) \leq c(S_\phi) = f(\phi)$ holds. Since ϕ is a minimiser of f, $\phi_{\nu(S)}$ is also a minimizer of f. □

In order to obtain an extreme minimum solution, we prove the following one-to-one correspondence between the extreme minimum solution and the extreme minimum cut.

Lemma 5.8. *Let $f : D'^X \to \mathbb{R}_{\geq 0}$ be a function and (G, c) be a k-submodular network that represents f. Then, an assignment $\phi : X \to D'$ is an extreme minimum solution if and only if its corresponding cut S_ϕ is an extreme minimum cut.*

Proof. ($\Rightarrow$) Let S be a normalized minimum cut. If there exists a normalized minimum cut S' that dominates S, then, from the definition, it holds that $\phi_S \neq \phi_{S'}$ and $\phi_S(v) \neq 0 \Rightarrow \phi_S(v) = \phi_{S'}(v)$. Thus, ϕ_S is not an extreme minimum solution.

($\Leftarrow$) Let ϕ be a minimum solution. If there exists a minimum solution ϕ' that dominates ϕ, then, from the definition, it holds that $S_\phi \subset S_{\phi'}$. Thus, S_ϕ is not an extreme minimum cut. □

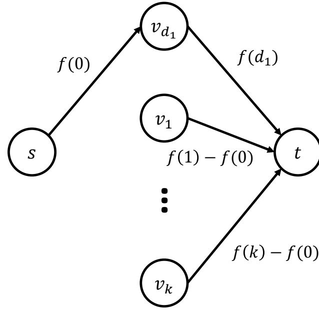

Figure 5.1: Unary $f(v)$

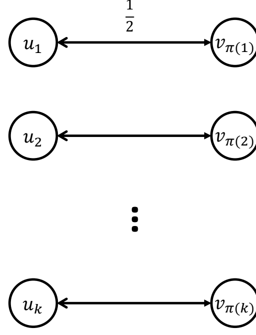

Figure 5.2: $(v = \pi(u))$

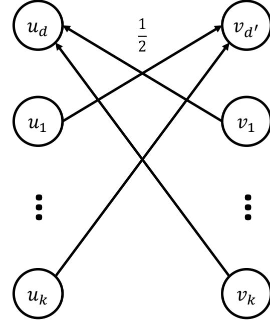

Figure 5.3: $(u = d \vee v = d')$

From the above lemma, in order to compute an extreme minimum solution, it suffices to compute an extreme minimum cut. Although the definitions are different, we can use completely the same algorithm and proof used for VERTEX COVER (Algorithm 3 in Subsection 3.2.3).

Lemma 5.9. *Given a k-submodular (X, k)-network $(G = (V, E), c)$ and its maximum flow, an extreme minimum cut of the network can be computed in $O(|V| + |E|)$ time.*

Now we show that any basic k-submodular function is k-submodular representable. For the definition of the basic k-submodular functions, please refer to Subsection 5.3.1.

Lemma 5.10. *Any unary function $f : D' \to \mathbb{R}_{\geq 0}$ is k-submodular representable.*

Proof. Let $d_1 = \arg\min_{d \in D} f(x)$. Then, we construct a $(\{v\}, k)$-network as follows (Figure 5.1):

- $c(s, v_{d_1}) = f(0)$,
- $c(v_{d_1}, t) = f(d_1)$,
- $c(v_d, t) = f(d) - f(0)$ for any $d \neq d_1$.

Note that, for $d \neq d_1$, $f(d) - f(0) \geq 0$ holds because it holds that $2f(0) \leq f(d_1) + f(d) \leq 2f(d)$.

If $\phi(v) = 0$, the capacity of the corresponding cut is $c(S_\phi) = c(s, v_{d_1}) = f(0)$. If $\phi(v) = d_1$, the capacity of the corresponding cut is $c(S_\phi) = c(v_{d_1}, t) = f(d_1)$. If $\phi(v) = d$ for $d \neq d_1$, the capacity of the corresponding cut is $c(S_\phi) = c(s, v_{d_1}) + c(v_d, t) = f(d)$. Thus the network actually represents f.

Let $D' \subseteq D$ be a set of size at least 2 and let $S = \{s\} \cup \{v_d \mid d \in D'\}$ be a cut. When D' does not contain d_1, let d_2, d_3 be distinct elements contained in D'. Then, $c(S)$ is at least $c(s, v_{d_1}) + c(v_{d_2}, t) + c(v_{d_3}, t) = f(d_2) + f(d_3) - f(0)$. Since f is k-submodular, $f(d_2) + f(d_3) \geq 2f(0)$. Therefore, $c(S) \geq f(0) = c(\nu(S))$ holds. When D' contains d_1, let d_2 be another element contained in D'. Then, $c(S)$ is at least $c(v_{d_1}, t) + c(v_{d_2}, t) = f(d_1) + f(d_2) - f(0) \geq f(0)$. Therefore, $c(S) \geq c(\nu(S))$ holds. Thus the network is actually k-submodular. □

Lemma 5.11. *For any permutation π on D, the basic k-submodular relaxation f of the soft version of a constraint $(x = \pi(y))$ is k-submodular representable.*

Proof. Let u, v be variables. We construct a $(\{u, v\}, k)$-network as follows (Figure 5.2):

- $c(u_i, v_{\pi(i)}) = \frac{1}{2}$ for any $i \in D$,
- $c(v_j, u_{\pi^{-1}(j)}) = \frac{1}{2}$ for any $j \in D$.

If $\phi(u) = \phi(v) = 0$, the capacity of the corresponding cut is $c(S_\phi) = 0 = f(\phi)$. If $\phi(u) = i \in D$ and $\phi(v) = 0$, the capacity of the corresponding cut is $c(S_\phi) = c(u_i, v_{\pi(i)}) = \frac{1}{2} = f(\phi)$. Similarly, if $\phi(u) = 0$ and $\phi(v) \neq 0$, the capacity of the corresponding cut is equal to $f(\phi)$. If $\phi(u) = i \in D, \phi(v) = j \in D$ and $j = \pi(i)$, the capacity of the corresponding cut is $c(S_\phi) = 0 = f(\phi)$. Otherwise, the capacity of the corresponding cut is $c(S_\phi) = c(u_i, v_{\pi(i)}) + c(v_j, u_{\pi^{-1}(j)}) = 1 = f(\phi)$. Thus the network actually represents f.

Let S be a cut and I, J be two sets such that $I = \{i \in D \mid u_i \in S\}$ and $J = \{j \in D \mid v_j \in S\}$. If $|I| \leq 1$ and $|J| \leq 1$, the cut S is already normalized. If $|I| = 0$ or $|I| \geq 2$, and $|J| = 0$ or $|J| \geq 2$, the capacity of the normalized cut is $c(\nu(S)) = c(\{s\}) = 0$ and the capacity of the original cut is nonnegative. Therefore, $c(S) \geq c(\nu(S))$ holds. If $I = \{i\}$ and $|J| \geq 2$, the capacity of the normalized cut is $c(\nu(S)) = c(\{s, u_i\}) = c(u_i, v_{\pi(i)}) = \frac{1}{2}$. Because π is a permutation, for at least one $j \in J$, $\pi^{-1}(j)$ is different from i. Therefore, the capacity of the original cut is at least $\frac{1}{2}$. Thus, it holds that $c(S) \geq c(\nu(S))$. Similarly, if $|I| \geq 2$ and $|J| = 1$, it holds that $c(S) \geq c(\nu(S))$. Thus, the network is actually k-submodular. □

Lemma 5.12. *For any $d, d' \in D$, the basic k-submodular relaxation f of the soft version of a constraint $(x = d \vee y = d')$ is k-submodular representable.*

Proof. Let u, v be variables. We construct a $(\{u, v\}, k)$-network as follows (Figure 5.3):

- $c(u_i, v_{d'}) = \frac{1}{2}$ for any $i \in D \setminus \{d\}$,
- $c(v_j, u_d) = \frac{1}{2}$ for any $j \in D \setminus \{d'\}$.

If $\phi(u) = \phi(v) = 0$, $\phi(u) = d$, or $\phi(v) = d'$, the capacity of the corresponding cut is $c(S_\phi) = 0 = f(\phi)$. If $\phi(u) = i \in D \setminus \{d\}$ and $\phi(v) = 0$, the capacity of the corresponding cut is $c(S_\phi) = c(u_i, v'_d) = \frac{1}{2} = f(\phi)$. Similarly, if $\phi(u) = 0$ and $\phi(v) \in D \setminus \{d'\}$, the capacity of the corresponding cut is equal to $f(\phi)$. If $\phi(u) = i \in D \setminus \{d\}, \phi(v) = j \in D \setminus \{d'\}$, the capacity of the corresponding cut is $c(S_\phi) = c(u_i, v_{d'}) + c(v_j, u_d) = 1 = f(\phi)$. Thus the network actually represents f.

Let S be a cut and I, J be two sets such that $I = \{i \in D \mid u_i \in S\}$ and $J = \{j \in D \mid v_j \in S\}$. If $|I| \leq 1$ and $|J| \leq 1$, the cut S is already normalized. If $|I| = 0$ or $|I| \geq 2$, and $|J| = 0$ or $|J| \geq 2$, the capacity of the normalized cut is $c(\nu(S)) = c(\{s\}) = 0$ and the capacity of the original cut is nonnegative. Therefore, $c(S) \geq c(\nu(S))$ holds. If $I = \{d\}$ and $|J| \geq 2$, both of the normalized cut and the original cut have the capacity zero. If $I = \{i\}$ for $i \neq d$ and $|J| \geq 2$, since J contains at least one element j which is different from d', the capacity of the original cut is at least $c(v_j, u_d) = \frac{1}{2}$. On the other hand, the capacity of the normalized cut is $c(\nu(S)) = c(u_i, v'_d) = \frac{1}{2}$. Therefore, it holds that $c(S) \geq c(\nu(S))$. Similarly, if $|I| \geq 2$ and $|J| = 1$, it holds that $c(S) \geq c(\nu(S))$. Thus, the network is actually k-submodular. □

Finally, we prove Lemma 5.5.

Proof of Lemma 5.5. We can construct a k-submodular (X, k)-network that represents the objective function $f_{I'}$ by using Lemmas 5.10–5.12. Since we create $O(k)$ edges per each valued constraint, the size of the network is $O(km)$. Because the capacity of the minimum cut of the network is equal to $\mathrm{OPT}(I')$ and each capacity is a multiple of $\frac{1}{2}$, we can compute the maximum flow of the network in $O(\mathrm{OPT}(I')km)$ time. Then, by using Lemma 5.9, we can compute an extreme minimum cut in $O(km)$ time. Finally, by using Lemma 5.8, we can obtain an extreme minimum solution. The total running time is $O(\mathrm{OPT}(I')km)$. □

5.5 Discrete Relaxations for Vertex-deletion Problems

We note that the problems we have dealt with in the previous sections are generally *edge-* or *constraint-deletion* problems (e.g., ALMOST 2-SAT and UNIQUE LABEL COVER are problems to find a minimum subset of constraints whose removal makes the remaining constraints satisfiable, and EDGE MULTIWAY CUT is a problem to find a minimum subset of edges whose removal makes the terminals separated). In this section, we extend the result to *vertex-* or *variable-deletion* problems. First, we define three problems: VARIABLE-DELETION ALMOST 2-SAT, NODE MULTIWAY CUT, and VERTEX-DELETION UNIQUE LABEL COVER.

VARIABLE-DELETION ALMOST 2-SAT **Parameter:** p
Input: A 2-CNF $\mathcal{C}$ on a variable set X and an integer p.
Question: Is there a subset of variables $S \subseteq X$ of size at most p such that the 2-CNF $\{C \in \mathcal{C} \mid C \subseteq X \setminus S\}$ becomes satisfiable?

NODE MULTIWAY CUT **Parameter:** p
Input: An undirected graph $G = (V, E)$, a set of terminals $T \subseteq V$, and an integer p.
Question: Is there a subset of vertices $S \subseteq V \setminus T$ of size at most p such that each terminal in T belongs to a different connected component in $G[V \setminus S]$?

VERTEX-DELETION UNIQUE LABEL COVER **Parameter:** p
Input: A directed graph $G = (V, E)$, where each edge $uv \in E$ is associated with a permutation π_{uv} of a constant size label set Σ, and an integer p.
Question: Is there a subset of vertices $S \subseteq V$ of size at most p such that there exists a labelling $\phi : V \setminus S \to \Sigma$ for which the constraint $\phi(u) = \pi_{uv}(\phi(v))$ is satisfied for any edge uv of $G[V \setminus S]$?

For all of these three problems, FPT algorithms with the same running time as the corresponding edge-deletion problems are known [71, 34, 29]. In Section 5.3, by using k-submodular relaxations, we re-derived the known result for ALMOST 2-SAT and EDGE MULTIWAY CUT, and also gave the improved result for UNIQUE LABEL COVER. Similarly, in this section, we will re-derive the known FPT results for VARIABLE-DELETION ALMOST 2-SAT and NODE MULTIWAY CUT, and will give an improved result for VERTEX-DELETION UNIQUE LABEL COVER.

One of the possible approaches to deal with vertex-deletion problems is as follows. For each constraint C on variables S, we introduce new variables $S_C = \{v_C \mid v \in S\}$ and replace the occurrence of a variable v in C by v_C. Then, for each variable v appearing in constraints $C_1, \ldots, C_d$, we introduce a valued constraint $(v_{C_1} = \ldots = v_{C_d})$ (a *soft wide equality*), which takes 0 if all the v_C's take an identical value and 1 otherwise. Now, with a cost of 1, each variable can take different values for different constraints, which simulates the deletion of the variable.

However, there is a problem on this approach; the soft wide equality functions

have unbounded arity, and therefore we cannot use the basic LP relaxation to solve the relaxation problem. For bisubmodular relaxations, this is acceptable, both since we have an oracle minimizer [42], and since it can be encoded by a 2-CNF with additional variables, e.g., a soft wide equality $(v_1 = \ldots = v_d)$ can be encoded as $(v_1 \to y)\wedge\ldots\wedge(v_d \to y)\wedge(y \to z)\wedge(z \to v_1)\wedge\ldots\wedge(z \to v_d)$. Thus, we can re-derive the known FPT result for VARIABLE-DELETION ALMOST 2-SAT. Moreover, by using the latter option, the problem can be encoded by VCSP($\mathcal{B}_2$), and thus we can obtain the linear-time FPT algorithm via Theorem 5.1.

Corollary 5.10. VARIABLE-DELETION ALMOST 2-SAT *can be solved in* $O(4^p m)$ *time.*

Unfortunately, for k-submodular relaxations, neither of the above two options works, and thus we will need another approach. In the joint version [59], we solved the problem by constructing a different LP. However, in this thesis, we present another approach; we encode the problems by a language on a larger domain and give a discrete relaxation which is not k-submodular.

Let $D = \{0, 1, \ldots, k\}$ be a domain, where variables having the value $0 \in D$ will correspond to the *deleted* variables, and let $\bar{D} = \{1, 2, \ldots, k\}$ denote the subdomain. We note that D is not the relaxed domain but the integral domain on which problems will be encoded. For a constraint R on $\bar{D}^r$, we redefine the *soft version* of R as the cost function $f : D^r \to \{0, 1\}$ such that $f(\boldsymbol{a}) = 0$ if $\boldsymbol{a} \notin \bar{D}^r$ or $R(\boldsymbol{a})$ is satisfied, and $f(\boldsymbol{a}) = 1$ otherwise. We define the vertex-deletion version of the basic functions, denoted by $\mathcal{V}_k$, as the language consisting of the following functions.

1. Any integral unary function $f : D \to \mathbb{N}$;
2. the soft version of a constraint $(x = \pi(y))$ for any permutation π on $\bar{D}$;
3. the soft version of a constraint $(x = a \vee y = b)$ for $a, b \in \bar{D}$.

We can observe that $\mathcal{V}_k$ can naturally encode vertex-deletion problems, e.g., VERTEX-DELETION UNIQUE LABEL COVER can be encoded by introducing the unary constraint f_0, defined by $f_0(0) = 1$ and $f_0(x) = 0$ otherwise, for each vertex $v \in V$ and the soft version of the constraint $(u = \pi_{uv}(v))$ for each edge $uv \in E$.

Let $D' = D \cup \{0', 1', \ldots, k'\}$ be the relaxed domain. We define the half-integral discrete relaxation $\mathcal{V}'_k$ as follows.

1. For the first case, we define $f'(i') = \frac{1}{2}(f(0) + f(i))$ for $i \in \bar{D}$ and $f'(0') = \frac{1}{2}(f(a_1) + f(a_2))$, where $a_1 = \arg\min_{a \in \bar{D}} f(a)$, and $a_2 = \arg\min_{a \in \bar{D}: a \neq a_1} f(a)$.
2. For the second case, we define $f'(i, j') = f'(i', j) = \frac{1}{2}$ for $i, j \in \bar{D}$ with $i \neq \pi(j)$, $f'(i, 0') = f'(0', i) = \frac{1}{2}$ for $i \in \bar{D}$, and $f'(x, y) = 0$ otherwise.
3. For the third case, we define $f'(i, j') = f'(i', j) = f'(i, 0') = f'(0', j) = \frac{1}{2}$ for $i, j \in \bar{D}$ with $i \neq a$ and $j \neq b$, and $f'(x, y) = 0$ otherwise.

Lemma 5.13. $\mathcal{V}'_k$ *is a persistent half-integral discrete relaxation of* $\mathcal{V}_k$*. Moreover,* VCSP($\mathcal{V}'_k$) *is in* P*.*

Proof. Let $\sqcup, \sqcap : D^2 \to D$ be symmetric idempotent operations defined as follows;

- $0 \sqcup j = j'$ and $0 \sqcap j = j'$ for $j \in \bar{D}$;
- $0 \sqcup 0' = 0$ and $0 \sqcap 0' = 0'$;

- $0 \sqcup j' = 0$ and $0 \sqcap j' = j'$ for $j \in \bar{D}$;
- $i \sqcup j = 0'$ and $i \sqcap j = 0'$ for $i, j \in \bar{D}$ with $i \neq j$;
- $i \sqcup 0' = i$ and $i \sqcap 0' = 0'$ for $i \in \bar{D}$;
- $i \sqcup i' = i$ and $i \sqcap i' = i'$ for $i \in \bar{D}$;
- $i \sqcup j' = i'$ and $i \sqcap j' = 0'$ for $i, j \in \bar{D}$ with $i \neq j$;
- $0' \sqcup j' = j'$ and $0' \sqcap j' = 0'$ for $j \in \bar{D}$;
- $i' \sqcup j' = 0$ and $i' \sqcap j' = 0'$ for $i, j \in \bar{D}$ with $i \neq j$.

Although we can prove that $\langle \sqcup, \sqcap \rangle$ is actually multimorphism by straight-forward case analysis, we only provide a proof for the first case of the basic functions (any integral unary function) here. Instead of giving a long boring case analysis, we provide a program code[4] checking the inequalities for the other two cases. We note that since the arity of the functions is at most two, each inequality contains at most four variables, and therefore it suffices to check the inequalities for the case of $k = 4$. We also note that, by symmetry, it suffices to check the inequalities against identity bijection for the second case and a constraint $(x = 1 \vee y = 1)$ for the third case.

Now, we show that the operations $\langle \sqcup, \sqcap \rangle$ is a multimorphism of any unary function $f' : D' \to \frac{1}{2}\mathbb{N}$ defined in the first case by the following case analysis;

- $f'(0 \sqcup j) + f'(0 \sqcap j) = f'(j') + f'(j') = f'(0) + f'(j)$ for $j \in \bar{D}$;
- $f'(0 \sqcup 0') + f'(0 \sqcap 0') = f'(0) + f'(0')$;
- $f'(0 \sqcup j') + f'(0 \sqcap j') = f'(0) + f'(j')$ for $j \in D$;
- $f'(i \sqcup j) + f'(i \sqcap j) = f'(0') + f'(0') = f'(a_1) + f'(a_2) \leq f'(i) + f'(j)$ for $i, j \in \bar{D}$ with $i \neq j$;
- $f'(i \sqcup 0') + f'(i \sqcap 0') = f'(i) + f'(0')$ for $i \in \bar{D}$;
- $f'(i \sqcup i') + f'(i \sqcap i') = f'(i) + f'(i')$ for $i \in \bar{D}$;
- $f'(i \sqcup j') + f'(i \sqcap j') = f'(i') + f'(0') = \frac{1}{2}(f'(0) + f'(i)) + \frac{1}{2}(f'(a_1) + f'(a_2)) \leq \frac{1}{2}(f'(0) + f'(i)) + \frac{1}{2}(f'(i) + f'(j)) = f'(i) + f'(j')$ for $i, j \in \bar{D}$ with $i \neq j$;
- $f'(0' \sqcup j') + f'(0' \sqcap j') = f'(j') + f'(0') = f'(0') + f'(j')$ for $j \in \bar{D}$;
- $f'(i' \sqcup j') + f'(i' \sqcap j') = f'(0) + f'(0') = f'(0) + \frac{1}{2}(f'(a_1) + f'(a_2)) \leq f'(0) + \frac{1}{2}(f'(i) + f'(j)) = f'(i') + f'(j')$ for $i, j \in \bar{D}$ with $i \neq j$.

Let $(\circ)$ be an operation defined by $a \circ b = (a \sqcup b) \sqcup b$. We can observe that $a \circ b \in D$ holds for any $a \in D$ and any $b \in D'$, and $a \circ b = b$ holds for any $a, b \in D$, which can be checked by the provided program code. Then, by the same discussion as we did in the proof of Lemma 5.2, we can prove the persistency. □

From the above lemma, we can obtain an $O^*((k+1)^{2p'})$-time algorithm for VCSP($\mathcal{V}_k$) (remind that the size of the domain is $k+1$), which is slightly worse than the edge-deletion version of $O^*(k^{2p'})$ time (Corollary 5.3). The running time can be improved by using the following *strong persistency*.

[4] `http://www-imai.is.s.u-tokyo.ac.jp/~y.iwata/CheckMultimorphism.java`

Algorithm 7 Improved algorithm for $\text{VCSP}(\mathcal{V}_k)$

```
 1: procedure SOLVE(I')
 2:     if OPT(I') > p then return false
 3:     if the variable set X of I' is empty then return true
 4:     pick a variable x ∈ X
 5:     for a ∈ D do
 6:         if OPT(I'[x ← a]) = OPT(I') then
 7:             return SOLVE(I'[x ← a])
 8:     for i ∈ D̄ do
 9:         if OPT(I'[x ← i']) = OPT(I') then
10:             return SOLVE(I'[x ← 0]) ∨ SOLVE(I'[x ← i])
11:     return ⋁_{i∈D̄} SOLVE(I' + f'_i(x))        ▷ enforce φ(x) ∈ {0, i, i'}
```

Lemma 5.14. *For any optimal assignment ϕ^* of an instance of* $\text{VCSP}(\mathcal{V}'_k)$*, there is an optimal integral assignment ϕ such that, in addition to the condition of the persistency, for each variable x with $\phi^*(x) = i'$ $(i \in \bar{D})$, it holds that $\phi(x) \in \{0, i\}$.*

Proof. The operation $(\circ)$ defined in the proof of Lemma 5.13 additionally admits the following property: for any $a \in D$ and any $j \in \bar{D}$, it holds that $a \circ j' \in \{0, j\}$. Thus, for any integral optimum $\boldsymbol{a}^* \in D^X$ and any relaxed optimum $\boldsymbol{b}^* \in D'^X$, $\boldsymbol{a}^* \circ \boldsymbol{b}^*$ becomes an integral optimum with the desired property. We can check that the operation $(\circ)$ actually admits the property by the provided program code. □

Now we prove the following lemma.

Lemma 5.15. *Any instance I of* $\text{VCSP}(\mathcal{V}_k)$ *can be solved in $O^*(k^{2p'})$ time, where $p' = \text{OPT}(I) - \text{OPT}(I')$ is the relaxation gap.*

Proof. The improved algorithm is described in Algorithm 7. In each recursive call, we pick a variable x and try to fix it to an integral value (line 7). If we failed, we check the existence of an optimal solution for which the variable x takes a value i' for some value $i \in \bar{D}$. If there exists such a value i', from the strong persistency, there exists an optimal integral solution for which the variable x takes a value 0 or i. Thus, we can branch into the two cases (line 10). In each case, the lower bound increases by at least $\frac{1}{2}$.

If there exists no such a value, the variable x has the value $0'$ for any optimal solutions. For a value $i \in \bar{D}$, let $f'_i : D' \to \frac{1}{2}\mathbb{N}$ denote the half-integral discrete relaxation of the soft version of the constraint $(x = i)$. Note that it holds that $f'_i(0) = f'_i(i) = f'_i(i') = 0$ and $f'_i(a) \geq \frac{1}{2}$ for any other $a \in D'$. Thus, by adding a valued constraint $f'_i(x)$ of a sufficiently large weight, we can limit the domain of x to values $\{0, i, i'\}$. We denote by $I' + f'_i(x)$ the instance obtained by adding the valued constraint $f'_i(x)$. Then, we branch into $|\bar{D}| = k$ cases: $I' + f'_i(x)$ for each $i \in \bar{D}$ (line 11). In each case, the lower bound increases by at least $\frac{1}{2}$.

Thus, the depth of the search tree is bounded by $2p'$, and at each recursive call, we branch into at most k cases. Therefore, the running time is bounded by $O^*(k^{2p'})$. □

From the above lemma, we can re-derive the known FPT result for Node Multiway Cut (by some extra work discussed in Subsection 5.3.3) and obtain an improved FPT algorithm for vertex-deletion Unique Label Cover.

Corollary 5.11 ([34]). Node Multiway Cut *is FPT with a running time of* $O^*(2^p)$.

Corollary 5.12. vertex-deletion Unique Label Cover *is FPT with a running time of* $O^*(|\Sigma|^{2p})$.

Chapter 6

Equivalence among Problems of Bounded Width

In this chapter, we introduce our new reduction technique called *decomposition-based reduction* and show equivalence among various problems. First, in Section 6.1, we provide useful lemmas for bounding the tree-width, which are used in the many proofs in this chapter. Then, in Section 6.2, we explain the basic idea of decomposition-based reductions. We prove the equivalence among problems of bounded tree-width in Section 6.3 and the equivalence of Independent Set parameterized by tree-width and clique-width in Section 6.4. In Section 6.5, we give a tree-width preserving reduction from #Perfect Matching to #SAT. Finally, in Section 6.6, we prove the equivalence among problems parameterized by branch-width. The definitions of tree-width, clique-width, and branch-width can be found in Sections 2.2, 6.4, and 6.6, respectively.

We note that since we are considering decision problems in this chapter, for optimization problems such as Vertex Cover, we assume that an integer k is given as a part of the input and the task is to determine whether the objective value is at most (or at least for maximization problems) k. In our reductions, we will often use a binary representation of an integer. Let $\{a_1, a_2, \ldots, a_M\}$ be Boolean variables. We denote the integer $\sum_{i\in[M],a_i=\mathrm{true}} 2^{i-1}$ by $(a_1a_2\ldots a_M)_2$, or $(a_*)_2$ for short. For readability, we will frequently use (arithmetic) constraints such as $(a_*)_2 = (b_*)_2 + (c_*)_2$. Note that any arithmetic constraint on M variables can be trivially simulated by at most 2^M M-clauses. Thus, if M is logarithmic in the input size, the number of required clauses is polynomial in the input size.

6.1 Useful Lemmas for Bounding Tree-width

For a vertex $v \in V$, let G/v denote the graph obtained by removing v and making the neighbors of v form a clique. We call this operation *eliminating* v. Similarly, for a subset $S \subseteq V$, we denote by G/S the graph obtained by removing S and making the neighbors of S form a clique.

In order to prove the upper bound on tree-width, we will often use the following lemma.

Lemma 6.1 (Arnborg [7]). *For a graph $G = (V, E)$ and a vertex $v \in V$, $\mathbf{tw}(G) \le \max(d(v), \mathbf{tw}(G/v))$. Moreover, if we are given a tree-decomposition of G/v of width w, we can construct a tree-decomposition of G of width $\max(d(v), w)$ in linear time.*

Proof. Let $T = (I, F)$ be a tree-decomposition of G/v of width w. Since the neighbors $N(v)$ form a clique in G/v, there exists a node $i \in I$ such that the bag X_i contains $N(v)$. Therefore, by creating a node j with $X_j = N[v]$ and

adding an arc ij, we can obtain a tree-decomposition of G. The width of this tree-decomposition is $\max(|X_j| - 1, w) = \max(d(v), w)$. □

We extend the above lemma to subsets.

Lemma 6.2. *For a graph $G = (V, E)$ and a vertex subset $S \subseteq V$, $\mathbf{tw}(G) \leq \max(|N[S]| - 1, \mathbf{tw}(G/S))$.*

Proof. Let $S = \{v_1, ..., v_k\}$. We eliminate each vertex of S one by one. We denote the graph after the i-th elimination by $G_i = ((G/v_1)/v_2) \dots /v_i$. By eliminating v_i from G_{i-1}, we obtain $\mathbf{tw}(G_{i-1}) \leq \max(d_{G_{i-1}}(v_i), \mathbf{tw}(G_i))$. Since $N_{G_{i-1}}(v_i) \subseteq N_G[S] \setminus \{v_i\}$, we have $\mathbf{tw}(G) = \mathbf{tw}(G_0) \leq \max(|N[S]| - 1, \mathbf{tw}(G_k))$. Because G_k is a subgraph of G/S and any tree-decomposition of a graph is also a tree-decomposition of its subgraph, we obtain $\mathbf{tw}(G) \leq \max(|N[S]| - 1, \mathbf{tw}(G/S))$. □

Lemma 6.3. *Let X and Y be disjoint vertex sets of a graph G such that for each vertex $x \in X$, $|N(x) \cap Y| \leq 1$. Then, $\mathbf{tw}(G) \leq \max(|N[X] \setminus Y|, \mathbf{tw}(G/X))$.*

Proof. Let $X = \{x_i \mid i \in [k]\}$ and $U = N(X) \setminus Y$. For an integer i, we denote the vertex set $\{x_j \mid j \in [i]\}$ by X_i. We eliminate each vertex of X one by one. After eliminating vertices X_{i-1}, x_i can be adjacent only to vertices in $(X \setminus X_i) \cup \{Y \cap N(X_i)\} \cup U$. Since $|Y \cap N(X_i)| \leq i$, we have $d(x_i) \leq |X| - i + i + |U| = |X| + |U| = |N[X] \setminus Y|$. By iteratively applying Lemma 6.1, we obtain $\mathbf{tw}(G) \leq \max(|N[X] \setminus Y|, \mathbf{tw}(G/X))$. □

Lemma 6.4. *Let $\{S_i \mid i \in [d]\}$ be a family of disjoint vertex sets of a graph G such that each set has size at most k and there are no edges between S_i and S_j for any $|i - j| > 1$. Then, $\mathbf{tw}(G) \leq \max(2k + |N(S)| - 1, \mathbf{tw}(G/S))$, where $S = \bigcup_{i \in [d]} S_i$.*

Proof. Let $U = N(S)$. We eliminate each vertex set S_i one by one. After eliminating vertex sets $\{S_j \mid j \in [i-1]\}$, it holds that $N(S_i) \subseteq S_{i+1} \cup U$. Thus, we have $|N[S_i]| \leq 2k + |U|$. By iteratively applying Lemma 6.2, we obtain $\mathbf{tw}(G) \leq \max(2k + |N(S)| - 1, \mathbf{tw}(G/S))$. □

For a vertex set S, if we can obtain $\mathbf{tw}(G) \leq \max(d, \mathbf{tw}(G/S))$ by applying one of these lemmas, we say that the elimination has *degree* d. If we can reduce a graph G into a graph G' by a series of eliminations of degree at most d, we can obtain $\mathbf{tw}(G) \leq \max(d, \mathbf{tw}(G'))$.

6.2 Overview of Decomposition-based Reductions

We explain the basic idea of decomposition-based reductions. Although we deal with four different decompositions in this thesis (tree-, clique-, branch-, and path-decompositions), the basic idea is the same. We believe that the same idea can be used to many other decompositions.

A decomposition can be seen as a collection of sets forming a tree. For example, tree-decomposition is a collection of *bags* forming a tree, clique-decomposition is a collection of *labels* forming a tree, and branch-decomposition is a collection of *middle sets* forming a tree. First, for each node i of a decomposition tree, we create gadgets as follows: (1) for each element x in the corresponding set X_i, create a *path-like* gadget x_i that expresses the *state* of the element (e.g., the value of the variable x for the case of SAT), and (2) create several gadgets to solve *subproblem* corresponding to this node (e.g., simulate clauses inside X_i for

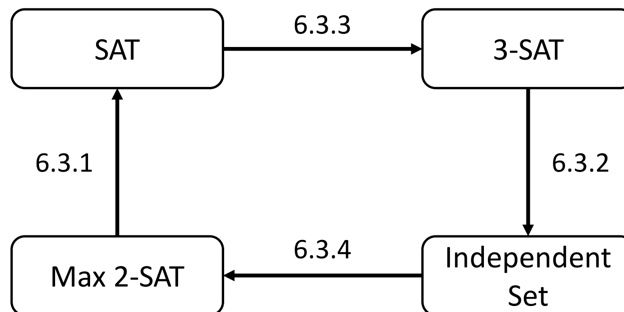

Figure 6.1: The cycle reductions

the case of SAT). Then, for each node c, its parent p, and each common element $x \in X_c \cap X_p$, by connecting the tail of x_c and the head of x_p, we establish *local consistency*. From the definition of the decomposition, this leads to *global consistency*. Since the obtained graph has a *locality*, it has a small width. We may need additional tricks to establish local consistency without increasing the width.

6.3 Equivalence among Problems of Bounded Tree-width

In this section, we prove the following theorem.

Theorem 6.1. *For any $\alpha > 0$, the following are equivalent:*

1. SAT *can be solved in $O^*(2^{\alpha \mathbf{tw}})$ time,*
2. 3-SAT *can be solved in $O^*(2^{\alpha \mathbf{tw}})$ time,*
3. Max 2-SAT *can be solved in $O^*(2^{\alpha \mathbf{tw}})$ time, and*
4. Independent Set *can be solved in $O^*(2^{\alpha \mathbf{tw}})$ time,*

where **tw** *is the width of the given tree-decomposition.*

The proof consists of cycle reductions shown in Figure 6.1. First, we give a tree-decomposition-based reduction from Max 2-SAT to SAT in Subsection 6.3.1. The reduction is rather simple but contains an essential idea of decomposition-based reductions. In Subsection 6.3.2, we give a tree-decomposition-based reduction from 3-SAT to Independent Set. The reduction contains an interesting trick for bounding the tree-width, which may be useful for other reductions. The standard reductions from SAT to 3-SAT and Independent Set to Max 2-SAT can preserve the tree-width, which are presented in Subsections 6.3.3 and 6.3.4.

6.3.1 Reduction from Max 2-SAT to SAT

Let $(X, \mathcal{C} = \{C_1, \ldots, C_m\}, k)$ be an instance of Max 2-SAT. We want to construct an instance $(X', \mathcal{C}')$ of SAT such that $\mathcal{C}'$ is satisfiable if and only if at least k clauses of $\mathcal{C}$ can be satisfied. Let $M = \lceil \log(m+1) \rceil$. In the following reductions, we will use arithmetic constraints on $O(M)$ variables, which can be simulated by poly(m) clauses.

First, we explain a naive reduction from Max 2-SAT to SAT that does not preserve tree-width. For each clause $C_i = (x \vee y) \in \mathcal{C}$, we create a new variable

w_i and a constraint $w_i \Leftrightarrow (x \vee y)$. The variable w_i represents whether the clause C_i is satisfied. For each $j \in [M]$, we create a variable $s_{0,j}$ and a clause $(\overline{s_{0,j}})$. For each $i \in [m]$ and $j \in [M]$, we create a variable $s_{i,j}$. Then, we will insert clauses so that $(s_{i,*})_2$ represents the number of satisfied clauses in $\{C_1, C_2, \ldots, C_i\}$. This can be done by inserting a constraint of $(s_{i,*})_2 = (s_{i-1,*})_2 + (w_i)_2$. Finally, we create a constraint that $(s_{m,*})_2 \geq k$. Now, we have obtained an instance $\mathcal{C}'$ of polynomial size. We can easily check that $\mathcal{C}'$ is satisfiable if and only if at least k clauses of $\mathcal{C}$ can be satisfied.

Let $\mathbf{tw}$ be the tree-width of $\mathcal{C}$. We want to show that $\mathcal{C}'$ has tree-width at most $\mathbf{tw}+O(\log m)$. We note that the additive $O(\log m)$ factor is allowed because $O^*(2^{\alpha(\mathbf{tw}+O(\log m))}) = O^*(2^{\alpha\mathbf{tw}}\text{poly}(m)) = O^*(2^{\alpha\mathbf{tw}})$. Unfortunately, however, we cannot obtain such an upper bound for the naive reduction above. This is because the variables w_i's break the structure of the tree-decomposition and blow up the tree-width of the resulting graph. To resolve this issue, we determine the order of adding w_i's based on the tree-decomposition.

Now, we introduce a tree-decomposition-based reduction. Let $T = (I, F)$ be a given nice tree-decomposition of width $\mathbf{tw}$. We will create an instance of SAT whose tree-width is at most $\mathbf{tw} + O(\log m)$. For each node $i \in I$, we create variables $\{x_i \mid x \in X_i\} \cup \{s_{i,j} \mid j \in [M]\} \cup \{w_i\}$. The value $(s_{i,*})_2$ will represent the number of satisfied clauses in the subtree rooted at i. For each node i and its parent p, we create a constraint $x_i = x_p$ for each variable $x \in X_i \cap X_p$. Because the nodes containing the same variable form a connected subtree in T, these constraints ensure that for any variable $x \in X$, all the variables $\{x_i \mid x \in X_i\}$ take the same value. For each node i, according to its type, we do as follows:

1. Leaf: create a clause $(\overline{s_{i,j}})$ for each $j \in [M]$.
2. Introduce(v): create a constraint $s_{i,j} = s_{c,j}$ for each $j \in [M]$.
3. Introduce($x \vee y$): create a constraint $w_i \Leftrightarrow (x_i \vee y_i)$ and a constraint $(s_{i,*})_2 = (s_{c,*})_2 + (w_i)_2$.
4. Forget(v): create a constraint $s_{i,j} = s_{c,j}$ for each $j \in [M]$.
5. Join: create a constraint $(s_{i,*})_2 = (s_{l,*})_2 + (s_{r,*})_2$.

Finally for the root node r, we create a constraint $(s_{r,*})_2 \geq k$. Now, we have obtained an instance $(X', \mathcal{C}')$ of polynomial size. We note that, from the definition of a nice tree-decomposition, there exists exactly one Introduce(C) node for each clause $C \in \mathcal{C}$. Thus, the sum $\sum_{i \in I}(w_i)_2$, which is equal to $(s_{r,*})_2$, represents the number of satisfied clauses. Therefore, $\mathcal{C}'$ is satisfiable if and only if at least k clauses of $\mathcal{C}$ can be satisfied. Finally, we show that the reduction preserves the tree-width.

Lemma 6.5. *$\mathcal{C}'$ has tree-width at most* $\mathbf{tw} + O(\log m)$*.*

Proof. We will prove the bound by reducing the primal graph of $\mathcal{C}'$ into an empty graph by a series of eliminations of degree at most $\mathbf{tw}+O(\log m)$. For a node i, let Y_i denote the vertex set $\{x_i \mid x \in X_i\}$ and V_i denote the vertex set $Y_i \cup \{w_i\} \cup \{s_{i,j} \mid j \in [M]\}$. Starting from the primal graph of $\mathcal{C}'$ and the given tree-decomposition T of $\mathcal{C}$, we eliminate the vertices as follows. First, we choose an arbitrary leaf i of T. Then, we eliminate all the vertices of V_i in a certain order, which will be described later. Finally, we remove i from T and repeat the process until T becomes empty.

Let i be a leaf and p be its parent. If i is the only child of p, we have $N(V_i) \subseteq V_p$. Thus, the eliminations of V_i can create edges only inside V_p. If p has another child q, we have $N(V_i) \subseteq V_p \cup \{s_{q,j} \mid j \in [M]\}$. Thus, the eliminations of V_i can create edges only inside $V_p \cup \{s_{q,j} \mid j \in [M]\}$. Therefore, after processing each node, we can ensure that the edges created by previous eliminations are only inside $V_i \cup \{s_{c,j} \mid c$ is a child of i and $j \in [M]\}$ for each node i.

Now, we describe the details of the eliminations. Let i be the current node to process. If i is the root, the number of remaining vertices is $O(\log m)$. Thus, the elimination of these vertices has degree $O(\log m)$. Otherwise, let p be the parent of i. First, we eliminate the vertices Y_i. Because each vertex of Y_i is adjacent to at most one vertex of Y_p, Lemma 6.3 gives the elimination of degree $|N[Y_i] \setminus Y_p| \leq |V_i| \leq \mathbf{tw} + O(\log m)$. Then, we eliminate the remaining vertices $V_i \setminus Y_i$. If i is the only child of p, let $V_q = Y_q = \emptyset$, and otherwise, let q be the another child of p. By applying Lemma 6.2, we obtain the elimination of degree $|N[V_i \setminus Y_i]| - 1 \leq |V_i \setminus Y_i| + |V_p| + |V_q \setminus Y_q| \leq \mathbf{tw} + O(\log m)$. □

6.3.2 Reduction from 3-SAT to Independent Set

Let $(X, \mathcal{C} = \{C_1, \ldots, C_m\})$ be an instance of 3-SAT. We want to construct an instance (G, k) of Independent Set with essentially the same tree-width such that G has an independent set of size at least k if and only if $\mathcal{C}$ is satisfiable. Actually, in our reductions, we choose k so that any independent set has size at most k. In the following reductions, we will use two gadgets depicted in Figure 6.2.

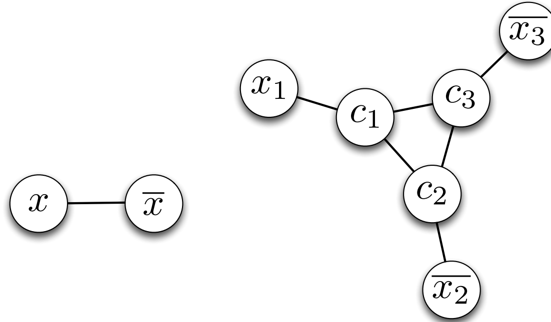

Figure 6.2: The variable gadget for a variable x and the clause gadget for a clause $(\overline{x_1} \vee x_2 \vee x_3)$

A *variable gadget* of a variable x consists of two vertices x and $\overline{x}$ connected by an edge. Any independent set can contain at most one of x and $\overline{x}$. By choosing k properly, we ensure that any independent set of size k contains exactly one of them. This gadget will represent whether a variable x is assigned true (the vertex x is in the independent set) or false (the vertex $\overline{x}$ is in the independent set).

A *clause gadget* of a clause $C = (x_1 \vee x_2 \vee \ldots \vee x_d)$ consists of d vertices $\{c_i \mid i \in [d]\}$ forming a clique ($x_1, \ldots, x_d$ are literals rather than variables). By choosing k properly, we ensure that any independent set of size k contains exactly one of them. We call the operation of creating a clique $\{c_i \mid i \in [d]\}$ and inserting edges $\{c_i\overline{x_i} \mid i \in [d]\}$ *creating a clause gadget C*. If an independent set contains one vertex from the clause gadget, at least one of the vertices $\{\overline{x_i} \mid i \in [d]\}$ are not in the independent set. By our choice of k, we ensure that at least one of $\{x_i \mid i \in [d]\}$ must be in the independent set. Therefore, it acts as a clause C.

First, we explain a naive reduction from 3-SAT to Independent Set that

does not preserve tree-width. For each variable $x \in X$, we create a corresponding variable gadget, and for each clause $(x \vee y \vee z) \in \mathcal{C}$, we create a corresponding clause gadget. Finally, we set k as the number of variable gadgets plus the number of clause gadgets. Now, we have obtained an instance (G, k) of Independent Set. From our choice of k, if G contains an independent set of size k, it must contain exactly one vertex from each variable gadget and clause gadget. Therefore $\mathcal{C}$ is satisfiable. Conversely, if $\mathcal{C}$ is satisfiable, we can construct an independent set of size k by choosing an appropriate vertex from each gadget.

Let $\mathbf{tw}$ be the tree-width of $\mathcal{C}$. We omit the proof but the above naive reduction increases the tree-width of G to $2\mathbf{tw} + O(1)$. This is because, instead of a single variable x, we need to keep two vertices x and $\overline{x}$ of the variable gadget in a bag. Intuitively, in order to preserve tree-width, we can put only one of x and $\overline{x}$ in a bag. Our solution is forgetting and remembering the state of x and $\overline{x}$ along the tree-decomposition.

Now, we explain our tree-decomposition-based reduction. Let $M = \lceil \log(\mathbf{tw} + 2) \rceil$. We will construct a graph with tree-width at most $\mathbf{tw} + O(\log \mathbf{tw})$. As we discussed before, the additive $O(\log \mathbf{tw})$ factor is allowed. Let $T = (I, F)$ be a given nice tree-decomposition of width $\mathbf{tw}$. For each node $i \in I$, we create a variable gadget for each of $\{x_i \mid x \in X_i\}$. If i is an Introduce$(x \vee y \vee z)$ node, we create a clause gadget for $(x_i \vee y_i \vee z_i)$. If i is not the root, let p be its parent and P_i be the set $X_i \cap X_p$. Then, for each $x \in P_i$, we connect $\overline{x_i}$ and x_p by an edge. We want to ensure that for any independent set S of size k, x_i is in S if and only if x_p is in S. If $\overline{x_i}$ is in S, x_p cannot be in S, and therefore $\overline{x_p}$ must be in S. On the other hand, even if x_i is in S, $\overline{x_p}$ can be in S. In order to avoid such a situation, we will create a gadget to count the number of vertices in $(\{x_i \mid x \in P_i\} \cup \{\overline{x_p} \mid x \in P_i\}) \cap S$ (this is the most interesting part of our reduction). Because $x_i \notin S$ implies $\overline{x_p} \in S$, the number is always at least $|P_i|$, and if (and only if) the number is exactly $|P_i|$, it holds that $x_i \in S \Leftrightarrow x_p \in S$ for any $x \in P_i$. Since the nodes containing the same variable form a connected subtree, this ensures that for any independent set of size k and for any variable x, all the vertices $\{x_i \mid x \in X_i\}$ are in S or none of them are in S. By using the binary encoding, the number can be expressed by $O(\log \mathbf{tw})$ variables. Thus, we can make the gadget to increase the tree-width only by $O(\log \mathbf{tw})$.

We will construct such a gadget by using the following gadget. Let $U = \{u_1, \ldots, u_d\}$ be a set of vertices. A *counting gadget of* U consists of the following $d+1$ layers of variable gadgets connected by clause gadgets. For each $a \in [d]$, the a-th layer consists of a variable gadget for y_a and variable gadgets for each of $\{s_{a,j} \mid j \in [M]\}$. The last layer consists of variable gadgets for each of $\{s_{d+1,j} \mid j \in [M]\}$. Then, for each $j \in [M]$, we create a clause gadget for $(\overline{s_{1,j}})$, and for each $a \in [d]$, we create clause gadgets simulating an arithmetic constraint $(s_{a+1,*})_2 = (s_{a,*})_2 + (y_a)_2$. Finally, for each $a \in [d]$, we connect u_a and $\overline{y_a}$ by an edge. For an independent set S, the number $(s_{d+1,*})_2$ in the last layer represents the size of $\{y_a \mid a \in [d]\} \cap S$. Since $u_a \in S$ implies $y_a \in S$, the number is at least the size of $U \cap S$.

Now, we construct the gadget (see Figure 6.3). First, we construct a counting gadget for the set $\{x_i \mid x \in P_i\}$, called a *child counting gadget for* i. Then, we construct a counting gadget for the set $\{\overline{x_p} \mid x \in P_i\}$, called a *parent counting gadget for* i. Finally, we create clause gadgets simulating the arithmetic constraint that the sum of the numbers represented by the last layers of these two counting gadgets must be at most $|P_i|$. As we discussed before, the size $|(\{x_i \mid x \in P_i\} \cup \{\overline{x_p} \mid x \in P_i\}) \cap S|$ is always at least $|P_i|$ and becomes exactly $|P_i|$ if and only if $x_i \in S \Leftrightarrow x_p \in S$ holds for any $x \in P_i$. Since the sum is at least the size

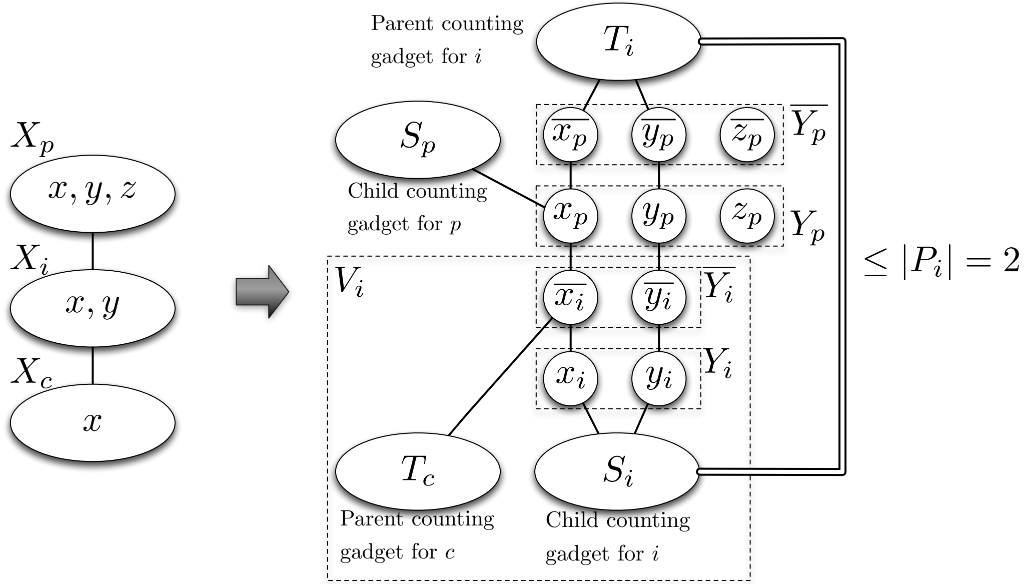

Figure 6.3: Reduction from 3-SAT to Independent Set

$|(\{x_i \mid x \in P_i\} \cup \{\overline{x_p} \mid x \in P_i\}) \cap S|$, the constraint that the sum is at most $|P_i|$ implies that $x_i \in S \Leftrightarrow x_p \in S$ for any $x \in P_i$.

Now, we have obtained a graph G of polynomial size and we set k as the number of variable gadgets plus the number of clause gadgets. From our construction, for any independent set S of size k and a variable $x \in X$, all the vertices $\{x_i \mid i \in I \text{ s.t. } x \in X_i\}$ are in S or none of them are in S. Thus, if G has an independent set of size k, $\mathcal{C}$ is satisfiable. Conversely, if $\mathcal{C}$ is satisfiable, by taking an appropriate vertex from each gadget, we can obtain an independent set of size k. Finally, we show that the reduction preserves the tree-width.

Lemma 6.6. *G has tree-width at most* $\mathbf{tw} + O(\log \mathbf{tw})$.

Proof. We will prove the bound by reducing G into an empty graph by a series of eliminations of degree at most $\mathbf{tw} + O(\log \mathbf{tw})$. Starting from G and the given tree-decomposition T of $\mathcal{C}$, we eliminate the vertices as follows.

First, for each clause gadget other than the clause gadgets for $C \in \mathcal{C}$ (created when processing the Introduce(C) node), we eliminate its vertices S. Since the size of $N[S]$ is $O(\log \mathbf{tw})$ and no two vertices in different clause gadgets are adjacent, from Lemma 6.2, we obtain eliminations of degree $O(\log \mathbf{tw})$.

For a node $i \in I$, let Y_i and $\overline{Y_i}$ denote the vertex sets $\{x_i \mid x \in X_i\}$ and $\{\overline{x_i} \mid x \in X_i\}$, respectively. If i is an Introduce node, then let C_i denote the set of vertices in the corresponding clause gadget, and otherwise, let C_i be an empty set. If i is not the root and has a parent p, let $S_{i,a}$ be the set of vertices in the variable gadgets of the a-th layer of the child counting gadget for i. If i is the root, we set $S_{i,a}$ as an empty set. We denote the set of all the vertices of the child counting gadget by $S_i = \bigcup_{a \in [d+1]} S_{i,a}$, where $d = |X_i \cap X_p|$. Similarly, let $T_{i,a}$ be the set of vertices in the variable gadgets of the a-th layer of the parent counting gadget for i and $T_i = \bigcup_{a \in [d+1]} T_{i,a}$. Let V_i denote the union of Y_i, $\overline{Y_i}$, C_i, S_i, and T_c for each child c of i. Now, we eliminate each V_i as follows.

First, we choose an arbitrary leaf i of the tree T. Then, we eliminate all the vertices of V_i in a certain order, which will be described later. Finally, we remove i from T and repeat the process until T becomes empty.

Since $N(V_i) \subseteq Y_p \cup T_{i,d+1}$ holds for a leaf i and its parent p, where $d = |X_i \cap X_p|$, the eliminations of V_i can create edges only within $Y_p \cup T_{i,d+1}$. Thus, after processing each node, we can ensure that the edges created by previous eliminations only connect vertices in the same vertex set $Y_p \cup T_{i,d+1}$ for some node i, its parent p, and $d = |X_i \cap X_p|$.

Now, we describe the details of the eliminations. Let i be the current node to process. If i is the root, the number of remaining vertices is $O(\log \mathbf{tw})$. Thus, the elimination of these vertices has degree $O(\log \mathbf{tw})$. Otherwise, let p be the parent of i, $d = |X_i \cap X_p|$, and J be the set of children of i in the original tree-decomposition. We note that from the definition of nice tree-decompositions, the size of J is at most two. First, we eliminate S_i. Since there are no edges between $S_{i,a}$ and $S_{i,b}$ for any $|a-b| > 1$, from Lemma 6.4, the elimination has degree $2(M+2) + |N(S_i)| - 1 = O(\log \mathbf{tw}) + |Y_i \cup T_{i,d+1}| \leq \mathbf{tw} + O(\log \mathbf{tw})$. Then, we eliminate Y_i. Note that each vertex $x_i \in Y_i$ can be adjacent only to the vertex $\overline{x_i} \in \overline{Y_i}$, vertices in $Y_i \cup C_i \cup T_{i,d+1}$ (as we have eliminated S_i), and vertices in $T_{c,|X_c \cap X_i|+1}$ for a child $c \in J$ (as $\overline{x_c}$ is adjacent to x_i and the path $T_{c,|X_c \cap X_i|+1}$-S_c-x_c-$\overline{x_c}$ is eliminated when processing c). Hence by Lemma 6.3, the elimination has degree $|N[Y_i] \setminus \overline{Y_i}| \leq |Y_i \cup C_i \cup T_{i,d+1}| + \sum_{c \in J} |T_{c,|X_c \cap X_i|+1}| \leq \mathbf{tw} + O(\log \mathbf{tw})$. Next, we eliminate C_i. From Lemma 6.2, the elimination has degree $N[C_i] - 1 \leq 5 + |\overline{Y_i} \cup T_{i,d+1}| + \sum_{c \in J} |T_{c,|X_j \cap X_i|+1}| \leq \mathbf{tw} + O(\log \mathbf{tw})$. Then, for each child $c \in J$, we eliminate T_c. Since there are no edges between $T_{c,a}$ and $T_{c,b}$ for any $|a-b| > 1$, from Lemma 6.4, the elimination has degree $2(M+2) + |N(T_c)| - 1 = O(\log \mathbf{tw}) + |\overline{Y_i} \cup T_{i,d+1}| + \sum_{j \in J} |T_{j,|X_j \cap X_i|+1}| \leq \mathbf{tw} + O(\log \mathbf{tw})$. Finally, we eliminate $\overline{Y_i}$. Since each vertex $\overline{x_i} \in \overline{Y_i}$ can be adjacent only to the vertex $x_p \in Y_p$ and vertices in $\overline{Y_i} \cup T_{i,d+1}$, from Lemma 6.3, the elimination has degree $|N[\overline{Y_i}] \setminus Y_p| \leq |\overline{Y_i} \cup T_{i,d+1}| \leq \mathbf{tw} + O(\log \mathbf{tw})$. □

6.3.3 Reduction from SAT to 3-SAT

Let $(X, \mathcal{C} = \{C_1, \ldots, C_m\})$ be an instance of SAT and $\mathbf{tw}$ be its tree-width. We can use the standard reduction from SAT to 3-SAT: replacing each clause $(x_1 \vee \ldots \vee x_k)$ with clauses $(x_1 \vee x_2 \vee y_1), (\overline{y_1} \vee x_3 \vee y_2), \ldots, (\overline{y_{k-3}} \vee x_{k-1} \vee x_k)$.

Now, we show that the tree-width of the obtained 3-CNF is at most $\mathbf{tw} + 2$. For each clause $(x_1 \vee \ldots \vee x_k)$ of the original CNF, we have created $k-3$ new variables $Y = \{y_1, y_2, \ldots, y_{k-3}\}$. Let $S_i = \{y_i\}$ for $i \in [k-3]$. Since there is no edge between S_i and S_j for $|i-j| > 1$, from Lemma 6.4, we obtain the elimination of Y of degree $2 \times 1 + |N(S)| - 1 = k + 1$. Since variables in the same clause form a clique in the primal graph, we have $\mathbf{tw} \geq k - 1$. Thus, the elimination has degree at most $\mathbf{tw} + 2$. After applying the above elimination to all the clauses, the graph coincides with the primal graph of $\mathcal{C}$. Therefore, the tree-width of the obtained 3-CNF is at most $\mathbf{tw} + 2$.

6.3.4 Reduction from Independent Set to Max 2-SAT

Let $(G = (V, E), k)$ be an instance of Independent Set. We use the following naive reduction to make an instance $(X', \mathcal{C}', k')$ of Max 2-SAT.

For each vertex $v \in V$, we create a variable x_v and add a clause (x_v) of length one. This variable represents whether a vertex v is in an independent set or not. Then, for each edge $uv \in E$, we create $|V| + 1$ copies of a clause $(\overline{x_u} \vee \overline{x_v})$. This clause simulates the constraint that at most one of u and v can be in an independent set. Finally, we set $k' = |E|(|V| + 1) + k$.

If there exists an independent set S of size at least k, we can satisfy at least

k' clauses by setting $x_v = \text{true}$ if and only if $v \in S$. If there exists an assignment that satisfies k' clauses, it must satisfy all the constraints $(\overline{x_u} \vee \overline{x_v})$. Thus, we can construct an independent set S of size at least k by taking $v \in S$ if and only if $x_v = \text{true}$. Because the primal graph of the obtained CNF $\mathcal{C}'$ is completely the same as the original graph G, they have the same tree-width.

6.4 Equivalence between Tree-width and Clique-width

We first define the notion of clique-width formally. The *clique-decomposition* of a graph G is an algebraic expression constructing G by means of the following four operations.

- Creation of a vertex v with a label i (denoted by $i(v)$).
- Disjoint union of two labeled graphs G and H (denoted by $G \oplus H$).
- Joining each vertex with label i to each vertex with label j, where $i \neq j$ (denoted by $\eta_{i,j}$).
- Renaming label i to label j (denoted by $\rho_{i \to j}$).

Every graph can be defined by an algebraic expression using these four operations. For instance, a chordless path P_4 on four consecutive vertices a, b, c, d can be defined as follows:

$$\eta_{3,2}(3(d) \oplus \rho_{3\to 2}(\rho_{2\to 1}(\eta_{3,2}(3(c) \oplus \eta_{2,1}(2(b) \oplus 1(a)))))).$$

The *width* of a clique-decomposition is the number of different labels used in the expression, and the *clique-width* of a graph G, $\mathbf{cw}(G)$, is the minimum width among all the possible clique-decompositions of G. For instance, from the above example, we conclude $\mathbf{cw}(P_4) \leq 3$.

It is known that $\mathbf{cw}(G) \leq 2^{\mathbf{tw}(G)}$ holds for any graph G [30]. However, bounded clique-width does not imply bounded tree-width. For example, the complete graph of n vertices has tree-width $n - 1$ and clique-width 2.

In this section, we prove the following theorem.

Theorem 6.2. *For any $\alpha > 0$, the following are equivalent:*

1. Independent Set *can be solved in $O^*(2^{\alpha \mathbf{tw}})$ time and*
2. Independent Set *can be solved in $O^*(2^{\alpha \mathbf{cw}})$ time,*

where **tw** *is the width of the given tree-decomposition and* **cw** *is the width of the given clique-decomposition.*

The proof consists of two reductions. In Subsection 6.4.1, we give a clique-decomposition-based reduction from Independent Set parameterized by clique-width to SAT parameterized by tree-width, and in Subsection 6.4.2, we give a tree-decomposition-based reduction from 3-SAT parameterized by tree-width to Independent Set parameterized by clique-width. From Theorem 6.1, SAT, 3-SAT, and Independent Set parameterized by tree-width are equivalent. Thus, these two reductions prove the theorem.

6.4.1 From Independent Set Parameterized by Clique-width to SAT Parameterized by Tree-width

Let $(G = (V, E), k)$ be an instance of INDEPENDENT SET of n vertices. Let $\mathbf{cw}$ be the width of the given clique-decomposition of G. We want to construct an instance $(X, \mathcal{C})$ of SAT with tree-width $\mathbf{cw} + O(\log n)$ such that $\mathcal{C}$ is satisfiable if and only if there is an independent set of size k in G. Let $M = \lceil \log(n + 1) \rceil$.

Let O be the set of operations in the given clique-decomposition. Note that the clique-decomposition can be represented as a tree, which we call the *expression tree* of G, and we will often identify an operation and the corresponding node in the expression tree. For each operation $o \in O$, we associate a subgraph G_o constructed by performing operations in the subtree rooted at the operation o. For each operation $o \in O$, we introduce variables $\{o_i \mid i \in [\mathbf{cw}]\} \cup \{s_{o,i} \mid i \in [M]\}$. For each $i \in [\mathbf{cw}]$, the variable o_i represents whether vertices with label i are chosen to be an independent set in G_o, and $\{s_{o,i} \mid i \in [M]\}$ represents the size of the independent set in G_o.

We add constraints as follows depending on the type of the operation o.

- $o = i(v)$: create a constraint $(s_{o,*})_2 = (o_i)_2$.
- $o = c \oplus c'$: create two constraints $c_i \to o_i$ and $c'_i \to o_i$ for each $i \in [\mathbf{cw}]$, and a constraint $(s_{o,*})_2 = (s_{c,*})_2 + (s_{c',*})_2$.
- $o = \eta_{i,j}(c)$: create a constraint $o_i = c_i$ for each $i \in [\mathbf{cw}]$, a constraint $\overline{o_i} \vee \overline{o_j}$, and a constraint $(s_{o,*})_2 = (s_{c,*})_2$.
- $o = \rho_{i \to j}(c)$: create a constraint $o_k = c_k$ for each $k \in [\mathbf{cw}] \setminus \{i, j\}$ and three constraints $(s_{o,*})_2 = (s_{c,*})_2$, $o_j = c_i \vee c_j$, and $(\overline{o_i})$.

Finally, for the root operation $o \in O$, we add a constraint $(s_{o,*})_2 \geq k$. Note that for an operation $o = c \oplus c'$, the created constraints $c_i \to o_i$ and $c'_i \to o_i$ actually perform as a constraint $o_i = (c_i \vee c'_i)$. This is because if there exists a satisfiable assignment for which both of c_i and c'_i are set to false but o_i is set to true, we can obtain another satisfiable assignment by setting o_i and the variables connected by equality constraints to false.

Now we have obtained an instance $(X, \mathcal{C})$ of polynomial size. As the above construction directly simulates the dynamic programming for solving INDEPENDENT SET [31], $\mathcal{C}$ is satisfiable if and only if there is an independent set of size at least k. Now we show that the tree-width of the instance $(X, \mathcal{C})$ has essentially the same clique-width of the graph G.

Lemma 6.7. *$\mathcal{C}$ has tree-width at most* $\mathbf{cw} + O(\log n)$.

Proof. We will prove the bound by reducing the primal graph of $\mathcal{C}$ into an empty graph by a series of eliminations of degree at most $\mathbf{cw}+O(\log n)$. For an operation o, let Y_o denote the vertex set $\{o_i \mid i \in [\mathbf{cw}]\}$, and V_i denote the vertex set $Y_o \cup \{s_{o,i} \mid i \in [M]\}$.

Starting from the primal graph of $\mathcal{C}$ and the given clique-decomposition of G, we eliminate the vertices as follows. First, we choose an arbitrary operation o corresponding to a leaf in the expression tree. Then, we eliminate all the vertices of V_i in a certain order, which will be described later. Finally, we remove o from the expression tree and repeat the process until the expression tree becomes empty.

Let o be an operation corresponding to a leaf of the current expression tree, and p be its parent. If o is the only child of p, it holds that $N(V_o) \subseteq V_p$. Thus, the eliminations of V_o can create edges only inside V_p. If p has another child q, it

holds that $N(V_o) \subseteq V_p \cup \{s_{q,i} \mid i \in [M]\}$. Thus, the eliminations of V_i can create edges only inside $V_p \cup \{s_{q,i} \mid i \in [M]\}$. Therefore, after processing each node, we can ensure that the edges created by previous eliminations are only inside $V_o \cup \{s_{c,i} \mid c$ is a child of o and $i \in [M]\}$ for each operation o.

Now, we describe the details of the eliminations. Let o be the current operation to process. If o is the root, the number of remaining vertices is $\mathbf{cw} + O(\log n)$. Thus, the elimination of these vertices has degree $\mathbf{cw} + O(\log n)$. Otherwise, let p be the parent of o. First, we eliminate vertices Y_o. Because each vertex of Y_i is adjacent to at most one vertex of Y_p, Lemma 6.3 gives the elimination of degree $|N[Y_o] \setminus Y_p| \leq |V_o| \leq \mathbf{cw} + O(\log n)$. Then, we eliminate the remaining vertices $V_o \setminus Y_o$. If o is the only child of p, let $V_q = Y_q = \emptyset$, and otherwise, let q be the another child of p. By applying Lemma 6.2, we obtain the elimination of degree $|N[V_o \setminus Y_o]| - 1 \leq |V_o \setminus Y_o| + |V_p| + |V_q \setminus Y_q| \leq \mathbf{cw} + O(\log n)$. □

6.4.2 From 3-SAT Parameterized by Tree-width to Independent Set Parameterized by Clique-width

Let $(X, \mathcal{C} = \{C_1, \ldots, C_m\})$ be an instance of 3-SAT with tree-width $\mathbf{tw}$. We want to construct an instance (G, k) of Independent Set with clique-width $\mathbf{tw} + O(\log \mathbf{tw})$ such that G has an independent set of size at least k if and only if $\mathcal{C}$ is satisfiable. For this purpose, we use the same construction of (G, k) as in Section 6.3.2. Hence, it suffices to show that the graph G has clique-width $\mathbf{tw} + O(\log \mathbf{tw})$.

Lemma 6.8. *The graph G has clique-width at most $\mathbf{tw} + O(\log \mathbf{tw})$.*

Proof. Let $T = (I, F)$ be a nice tree-decomposition of $(X, \mathcal{C})$. We inductively construct G by processing each node of T in a bottom-up manner.

For a node $i \in I$, let $I_i^{\downarrow} \subseteq I$ be the set consisting of i itself and descendants of i. Then, we define $X_i^{\downarrow}$ and $\mathcal{C}_i^{\downarrow}$ as the sets of variables and constraints, respectively, contained in a bag of $I_i^{\downarrow}$. Let $G_i^{\downarrow}$ be the subgraph of G induced by variable gadgets corresponding to vertices in $X_i^{\downarrow}$, clause gadgets corresponding to clauses in $\mathcal{C}_i^{\downarrow}$, child counting gadgets for nodes in $I_i^{\downarrow}$ and parent counting gadgets for nodes in $I_i^{\downarrow} \setminus \{i\}$. At node i, we will construct the graph $G_i^{\downarrow}$.

We introduce a special label #; if a vertex is once labeled #, then we will never relabel or connect new edges to that vertex. For each $i \in I$, we ensure that vertices $\overline{x}_i$ for $x \in X$ and vertices in the last layer of the child counting gadget for i have distinct labels, and all the other vertices in $G_i^{\downarrow}$ are labeled # after processing the node i.

Suppose that we have constructed $G_c^{\downarrow}$ for a child node c of i (if i is a Join node, we also have another graph $G_{c'}^{\downarrow}$ for the other child c'), and we want to construct a graph $G_i^{\downarrow}$. We have five cases depending on the type of the node i.

(i) If i is a leaf node, we have nothing to do.

(ii) Suppose i is an Introduce(x) node. Let $X_c = \{x^1, \ldots, x^d\}$ for some $d \leq \mathbf{tw}$. Note that $X_i = \{x^1, \ldots, x^d, x\}$ holds.

For each $j \in [d]$, we do the following: We first construct a variable gadget for x^j using new labels. We then connect $\overline{x}_c^j$ to x_i^j, and the label of $\overline{x}_c^j$ is set to #. Next, we create the j-th layer of the child counting gadget S_i for i, and connect $\overline{x}_i^j$ to it. This can be done using auxiliary $O(\log \mathbf{tw})$ labels. Then, the labels of $(j-1)$-th layer (if exists) of S_i and the label of x_i^j are set to #. Finally, we create the j-th layer of the parent counting gadget T_c for c, and connect $\overline{x}_i^j$ to it. This

can be done using auxiliary $O(\log \mathbf{tw})$ labels. Then, the labels of $(j-1)$-th layer (if exists) of T_c are set to #.

After processing $x^1, \ldots, x^d$, we create a variable gadget for x_i and connect x_i with the $(d+1)$-th layer of S_i for i. Then, the labels of d-th layer (if exists) of S_i are set to #. Finally, we connect the last layers of T_c and the child counting gadget S_c for c to make the constraint $|\{x_c^j \mid j \in [d]\} \cup \{\overline{x}_i^j \mid j \in [d]\} \cap S| \leq d$ for any independent set S. In total, we only need $\mathbf{tw} + O(\log \mathbf{tw})$ labels.

(iii) Suppose i is an Introduce($x \vee y \vee z$) node. The construction is very similar to the case (ii). The only difference is that we have to make a clause gadget corresponding to the clause $(x \vee y \vee z)$, where x, y, and z are literals. Recall that, in the case (ii), the label of x_i^j is set to # after the j-th iteration. Instead, if x_i^j is the literal used in the clause, then we keep it using a new label. After the d-th step, we construct a clause gadget using these kept literals. We only need $O(1)$ auxiliary labels for this construction since the clause $(x \vee y \vee z)$ has only three literals.

(iv) If i is a Forget(v) node or (v) a Join node, then the construction is almost the same as (ii), and we omit the detail.

To summarize, we can construct G using $\mathbf{tw} + O(\log \mathbf{tw})$ labels. □

6.5 Reduction from #Perfect Matching to #SAT

A *matching* of a graph $G = (V, E)$ is a set of pairwise non-adjacent edges, that is, no two edges share a common vertex. A *perfect matching* is a matching that matches all vertices of the graph, that is, a matching of size $|V|/2$. #Perfect Matching is the problem in which, given a graph G, the objective is to count the number of perfect matchings. In this section, we prove the following theorem.

Theorem 6.3. *For any $\alpha > 0$, if #SAT can be solved in $O^*(2^{\alpha \mathbf{tw}})$ time, then #Perfect Matching can be solved in $O^*(2^{\alpha \mathbf{tw}})$ time, where $\mathbf{tw}$ is the width of the given tree-decomposition.*

While the standard dynamic programming can solve #SAT in $O^*(2^{\mathbf{tw}})$ time, it takes $O^*(3^{\mathbf{tw}})$ for solving #Perfect Matching. By exploiting fast subset convolution [18], van Rooij et al. [100] improved the running time to $O^*(2^{\mathbf{tw}})$. Hence, #Perfect Matching seems more difficult than #SAT. However, the theorem shows that #Perfect Matching is as easy as #SAT. Interestingly, the inverse reduction from #SAT to #Perfect Matching, which looks easier at first glance because it is a reduction from the problem that can be solved by the simple algorithm to the problem that needs the sophisticated technique, seems rather difficult. We left it as an open problem.

Now, we prove the theorem by giving a tree-decomposition-based reduction. Let $G = (V, E)$ be an instance of #Perfect Matching. We want to construct an instance $(X, \mathcal{C})$ of SAT such that the number of perfect matchings of G can be computed from the number of satisfiable assignments of $(X, \mathcal{C})$.

Before describing our reduction, we first introduce the *fast subset convolution* [18] which is involved in the $O^*(2^{\mathbf{tw}})$-time algorithm for #Perfect Matching [100]. Let U be a set of n elements and R be an arbitrary ring. For a function $f : 2^U \to R$, the *zeta transform* $f\zeta$ is a function defined by $f\zeta(S) = \sum_{T \subseteq S} f(T)$, and the *Möbius transform* $f\mu$ is a function defined by $f\mu(S) = \sum_{T \subseteq S} (-1)^{|S \setminus T|} f(T)$. For two functions $f, g : 2^U \to R$, the *subset convolution* $f * g$ is a function defined by $(f * g)(S) = \sum_{T \subseteq S} f(T) g(S \setminus T)$. We write element-wise product of two functions f and g by $f \cdot g$. Let t be an indeterminate.

For a polynomial $p(t)$, we write $p(t)[t^i]$ to address the coefficient of the monomial t^i in $p(t)$. The *ranked function* f_t is a function defined by $f_t(S) = f(S)t^{|S|}$. Björklund *et al.* [18] showed that the subset convolution $f * g$ can be computed in $O^*(2^n)$ time by proving the following Lemma:

Lemma 6.9 ([18])**.**

$$(f * g)(S) = ((f_t\zeta \cdot g_t\zeta)\mu)(S)[t^{|S|}]$$

Now, we give a tree-decomposition-based reduction from #PERFECT MATCHING to #SAT. Let $M = \lceil \log(\mathbf{tw} + 2) \rceil$, and $T = (I, F)$ be a given nice tree-decomposition of width $\mathbf{tw}$. For each node $i \in I$, we create variables $U_i = \{x_i \mid x \in X_i\}$ and a *counting gadget* as follows. Let $U_i = \{u_1, \ldots, u_d\}$. First, we create $d + 1$ layers of variables $\{s_{i,j,k} \mid j \in [d + 1], k \in [M]\}$. For the first layer, we create a clause $\overline{s_{i,1,k}}$ for each $k \in [M]$. Then, for each $j \in [d]$, we create a constraint $(s_{i,j+1,*})_2 = (s_{i,j,*})_2 + (u_j)_2$. We denote the set of variables in the j-th layer by $S_{i,j} = \{s_{i,j,k} \mid k \in [M]\}$ and abbreviate the last layer $S_{i,d+1}$ as $S_i = \{s_{i,1}, \ldots, s_{i,M}\}$. This gadget ensures that the value $(s_{i,*})_2$ represents the number of true variables in U_i.

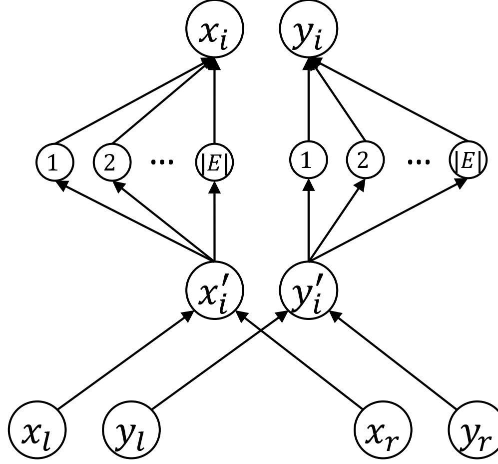

Figure 6.4: The subset convolution gadget

Then, for each node i, according to its type, we do as follows:

1. Leaf: do nothing
2. Introduce(v): create a constraint $x_i = x_c$ for each $x \in X_c$ and a clause $\overline{v_i}$.
3. Introduce(uv): create a constraint $x_i = x_c$ for each $x \in X_i \setminus \{u, v\}$ and a constraint $(u_i \wedge v_i \wedge \overline{u_c} \wedge \overline{v_c}) \vee ((u_i = u_c) \wedge (v_i = v_c))$.
4. Forget(v): create a constraint $x_i = x_c$ for each $x \in X_i$.
5. Join: create a *subset convolution gadget* depicted in Figure 6.4 (see below for the details).

For a Join node i, we construct a gadget that simulates the fast subset convolution algorithm. The gadget consists of two gadgets: a *zeta transform gadget* and a *Möbius transform gadget*. Let $U = \{u_1, \ldots, u_d\}$ and $V = \{v_1, \ldots, v_d\}$ be two variable sets. The zeta transform gadget from U to V consists of constraints

$\{u_i \to v_i \mid i \in [d]\}$. The Möbius transform gadget from U to V consists of variables $\{\mu_{i,j} \mid i \in [d], j \in [|E|]\}$ and constraints $\{u_i \to \mu_{i,j} \to v_i \mid i \in [d], j \in [|E|]\}$.

Let i be a Join node with two children l and r. We create a constraint $(s_{i,*})_2 = (s_{l,*})_2 + (s_{r,*})_2$. This ensures that the number of true variables in U_i equals that of true variables in $U_l \cup U_r$. Then, we create variables $U'_i = \{x'_i \mid x \in X_i\}$. Finally, we construct zeta transform gadgets from U_l to U'_i and U_r to U'_i, and construct a Möbius transform gadget from U'_i to U_i.

Now, we have obtained the CNF $(X, \mathcal{C})$ of polynomial size. We will show that we can compute the number of perfect matchings of the input graph G from the number of satisfiable assignments of the CNF.

Lemma 6.10. *The number of satisfiable assignments of the CNF $(X, \mathcal{C})$ modulo $(2^{|E|} + 1)$ is exactly the number of perfect matchings of the input graph G.*

Proof. For a node i, let $G_i = (V_i, E_i)$ represent a graph such that the vertex set V_i is a union of X_j over all descendants j of i and the edge set E_i consists of edges in Introduce(e) nodes in the subtree rooted at i. Let $\mathcal{C}_i$ be the CNF consisting of all clauses created in the reduction for the subtree rooted at i.

For a node i and a subset $S \subseteq X_i$, let $G_i(S)$ be the graph $G_i[(V_i \setminus X_i) \cup S]$ and $\mathcal{C}_i(S)$ be the CNF $\mathcal{C}_i \cup \{x_i \mid x \in S\} \cup \{\overline{x_i} \mid x \in (X_i \setminus S)\}$. We denote the number of perfect matchings of $G_i(S)$ by $m_i(S)$ and the number of satisfiable assignments of $\mathcal{C}_i(S)$ modulo $(2^{|E|}+1)$ by $a_i(S)$. By induction, we will show that for any node i and a subset $S \subseteq X_i$, it holds that $m_i(S) = a_i(S)$. Since for the root node r, it holds that $X_r = \emptyset$ and $G_r = G$, this proves the lemma.

For a leaf i, the equality trivially holds because G_i is an empty graph and $\mathcal{C}_i$ has a unique satisfiable assignment that assigns false to all the variables ($\{s_{i,1,k} \mid k \in [M]\}$).

Let i be an Introduce(v) node. Then, it holds that $X_i = X_c \cup \{v\}$, $V_i = V_c \cup \{v\}$, and v is an isolated vertex in G_i. When S contains v, $m_i(S)$ becomes zero because v has no neighbors in $G_i(S)$. Since we have created a clause $\overline{v_i}$, $a_i(S)$ also becomes zero. When S does not contain v, $G_i(S)$ equals $G_c(S)$. Therefore, $m_i(S)$ equals $m_c(S)$. Since we have created a constraint of $x_i = x_c$ for each vertex x of X_c and a clause $\overline{v_i}$, $a_i(S)$ also equals $a_c(S)$. From the inductive assumption, we have $m_i(S) = m_c(S) = a_c(S) = a_i(S)$.

Let i be an Introduce(uv) node. Then, it holds that $X_i = X_c$, $V_i = V_c$, and $E_i = E_c \cup \{uv\}$. When S does not contain at least one of u and v, $G_i(S)$ equals $G_c(S)$. Thus, $m_i(S)$ equals $m_c(S)$. We have created a constraint of $(u_i \wedge v_i \wedge \overline{u_c} \wedge \overline{v_c}) \vee ((u_i = u_c) \wedge (v_i = v_c))$. Since the constraint $(u_i \wedge v_i \wedge \overline{u_c} \wedge \overline{v_c})$ is not satisfiable, we must satisfy the constraint $(u_i = u_c) \wedge (v_i = v_c)$. Therefore, $a_i(S)$ also equals $a_c(S)$. From the inductive assumption, we have $m_i(S) = m_c(S) = a_c(S) = a_i(S)$. When S contains both of u and v, a perfect matching of $G_i(S)$ may contain the edge uv. If a perfect matching $\mathcal{M}$ contains the edge uv, $\mathcal{M} \setminus \{uv\}$ is a perfect matching of $G_c(S \setminus \{u, v\})$. Conversely, any perfect matching of $G_c(S \setminus \{u, v\})$ can be extended to a perfect matching of $G_i(S)$ by adding the edge uv. If a perfect matching $\mathcal{M}$ does not contain the edge uv, it must be a perfect matching of $G_c(S)$. Therefore, it holds that $m_i(S) = m_c(S) + m_c(S \setminus \{u, v\})$. Any satisfiable assignment of $\mathcal{C}_i(S)$ must satisfy exactly one of $(u_i \wedge v_i \wedge \overline{u_c} \wedge \overline{v_c})$ and $(u_i = u_c) \wedge (v_i = v_c)$. If the former constraint is satisfied, it also satisfies $\mathcal{C}_c(S \setminus \{u, v\})$. If the latter constraint is satisfied, it also satisfies $\mathcal{C}_c(S)$. Therefore, it holds that $a_i(S) = a_c(S) + a_c(S \setminus \{u, v\})$. From the inductive assumption, we have $m_i(S) = m_c(S) + m_c(S \setminus \{u, v\}) = a_c(S) + a_c(S \setminus \{u, v\}) = a_i(S)$.

Let i be a Forget(v) node. Then, it holds that $X_i = X_c \setminus \{v\}$ and $G_i = G_c$. Therefore, we have $m_i(S) = m_c(S)$. Since we have created a constraint of $x_i = x_c$

for each vertex x of X_i, $a_i(S)$ also equals $a_c(S)$. From the inductive assumption, we have $m_i(S) = m_c(S) = a_c(S) = a_i(S)$.

Let i be a Join node. Then, it holds that $X_i = X_l = X_r = V_l \cap V_r$, $E_i = E_l \cup E_r$, and $E_l \cap E_r = \emptyset$. Let $\mathcal{M}$ be a perfect matching of $G_i(S)$ and T be the set of vertices in S that are incident to $\mathcal{M} \cap E_l$. Since $\mathcal{M} \cap E_l$ is a perfect matching of $G_l(T)$ and $\mathcal{M} \cap E_r$ is a perfect matching of $G_r(S \setminus T)$, it holds that $m_i(S) = \sum_{T \subseteq S} m_l(T) m_r(S \setminus T) = (m_l * m_r)(S)$. Let $a'_i(T, k)$ be the number of satisfiable assignments modulo $(2^{|E|}+1)$ of the CNF consisting of $\mathcal{C}_l$, $\mathcal{C}_r$, the zeta transform gadgets from U_l to U'_i and U_r to U'_i, the clauses $\{x'_i \mid x \in T\} \cup \{\overline{x'_i} \mid x \in (X_i \setminus T)\}$, and the constraint that the number of true variables in $U_l \cup U_r$ is k. Since the CNF contains the constraints of $(x_l \to x'_i)$ and $(x_r \to x'_i)$, we have $a'_i(T, k) = \sum_{T_l, T_r \subseteq T, |T_l| + |T_r| = k} a_l(T_l) a_r(T_r)$. By using an indeterminate t, we can write $a'_i(T, k) = (\sum_{T_l \subseteq T} a_l(T_l) t^{|T_l|})(\sum_{T_r \subseteq T} a_r(T_r) t^{|T_r|})[t^k] = (((a_l)_t \zeta) \cdot ((a_r)_t \zeta))(T)[t^k]$. The CNF $\mathcal{C}_i$ additionally has the Möbius transform gadget from U'_i to U_i and the constraint that the number of true variables in U_i equals that of true variables in $U_l \cup U_r$. For each vertex $x \in X_i$, the Möbius transform gadget contains the constraints $\{x'_i \to \mu_j \to x_i \mid j \in [|E|]\}$. If x'_i is false and x_i is true, each μ_j can take arbitrary value. Thus, there are $2^{|E|} = -1$ (modulo $2^{|E|} + 1$) possibilities. Otherwise, all of x'_i, x_i, and μ_j must take the same value. Therefore, we have $a_i(S) = \sum_{T \subseteq S} (-1)^{|S \setminus T|} a'_i(T, |S|) = ((((a_l)_t \zeta) \cdot ((a_r)_t \zeta))\mu)(S)[t^{|S|}]$. From Lemma 6.9, we have $a_i(S) = (a_l * a_r)(S)$. Thus, from the inductive assumption, we have $m_i(S) = (m_l * m_r)(S) = (a_l * a_r)(S) = a_i(S)$. □

Finally, we show that the reduction preserves the tree-width.

Lemma 6.11. *$\mathcal{C}$ has tree-width at most* $\mathbf{tw} + O(\log \mathbf{tw})$.

Proof. Let G' be the primal graph of $\mathcal{C}$. We will prove the bound by reducing G' into an empty graph by a series of eliminations of degree at most $\mathbf{tw} + O(\log \mathbf{tw})$. Starting from G' and the given tree-decomposition T of the input graph G, we eliminate the vertices as follows.

First, for each constraint $(x'_i \to \mu \to x_i)$ of each Möbius transform gadget, we eliminate the intermediate vertex μ one by one. The elimination can only create an edge between x'_i and x_i. Since these vertices are independent and have degree two, the eliminations have a constant degree. Now, the graph consists of only three types of vertices: U_i and $S_{i,j}$ for each node i and U'_i for each Join node i. For a node i, we denote by V_i the set of vertices consisting of U_i, $\{S_{i,j} \mid j \in [|X_i|+1]\}$, and U'_i if i is a Join node.

Then, we choose an arbitrary leaf i of the tree T and eliminate all the vertices of V_i in a certain order, which will be described later. Finally, we remove i from T and repeat the process until T becomes empty. Let p be the parent of a leaf i. If p is a Join node, $N(V_i) \subseteq U'_p \cup S_p \cup S_j$, where j is another child of p, and otherwise $N(V_i) \subseteq U_p$.

Now, we describe the details of the eliminations. Let i be the current node to process. If i is a Join node, we first eliminate the vertices U'_i. Since for each vertex $x'_i \in U'_i$, it holds that $N(x'_i) \cap U_i = \{x_i\}$, from Lemma 6.3, the elimination has degree $|N[U'_i] \setminus U_i| \le |U'_i \cup S_i| \le \mathbf{tw} + O(\log \mathbf{tw})$. Then, we eliminate the vertices $S = \bigcup_{j \in [|X_i|+1]} S_{i,j}$. Since there are no edges between $S_{i,a}$ and $S_{i,b}$ for $|a - b| > 1$, from Lemma 6.4, the elimination has degree $2M + |N(S)| - 1 \le 2M + (|X_i| + 2M) - 1 \le \mathbf{tw} + O(\log \mathbf{tw})$. Finally, we eliminate the vertices U_i. If i is the root, there are no other vertices. Thus, the elimination has degree $|U_i| \le \mathbf{tw}$. Otherwise, let p be the parent of i. Let U be the set U'_p if p is a

Join node, $U_p \setminus \{u_p, v_p\}$ if p is an Introduce(uv) node, and U_p if otherwise. Since it holds that $|N(x_i) \cap U| \leq 1$ for each vertex $x_i \in U_i$, from Lemma 6.4, the elimination has degree $|N[U_i] \setminus U| \leq \mathbf{tw} + O(\log \mathbf{tw})$. □

6.6 Equivalence among Problems of Bounded Branch-width

We first define the notion of branch-width formally. A *branch-decomposition* of a graph $G = (V, E)$ is a ternary tree (i.e., every node has degree one or three) $T = (I, F)$ with one-to-one correspondence between edges of G and leaf nodes of T. By removing an arc $a \in F$ of T, we obtain two subtrees $T_1(a)$ and $T_2(a)$. Let $E_1(a)$ and $E_2(a)$ be edges of G corresponding to leaf nodes of $T_1(a)$ and $T_2(a)$, respectively. The *middle set* of an arc $a \in F$, denoted by $\mathbf{mid}(a)$, is the set of vertices of G incident both to an edge in $E_1(a)$ and an edge in $E_2(a)$. The *width* of a branch-decomposition is the maximum size of a middle set over all arcs of T. The *branch-width* of a graph G, denoted by $\mathbf{bw}(G)$, is the minimum width among all the possible branch-decompositions of G.

Tree-width and branch-width are strongly related graph parameters: for any graph G with $|E| \geq 3$, we have $\mathbf{bw}(G) \leq \mathbf{tw}(G)+1 \leq \frac{3}{2}\mathbf{bw}(G)$ [93]. Moreover, we can convert a tree-decomposition of width w into a branch-decomposition of width at most $w+1$, and branch-decomposition of width w into a tree-decomposition of width at most $\frac{3}{2}w$ in polynomial time. For any graph G with n vertices, $\mathbf{tw}(G)$ is at most $n-1$, and $\mathbf{bw}(G)$ is at most $\lceil \frac{2}{3}n \rceil$.

In this section, we prove the following theorem.

Theorem 6.4. *For any $\alpha > 0$, the following are equivalent:*

1. Max 2-SAT *can be solved in $O^*(2^{\alpha \mathbf{bw}})$ time and*
2. Independent Set *can be solved in $O^*(2^{\alpha \mathbf{bw}})$ time,*

where $\mathbf{bw}$ *is the width of the given branch-decomposition.*

The proof consists of bidirectional reductions between Max 2-SAT and Independent Set. The branch-width preserving reduction from Independent Set to Max 2-SAT is completely the same as the one for the tree-width case described in Subsection 6.3.4. Thus, the remaining task is to give a branch-width preserving reduction from Max 2-SAT to Independent Set.

Let $(X, \mathcal{C} = \{C_1, \ldots, C_m\}, k)$ be an instance of Max 2-SAT. If there exists a variable x such that any clause containing x always contains the same variable y, we can safely replace x with y or $\overline{y}$. Thus, without loss of generality, we can assume that each vertex in the primal graph of $\mathcal{C}$ has degree at least two. We want to construct an instance (G, k') of Independent Set such that G has an independent set of size at least k' if and only if at least k clauses of $\mathcal{C}$ can be satisfied. Especially, in our reductions, we choose k' so that any independent set has size at most k'. In the following reduction, we will use the variable gadget, the clause gadget, and the counting gadget defined in Subsection 6.3.2. Let $M = \lceil \log(m+1) \rceil$. We will also use (arithmetic) constraints on $O(M)$ variables, which can be simulated by poly(m) clauses.

We will construct a graph with branch-width at most $\mathbf{bw} + O(\log m)$. As we discussed before, the additive $O(\log m)$ factor is allowed. Let $T' = (I', F')$ be a given branch-decomposition of width $\mathbf{bw}$. For convenience, we modify it to a rooted binary tree $T = (I, F)$ by choosing an arbitrary arc $ij \in F'$ and inserting a root vertex r between i and j. For a node i with a parent p, we denote the middle set $\mathbf{mid}(ip)$ by X_i. For the root r, we define $X_r = \emptyset$. For a variable $x \in X$, let h_x

be the node $i \in I$ with two children l and r such that X_i does not contain x but both X_l and X_r contain x. Note that the set of arcs $T_x = \{a \in F \mid x \in \mathbf{mid}(a)\}$ is connected in the tree T and is non-empty as every variable has degree at least two in the primal graph. Thus, the node h_x is uniquely determined as the maximally high node incident to T_x. For a node $i \in I$, the *left variables* are defined as $L_i = \{x \in X_i \mid i \text{ is contained in the subtree rooted at the left child of } h_x\}$. In a similar way, we define the *right variables* as $R_i = \{x \in X_i \mid i \text{ is contained in the subtree rooted at the right child of } h_x\}$. We note that $X_i = L_i \cup R_i$ holds.

Now, we explain our branch-decomposition-based reduction. For each node $i \in I$, we create a variable gadget for each of $\{x_i \mid x \in X_i\}$ and a variable gadget for each of $\{w_{i,j} \mid j \in [M]\}$. We force the value $(w_{i,*})_2$ to represent the number of satisfied clauses in the subtree rooted at i as follows. If i is a leaf node, then we create clause gadgets on X_i and $\{w_{i,j} \mid j \in [M]\}$ so that $(w_{i,*})_2$ represents the number of satisfied clauses on the variables X_i. Since the size of X_i is two, such a constraint can be simulated by at most $2^{2+M} = O(\mathrm{poly}(m))$ clauses. If i is not a leaf node, then let l and r be its two children. Then, we create clause gadgets simulating a constraint $(w_{i,*})_2 = (w_{l,*})_2 + (w_{r,*})_2$.

Then, we will ensure that for any independent set S of size k' and for any variable $x \in X$, all the vertices $\{x_i \mid i \in I \text{ s.t. } x \in X_i\}$ are in S or none of them are in S.

First, for each non-root node i and its parent p, we insert edges and gadgets as follows; For each variable $x \in X_i$, if x is in L_p, then we connect the vertices $\overline{x_i}$ and x_p by an edge, and if x is in R_p, then we connect the vertices x_i and $\overline{x_p}$ by an edge. Then, we construct a counting gadget of $\{x_i \mid x \in X_i \cap L_p\} \cup \{\overline{x_i} \mid x \in X_i \cap R_p\}$, called an *ip-counting gadget* and a counting gadget of $\{\overline{x_p} \mid x \in X_i \cap L_p\} \cup \{x_p \mid x \in X_i \cap R_p\}$, called a *pi-counting gadget*. Finally, we create clause gadgets simulating the constraint that the sum of the numbers represented by the last layers of these two counting gadgets must be at most $|X_i \cap X_p|$. As we discussed in Subsection 6.3.2, this ensures that for any independent set of size k', $x_i \in S \Leftrightarrow x_p \in S$ holds for any $x \in X_i \cap X_p$.

Then, for each non-leaf node i and its two children l and r, we insert edges and gadgets as follows; For each variable x such that $h_x = i$, we connect the vertices $\overline{x_l}$ and x_r by an edge. Then, we construct a counting gadget of $\{x_l \mid h_x = i\}$, called an *lr-counting gadget* and a counting gadget of $\{\overline{x_r} \mid h_x = i\}$, called a *$rl$-counting gadget*. Finally, we create clause gadgets simulating a constraint that the sum of the numbers represented by the last layers of these two counting gadgets must be at most $|\{x \in X \mid h_x = i\}|$. This ensures that for any independent set of size k', $x_l \in S \Leftrightarrow x_r \in S$ holds for any $x \in X$ such that $h_x = i$.

Finally, for the root r, we create clause gadgets simulating a constraint $(w_{r,1} \dots w_{r,M})_2 \geq k$. Now, we have obtained a graph G of polynomial size and we set k' as the number of variable gadgets plus the number of clause gadgets. From our construction, for any independent set of size k' and a variable $x \in X$, all the vertices $\{x_i \mid i \in I \text{ s.t. } x \in X_i\}$ are in S or none of them are in S, and all the clauses in the clause gadgets must be satisfied. Thus, if G has an independent set of size k', then at least k clauses of $\mathcal{C}$ can be satisfied. Conversely, if at least k clauses of $\mathcal{C}$ can be satisfied, then by taking an appropriate vertex from each gadget, we can obtain an independent set of size k'.

Finally, we show that the reduction preserves the branch-width. A *caterpillar tree* of length N is a binary tree that consists of nodes $\{p_i \mid i \in [N+1]\} \cup \{q_i \mid i \in [N]\}$ and arcs $\{p_i p_{i+1} \mid i \in [N]\} \cup \{p_i q_i \mid i \in [N]\}$. Intuitively, our branch-decomposition is a tree obtained by replacing each arc of the given branch-

decomposition with a caterpillar tree. Let f be a function from arcs of a rooted tree T to subsets of vertices of a graph G. By removing an arc a of T, we obtain a subtree T_a. We denote the union of the vertex sets $f(b)$ for each arc b of T_a by $f'(a)$. Let $<$ be the post-ordering of the arcs of T, that is, $a < b$ if and only if a is contained in T_b or there exists a common ancestor node of a and b such that a is in the left subtree and b is in the right subtree. For a vertex v and an arc a, let $N_a(v) = N(v) \cap (\bigcup_{b \geq a} f(b))$. Similarly, for a vertex set S, we denote $N_a(S) = N(S) \cap (\bigcup_{b \geq a} f(b))$. We say that a vertex v is *consumed at an arc* a if for any vertex u of $N(v) \setminus N_a(v)$, there exists an arc b contained in T_a such that $u \in f(b)$. For a vertex set S and an arc a, we denote the set of unconsumed vertices in S at a by $U_a(S)$. We use the following auxiliary lemma.

Lemma 6.12. *Let G be a graph, T be a ternary tree, f be a function from arcs of T to subsets of vertices of G such that there exists exactly one arc a with $v \in f(a)$ for each vertex v of G, and w be an integer. If for any arc a of T, there exists an ordering $(v_1, \ldots, v_N)$ of $f(a)$ such that $|N_a(f'(a) \cup \{v_1, \ldots, v_i\})| + |U_a(f'(a) \cup \{v_1, \ldots, v_i\})| \leq w$ holds for any $i \in \{0\} \cup [N]$, then $\mathbf{bw}(G) \leq w + 1$.*

Proof. We construct a branch-decomposition of G as follows. For an arc a between a node c and its parent p of T, let $(v_1, \ldots, v_N)$ be the ordering of $f(a)$ such that $|N_a(f'(a) \cup \{v_1, \ldots, v_i\})| + |U_a(f'(a) \cup \{v_1, \ldots, v_i\})| \leq w$ holds. Then, by taking edges $\{v_i u \mid u \in N_a(v_i) \setminus \{v_1, \ldots, v_{i-1}\}\}$ from $i = 1$ to $i = N$, we obtain an ordered list $F_a = (e_1, \ldots, e_{N'})$ of edges. We replace an arc a with a caterpillar tree of length N' by connecting c to the head p_1 of the caterpillar and identifying p to the tail $p_{N'+1}$. For each $i \in [N']$, we associate the node q_i of the caterpillar with the edge e_i. Finally, by removing leaf nodes that have no correspondence to edges and eliminating internal nodes of degree two, we obtain a branch-decomposition of G.

Now, we show that the obtained branch-decomposition has width at most $w + 1$. Let b be an arc between p_j and p_{j+1} of a caterpillar inserted to replace an arc a and let $v_i \in f(a)$ be the vertex incident to the edge e_j corresponding to the node q_j. Then, from the definition of N_a and U_a, $\mathbf{mid}(b)$ is contained in $N_a(f'(a) \cup \{v_1, \ldots, v_i\}) \cup U_a(f'(a) \cup \{v_1, \ldots, v_i\}) \cup \{v_i\}$, whose size is at most $w + 1$. □

Lemma 6.13. *G has branch-width at most $\mathbf{bw} + O(\log m)$.*

Proof. In order to make the proof simple, we first eliminate the vertices of each clause gadget. If we can construct a branch-decomposition of width $\mathbf{bw}+O(\log m)$ for the eliminated graph such that there exists an arc a with $N(S) \subseteq \mathbf{mid}(a)$ for each clause gadget with the vertex set S, by replacing the arc a with a caterpillar tree T_F for edges F incident to the clause gadget, we can construct a branch-decomposition of width $\mathbf{bw} + O(\log m) + |N[S]| = \mathbf{bw} + O(\log m)$ for the original graph G. Actually, our construction described below has this property.

For a node i, let Y_i and $\overline{Y_i}$ denote the vertex sets $\{x_i \mid x \in L_i\} \cup \{\overline{x_i} \mid x \in R_i\}$ and $\{\overline{x_i} \mid x \in L_i\} \cup \{x_i \mid x \in R_i\}$, respectively, and let W_i be the set of vertices in the variable gadgets for $\{w_{i,j} \mid j \in [M]\}$. For a uv-counting gadget, let $S_{uv,a}$ be the set of vertices in the variable gadgets for the a-th layer of the counting gadget and S^+_{uv} be the one for the last layer. We denote the set of all the vertices in the counting gadget by S_{uv}. Let V_i denote the union of Y_i, $\overline{Y_i}$, W_i, and S_{ij} for each j.

Let T be a rooted binary tree of the given branch-decomposition of $\mathcal{C}$. We create a new root vertex r' and connect it to the original root. Let f be a

function from arcs of T to subsets of vertices of G such that $f(a) = V_i$ for an arc a between a node i and its parent p. Now, we give an ordering of V_i that satisfies the property desired in Lemma 6.12. If i is a leaf or the child of the root, the size of V_i is $O(\log m)$. Thus, any ordering has the desired property. Otherwise, let p be the parent of i, $\{l, r\}$ be the children of i, and q be the other child of p. From the construction of G, there are edges between V_i and V_j only when i is a parent of j, i is a child of j, or the parent of i and j is the same. Thus, $U_a(f'(a))$ is an empty set and $N_a(f'(a))$ is contained in $Y_i \cup W_i \cup S_{il}^+ \cup S_{ir}^+$, whose size is at most $\mathbf{bw} + O(\log m)$. In what follows, we denote the current set of $N_a(f'(a) \cup \{v_1, \ldots, v_j\})$ by N and $U_a(f'(a) \cup \{v_1, \ldots, v_j\})$ by U, where j gradually increases as we take vertices to make the ordering of $f(a)$.

Let assume that i is a left child of p. First, we take the vertices W_i in an arbitrary order. At this point, N is contained in $Y_i \cup W_p \cup W_q \cup S_{il}^+ \cup S_{ir}^+$ and U is an empty set. Actually, at any point, U is always an empty set when i is a left child. Then, from $j = 1$ to $j = [|X_i \cap X_p| + 1]$, we take all the vertices in $S_{ip,j}$ in an arbitrary order. After that, N is contained in $Y_i \cup W_p \cup W_q \cup S_{il}^+ \cup S_{ir}^+ \cup S_{pi}^+$. Similarly, from $j = 1$ to $j = [|L_i \cap R_q| + 1]$, we take all the vertices in $S_{iq,j}$ in an arbitrary order, which makes N contained in $Y_i \cup W_p \cup W_q \cup S_{il}^+ \cup S_{ir}^+ \cup S_{pi}^+ \cup S_{qi}^+$. Now, we take all the vertices in Y_i in an arbitrary order. After taking a vertex x_i, x_i is removed from N and only $\overline{x_i}$ is included to N. Thus, the size of N is always bounded by $\mathbf{bw} + O(\log m)$. After taking all the vertices of Y_i, N is contained in $\overline{Y_i} \cup W_p \cup W_q \cup S_{il}^+ \cup S_{ir}^+ \cup S_{pi}^+ \cup S_{qi}^+$. Then, from $j = 1$ to $j = |X_i \cap X_l| + 1$, we take all the vertices in $S_{il,j}$ in an arbitrary order. After that, N is contained in $\overline{Y_i} \cup W_p \cup W_q \cup S_{ir}^+ \cup S_{pi}^+ \cup S_{qi}^+$. Similarly, from $j = 1$ to $j = |X_i \cap X_r| + 1$, we take all the vertices in $S_{ir,j}$ in an arbitrary order, which makes N contained in $\overline{Y_i} \cup W_p \cup W_q \cup S_{pi}^+ \cup S_{qi}^+$. Finally, we take all the vertices in $\overline{Y_i}$ in an arbitrary order. After taking a vertex x_i, x_i is removed from N and only one of $\overline{x_p}$ or $\overline{x_q}$ is included to N. Thus, the size of N is always bounded by $\mathbf{bw} + O(\log m)$. Now, we have taken all the vertices in V_i.

When i is a right child of p, we can still use the same ordering and we only need a sight modification of the proof. Let q be the left child of p. After taking all the vertices in W_i, N is contained in $Y_i \cup W_p \cup S_{il}^+ \cup S_{ir}^+$ and U is contained in W_i. After taking all the vertices in S_{ip}, N is contained in $Y_i \cup W_p \cup S_{il}^+ \cup S_{ir}^+ \cup S_{pi}^+$ and U is contained in W_i. After taking all the vertices in S_{iq}, N is contained in $Y_i \cup W_p \cup S_{il}^+ \cup S_{ir}^+ \cup S_{pi}^+$ and U is contained in $W_i \cup S_{iq}^+$. After taking all the vertices in Y_i, N is contained in $\overline{Y_i} \cup W_p \cup S_{il}^+ \cup S_{ir}^+ \cup S_{pi}^+$ and U is contained in $W_i \cup S_{iq}^+$. After taking all the vertices in S_{il}, N is contained in $\overline{Y_i} \cup W_p \cup S_{ir}^+ \cup S_{pi}^+$ and U is contained in $W_i \cup S_{iq}^+$. After taking all the vertices in S_{ir}, N is contained in $\overline{Y_i} \cup W_p \cup S_{pi}^+$ and U is contained in $W_i \cup S_{iq}^+$. Finally, after taking a vertex x_i of $\overline{Y_i}$, x_i is removed from N and moved to U or x_i is removed from N and $\overline{x_p}$ is included to N. At any point, the size of $N \cup U$ is bounded by $\mathbf{bw} + O(\log m)$.

Now, we have obtained the function f and the ordering of $f(a)$ with the desired property. Thus, from Lemma 6.12, the graph G has branch-width at most $\mathbf{bw} + O(\log m)$. □

Chapter 7

Exactly Parameterized NL

In this chapter, we introduce a new parameterized complexity class *EPNL (Exactly Parameterized NL)* and prove robust exponential-time hardness for problems parameterized by path-width. First, in Section 7.1, we formally define EPNL and EPNL-hardness. Intuitively, EPNL is a class of parameterized problems that can be solved by a non-deterministic Turing machine with the space of $k + O(\log n)$ bits, where k is the parameter. If one of the NP-hard problems can be solved in polynomial time, any problem in NP can be solved in polynomial time. Similarly, if one of the EPNL-hard problems can be solved in $O^*(c^k)$ time, any problem in EPNL can be solved in $O^*(c^k)$ time. Since the class EPNL contains many famous problems, such as Set Cover parameterized by the number of elements and Directed Hamiltonicity parameterized by the number of vertices for which no $O^*((2-\epsilon)^n)$-time algorithms are known, EPNL-hardness is much more robust than SETH-hardness which relies only on the hardness of SAT. Examples of problems in EPNL are given in Section 7.2. In Section 7.3, we prove that SAT parameterized by path-width is EPNL-complete. And finally, by using decomposition-based reductions, we show that 3-SAT, Max 2-SAT, and Independent Set parameterized by path-width are also EPNL-complete in Section 7.4. Since path-width is always at least the tree-width, this immediately implies that the problems parameterized by tree-width are EPNL-hard.

7.1 Definitions

By extending the classical complexity class NL (Non-deterministic Logspace), we define a class of parameterized problems EPNL (Exactly Parameterized NL) which can be solved by a non-deterministic Turing machine with the space of $k + O(\log n)$ bits.

Definition 7.1 (EPNL). *A parameterized problem $L \subseteq \Sigma^* \times \mathbb{N}$ is in* EPNL *if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a verifying polynomial-time deterministic Turing machine $M : (\Sigma^* \times \mathbb{N}) \times \Sigma^* \to \{0, 1\}$ with four tapes, a read-only input tape, a read-only read-once certificate tape, and two read/write binary working tapes called the k-bit tape and the logspace tape, satisfying the the following properties for any input $(x, k) \in \Sigma^* \times \mathbb{N}$:*

- *It holds that $(x, k) \in L$ if and only if there exists a certificate $y \in \Sigma^{p(|(x,k)|)}$ such that $M((x, k), y) = 1$.*
- *For any $y \in \Sigma^{p(|(x,k)|)}$, the machine M uses at most k space from the k-bit tape and $O(\log |(x, k)|)$ space from the logspace tape.*

Note that the machine M is not allowed to use $O(k)$ bits from the k-bit tape but at most k bits. This is why we use two separated working tapes instead of one long working tape of length $k + O(\log |(x, k)|)$; in the latter case, because there is only one head, it may be difficult to simulate a random-access k-bit array.

Then, we define *logspace parameter-preserving reduction* and EPNL-hardness.

Definition 7.2 (Reducibility). *A parameterized problem A is* logspace parameter-preserving reducible *to a parameterized problem B, denoted by $A \leq_L^{pp} B$, if there exists a logspace computable function $\phi : (\Sigma^* \times \mathbb{N}) \to (\Sigma^* \times \mathbb{N})$ such that, given an instance (x, k) of A, outputs an instance $(x', k') = \phi(x, k)$ of B such that*

- $(x, k) \in A \iff (x', k') \in B$, *and*
- $k' \leq k + O(\log |(x, k)|)$.

Note that in the standard parameterized reduction, the computation can take $f(k)\text{poly}(|x|)$ time and the parameter k' of the reduced instance can be increased to any function of the original parameter k. However, in our reduction, we allow only a logspace computation and an additive increase by $O(\log |(x, k)|)$ of the parameter.

Proposition 7.1. *If $A \leq_L^{pp} B$ and $B \in$ EPNL, then $A \in$ EPNL.*

The proof of the proposition is an easy extension of the case for NL (see the text book by Arora and Barak [9, Chap.4.3.]), so we omit it here.

Definition 7.3 (EPNL-hardness). *A parameterized problem A is called* EPNL-hard *if for any $B \in$ EPNL, we have $B \leq_L^{pp} A$. Moreover, if $A \in$ EPNL, A is called* EPNL-complete.

Since there are at most $2^{k+O(\log |(x,k)|)} = O^*(2^k)$ configurations of the working tapes, any problem in EPNL can be solved in $O^*(2^k)$ time and space by dynamic programming. The following proposition follows from the definitions.

Proposition 7.2. *Any problem in* EPNL *can be solved in $O^*(2^k)$ time and space. If one of the* EPNL*-hard problem can be solved in $O^*(c^k)$ time and $O^*(d^k)$ space, then any problem in* EPNL *can also be solved in $O^*(c^k)$ time and $O^*(d^k)$ space.*

7.2 Problems in EPNL

We give several examples of problems in EPNL. For all the problems in Lemma 7.2, the current fastest algorithms take $O^*(2^n)$ time [21].

Lemma 7.1. SAT*,* 3-SAT*,* Max 2-SAT*, and* Independent Set *parameterized by path-width are in* EPNL*.*

Proof. We show that SAT parameterized by path-width is in EPNL. For the other problems, we can use similar proofs, so we omit them.

Let $(X_1, \ldots, X_d)$ be the list of bags of the nice path-decomposition from the root to the leaf. As a certificate, we use a list of partial assignments $f_i : X_i \to \{0, 1\}$. Starting from the root bag X_1, the machine M handles each bag one by one as follows. Let X_i be the current bag. By storing the current partial assignment f_i to the k-bit tape, we can check that there are no inconsistencies between two assignments f_i and f_{i-1}. From the definition of path-decomposition, if f_i and f_{i-1} are consistent for all i, all the partial assignments are consistent.

If X_i is an Introduce(C) bag, we check that the partial assignment satisfies the clause C. Since each clause C has an Introduce(C) node, this implies that the assignment given as the certificate satisfies all the clauses. □

Lemma 7.2. Directed Hamiltonicity, Optimal Linear Arrangement, Directed Feedback Arc Set *parameterized by the number of vertices of the input graph, and* Set Cover *parameterized by the number of elements are in* EPNL.

Proof. We show that Directed Hamiltonicity parameterized by the number of vertices is in EPNL. For the other problems, we can use similar proofs, so we omit them. Directed Hamiltonicity is the following problem: given a directed graph $G = (V, E)$ answer whether there exists a cycle that passes each vertex exactly once.

As a certificate, we use an ordering of vertices on the cycle. Then the machine reads each vertex in the ordering one by one. We can check the ordering is actually a cycle by putting the first and the last vertex on the logspace tape. Since the certificate tape is read-once, we cannot check whether each vertex appears exactly once by only using logspace tape. When the machine reads a vertex i from the certificate, it writes a symbol 1 on the i-th position of the k-bit tape. If the symbol in the i-th position is already 1, the certificate contains the vertex i multiple times. Finally, by checking all the symbols in the k-bit tape is 1, we can confirm that each vertex appears exactly once in the ordering. □

7.3 EPNL-completeness of SAT Parameterized by Path-width

In this section, we prove that SAT parameterized by path-width is EPNL-complete by showing that it is logspace parameter-preserving reducible from any parametrized problem in EPNL.

Theorem 7.1. SAT *parameterized by path-width is* EPNL*-complete.*

Proof. SAT parameterized by path-width is in EPNL. So it suffices to show that any parameterized problem $A \in$ EPNL can be reduced to SAT parameterized by path-width. Let M be a Turing machine that accepts A, Q be the set of (internal) states of M, and $t, s : \mathbb{N} \to \mathbb{N}$ be the polynomial time bound and logarithmic space bound of M, respectively. For simplicity, we assume that inputs and certificates of M are encoded as binary strings. We reduce an instance (x, k) of A to SAT as follows.

Let n be the input length. For each step $i \in [t(n)]$, we create the following variables:

- $Q_{i,q}$ for each $q \in Q$, which indicates that M is in state q,
- $H^I_{i,j}$ for each $j \in [\lceil \log n \rceil]$, which indicates the position of the input tape head in binary,
- $H^K_{i,j}$ for each $j \in [\lceil \log k \rceil]$, which indicates the position of the k-bit tape head in binary,
- $H^L_{i,j}$ for each $j \in [\lceil \log r(n) \rceil]$, which indicates the position of the logspace tape head in binary,
- $T^K_{i,h}$ for each $h \in [k]'$, which indicates the symbol written in the h-th cell of the k-bit tape,

- $T^L_{i,h}$ for each $h \in [s(n)]'$, which indicates the symbol written in the h-th cell of the logspace tape, and
- T^C_i, which indicates the symbol in the cell of the certificate tape.

Now, we create clauses. Let $q_s \in Q$ be the initial state and $q_t \in Q$ be the accepting state. First, we create the following clauses (consisting of single literals) to express the initial and the final configuration:

- Q_{1,q_s} (the machine is in the state q_s),
- $\overline{H^I_{1,j}}$ for each $j \in [\lceil \log n \rceil]$ (the input tape head is at the position 0),
- $\overline{H^K_{1,j}}$ for each $j \in [\lceil \log k \rceil]$ (the k-bit tape head is at the position 0),
- $\overline{H^L_{1,j}}$ for each $j \in [\lceil \log s(n) \rceil]$ (the logspace tape head is at the position 0),
- $\overline{T^K_{1,h}}$ for each $h \in [k]'$ (each cell of the k-bit tape has symbol 0),
- $\overline{T^L_{1,h}}$ for each $h \in [r(n)]'$ (each cell of the logspace tape has symbol 0), and
- $Q_{t(n),q_t}$ (the machine must finish in the state q_t).

Then, for each step $i \in [t(n)]$, we create clauses to express transitions as follows. The machine can take only one state at a time, so we create a clause $\overline{Q_{i,q}} \vee \overline{Q_{i,q'}}$ for each $q \neq q'$. If a cell changes, the head must be there (or equivalently, cells not pointed by the head must remain unchanged), so we create the following clauses:

- $T^K_{i,h^K} \neq T^K_{i+1,h^K} \to (H^K_{i,*})_2 = h^K$ for each $h^K \in [k]'$, and
- $T^L_{i,h^L} \neq T^L_{i+1,h^L} \to (H^L_{i,*})_2 = h^L$ for each $h^L \in [s(n)]'$.

Let $\delta : (q, c^I, c^K, c^L, c^C) \mapsto (q', c'^K, c'^L, d^I, d^K, d^L, d^C)$ be the transition function, which indicates that if the machine is in the state q, the symbol in the input tape is c^I, the symbol in the k-bit tape is c^K, the symbol in the logspace tape is c^L, and the symbol in the certificate tape is c^C, then the machine changes the state to q', write c'^K to the cell of the k-bit tape, write c'^L to the cell of the logspace tape, move the input tape head by d^I, move the k-bit tape head by d^K, move the logspace tape head by d^L, and move the certificate tape head by d^C. Note that since the certificate tape is read-once, $d^C \geq 0$. For each $h^I \in [n]'$, $h^K \in [k]'$, $h^L \in [s(n)]'$, and transition $(q, c^I, c^K, c^L, c^C) \mapsto (q', c'^K, c'^L, d^I, d^K, d^L, d^C)$, we create clauses as follows. If a symbol in the h^I-th position of the input tape is not c^I, this transition never occurs. Otherwise, let C be the constraint $Q_{i,q} \wedge (H^I_{i,*})_2 = h^I \wedge (H^K_{i,*})_2 = h^K \wedge (H^L_{i,*})_2 = h^L \wedge T^K_{i,h^K} = c^K \wedge T^L_{i,h^L} = c^L \wedge T^C_i = c^C$, which express the condition that the transition occurs. Then, we create the following clauses:

- $C \to Q_{i+1,q'}$ (the machine changes the state to q'),
- $C \to T^K_{i+1,h^K} = c'^K$ (c'^K is written in the cell of the k-bit tape),
- $C \to T^L_{i+1,h^L} = c'^L$ (c'^L is written in the cell of the the logspace tape),
- $C \to (H^I_{i+1,*})_2 = h^I + d^I$ (the input tape head moves by d^I),

- $C \to (H^K_{i+1,*})_2 = h^K + d^K$ (the k-bit tape head moves by d^K),
- $C \to (H^L_{i+1,*})_2 = h^L + d^L$ (the logspace tape head moves by d^L), and
- $C \to T^C_i = T^C_{i+1}$ if $d^C = 0$ (if the certificate tape head does not move, then the symbol in the certificate tape does not change).

It is not difficult to check that the reduction can be done in logspace and the obtained CNF is satisfiable if and only if there is a certificate such that the machine finishes in the accepting state. Finally, we show that the obtained CNF has path-width $k + O(\log n)$.

For a step i, let $T^K_i = \{T^K_{i,h} \mid h \in [k]'\}$ and X_i be the set of other variables. The primal graph of the obtained CNF has the following properties:

- $N[X_i] \subseteq T^K_{i-1} \cup X_{i-1} \cup T^K_i \cup X_i \cup T^K_{i+1} \cup X_{i+1}$,
- $N(T^K_{i,j}) \subseteq \{T^K_{i-1,j}, T^K_{i+1,j}\} \cup X_{i-1} \cup X_i \cup X_{i+1}$.

We can construct a path-decomposition as follows: starting from a bag $T^K_1 \cup X_1$ and $i = 1$, introduce X_{i+1}, introduce $T^K_{i+1,1}$, forget $T^K_{i,1}$, ..., introduce $T^K_{i+1,k}$, forget $T^K_{i,k}$, forget X_i (the current bag consists of $T^K_{i+1} \cup X_{i+1}$), and then increase i. Since the size of X_i is $O(\log n)$ and the size of T^K_i is exactly k, the width of the obtained path-decomposition is $k + O(\log n)$. □

7.4 EPNL-completeness of Problems Parameterized by Path-width

Finally, we show that 3-SAT, Independent Set, and Max 2-SAT parameterized by path-width are EPNL-complete.

Theorem 7.2. 3-SAT *parameterized by path-width is* EPNL-*complete.*

Proof. We prove the theorem by a reduction from SAT parameterized by path-width. The reduction is completely the same as the standard reduction (see Subsection 6.3.3). Starting from an empty bag and the leaf node i of the given nice path-decomposition of width $\mathbf{pw}$, we can construct a path-decomposition of the reduced instance as follows. If i is an Introduce(C) node of length more than three, let $\{y_1, \dots, y_k\}$ be the variables created to replace the clause C. Then, we introduce y_1, introduce y_2, forget y_1, introduce y_3, forget y_2, ..., introduce y_k, forget y_{k-1}, and forget y_k. If i is an Introduce(x) node, we introduce x, and if i is a Forget(x) node, we forget x. Finally, we change i to its parent and repeat the process until reaching to the root. The width of this path-decomposition is $\mathbf{pw} + O(1)$. □

Theorem 7.3. Independent Set *parameterized by path-width is* EPNL-*complete.*

Proof. We prove the theorem by a reduction from 3-SAT parameterized by path-width. The reduction is completely the same as that for the tree-width case (Subsection 6.3.2), so we only need to bound the path-width of the obtained graph. Starting from an empty bag and the leaf node i of the given nice path-decomposition of width $\mathbf{pw}$, we can construct a path-decomposition of the reduced instance as follows.

If i is not the leaf, let c be the child of i. For each variable $x \in X_i \cap X_c$, we introduce x_i and forget $\overline{x_c}$. If i is an Introduce(x) node, we introduce x_i, if i is a

Forget(x) node, we forget $\overline{x_c}$, and if i is an Introduce(C) node, we introduce the corresponding clause gadget.

Then, we process the child counting gadget for i as follows. First, we introduce the first layer $S_{i,1}$. Then, starting from $a = 1$, we repeat the following process by incrementing a: (1) introduce the next layer $S_{i,a+1}$, (2) for each clause gadget C connecting $S_{i,a}$ and $S_{i,a+1}$, introduce C and forget C one by one, (3) forget the current layer $S_{i,a}$. Note that the last layer of the counting gadget is remained in the bag.

Next, for each variable $x \in X_i$, we introduce $\overline{x_i}$ and forget x_i one by one. If i is an Introduce(C) node, we forget the corresponding clause gadget. We process the parent counting gadget for c in the same way as we did for the child counting gadget. Then, we process the last layers of the child and the parent counting gadget for c. For each clause gadget C connecting the last layers of the child and the parent counting gadget, we introduce C and forget C one by one, and then we forget these two layers . Finally, we change i to its parent and repeat the process until reaching to the root. The width of this path-decomposition is $\mathbf{pw} + O(\log \mathbf{pw})$. □

Theorem 7.4. MAX 2-SAT *parameterized by path-width is* EPNL*-complete.*

Proof. We prove the theorem by a reduction from INDEPENDENT SET parameterized by path-width. The proof is completely the same as that for the tree-width case (Subsection 6.3.4) □

Chapter 8

Decomposition-based Reductions in Practice

In this chapter, we investigate the practical performance of the decomposition-based reduction presented in Chapter 6. As a benchmark problem, we choose a reduction from Max SAT to SAT because practical methods for Max SAT are well studied and the state-of-the-art methods rely on reductions to SAT. First, in Section 8.1, we review the existing research on Max SAT. Then, we propose our decomposition-based reduction in Section 8.2. Finally, in Section 8.3, we explain our experimental results.

8.1 Review of the Existing Research on Max SAT

Max SAT is the optimization version of SAT where the objective is to maximize the number of satisfied clauses. Because Max SAT can encode many optimization problems more naturally than SAT, the development of practical MaxSAT solvers has attracted a lot of attention in recent years. Since 2006, annual competitions of Max SAT solvers (*MaxSAT Evaluation*[1]) have been held and a variety of new solvers have been developed. In this section, we review the existing approach for practical Max SAT solving.

When encoding an optimization problem to Max SAT, two types of clauses occur: 1) the clauses encoding the constraints of the problem and 2) the clauses introduced for expressing the objective function. For example, if we encode Independent Set problem to Max SAT, we create 1) a clause $(\overline{x_u} \vee \overline{x_v})$ for each edge $uv \in E$, which expresses the constraint that at most one of u and v can be contained in an independent set, and 2) a clause (x_v) for each vertex $v \in V$, which expresses the objective function for the vertex v. We call the first type of the clauses, which must be satisfied, as the *hard clauses* and the second type of the clauses, which can be unsatisfied, as *soft clauses*. *Partial Max SAT* is a variant of Max SAT in which we are given a list of hard clauses and soft clauses and the objective is to satisfy as many soft clauses as possible while satisfying all the hard clauses. Although we can simulate a hard clause using soft clauses by creating a sufficient number of copies, for practical performance, solvers for Partial Max SAT have been well studied. There is a more general version *Weighted Partial Max SAT*. However, in our research, we concentrate on the unweighted version for simplicity. We denote the set of soft clauses by $\mathcal{S} = \{S_1, \dots, S_m\}$ and the set of hard clauses by $\mathcal{H} = \{H_1, \dots, H_{m'}\}$. For convenience, we define Partial Max SAT as the minimization problem where the objective value is the number of unsatisfied soft clauses.

[1] `http://www.maxsat.udl.cat/`

8.1.1 Algorithms for Partial Max SAT

There are two types of major approaches for PARTIAL MAX SAT: 1) branch-and-bound-based approach and 2) SAT-based approach which reduces the problem to SAT and exploits the state-of-the-art SAT solvers. Although branch-and-bound-based approach performs well on *random* and *crafted* instances, the current SAT-based approach strongly outperforms the branch-and-bound-based approach on *industrial* instances. Note that we are interested in the performance on industrial instances because they are considered to be more structured than random or crafted instances. SAT-based approach can be further classified into two categories: 1) the *satisfiability-based* approach [15] and 2) the *unsatisfiability-based* approach [41].

Satisfiability-based Approach

The satisfiability-based approach is based on a simple reduction to SAT. Let k be a known upper bound of the number of unsatisfied soft clauses. By using a SAT solver, we try to find an assignment such that at most $k-1$ soft clauses are unsatisfied as follows. For each soft clause S_i, we create a *blocking variable* b_i and replace the clause with $S_i \vee b_i$. If the clause S_i is unsatisfied, the blocking variable b_i must be set to true. Then we encode the constraint $b_1 + \ldots + b_m < k$, which is called a *cardinality constraint*, to CNF. If the obtained CNF is unsatisfied, the optimal value is k. Otherwise, we can update the upper bound k and repeat the process. The initial upper bound can be computed just by introducing the blocking variables without constructing the cardinality constraint. The important part of the approach is how to encode the cardinality constraint. There are many studies on the encoding, which are reviewed in the next subsection.

In this approach, we repeatedly solve almost the same instances of SAT. The only difference is the right-hand side of the cardinality constraint. In many encoding methods, we can update the constraint by just adding a small number of clauses, or even by setting a single variable to false. Since the state-of-the-art SAT solvers are based on clause learning [95], we can reuse the clauses learned in the previous executions. Actually, most of the famous SAT solvers, including MiniSAT, support this incremental solving.

Solvers based on this approach won the first place in MaxSAT Evaluations 2008 [15] and 2010–2012 [66], and the second place[2] in 2013 [83].

Unsatisfiability-based Approach

Most of the current SAT solvers have a feature to extract an unsatisfiable subset of clauses (called a *core*) when the input CNF is unsatisfiable. In the satisfiability-based approach, we iteratively update the upper bound by finding satisfiable assignments. Contrastingly, in unsatisfiability-based approach, we iteratively update the lower bound by exploiting the cores. Thus, the approach is also called the *core-guided approach*.

We review a simple unsatisfiability-based approach by Fu and Malik [41]. Let $\mathcal{S}' = \{S'_1, \ldots, S'_p\}$ be a set of soft clauses in the extracted core. Since any assignment that satisfies all the hard clauses cannot satisfy at least one of $\mathcal{S}'$, we relax $\mathcal{S}'$ as follows. For each clause $S'_i \in \mathcal{S}'$, we introduce a blocking variables b_i and replace the clause with $S'_i \vee b_i$. Then, we encode the constraint $b_1 + \ldots + b_p =$

[2] *Portfolio* solvers, which use multiple independent solvers and select a solver with the best predicted performance for each instance, won the first place in 2013–2015 [6].

1. We iterate the process until the CNF becomes satisfiable. There have been developed many improved methods for relaxing the unsatisfiable core [76, 75, 77, 4, 48, 80, 5].

Solvers based on this approach won the first place in MaxSAT Evaluations 2009 [4], and the second place in 2014 [78] and 2015 [5].

Although, recent unsatisfiability-based solvers outperform satisfiability-based solvers, in our research, we focus on satisfiability-based approach because it is based on the simple reduction to SAT and therefore reduction techniques would strongly affect the performance. We note that the objective of our research is not to develop the fastest MaxSAT solver but to investigate the practical performance of the decomposition-based reduction.

8.1.2 Encoding of Cardinality Constraints

A constraint of the form $x_1 + \ldots + x_n < k$ is called a *cardinality constraint.* Since cardinality constraints appear in many practical problems, there are many research on CNF encoding of cardinality constraints, such as Totalizer [13], Modulo Totalizer [83], and Cardinality Networks [10]. Among them, we review Totalizer. Although in our decomposition-based reduction from Max 2-SAT to SAT in Subsection 6.3.1, we used a binary representation of an integer to encode the cardinality constraint, this encoding is rarely used in practice. This is because the binary encoding does not preserve *arc consistency.* In the constraint $x_1+\ldots+x_n < k$, if k variables among the x_i's are set to true, we can immediately know that the constraint is unsatisfied. If the encoding preserves arc consistency, we can drive an empty clause by the *unit propagation* as soon as k variables are set to true. Here, the unit propagation is the core of the DPLL algorithm [35], which is the basis of the state-of-the-art SAT solvers. Thus, preserving arc consistency is considered to be important for practical performance.

Totalizer

Totalizer was developed by Bailleux and Boufkhad [13] and proved to preserve arc consistency. In order to represent the sum, it uses a unary representation of an integer. Let $X = \{x_1, \ldots, x_n\}$ be a variable set and U be a list of variables representing the sum (e.g., the number of the true variables in X). If the sum is i, the first i variables of U are set to true and the other variables are set to false.

In the encoding, we recursively compute the sum $x_1 + \ldots + x_n$ as described in Algorithm 8. If $n = 1$, the variable x_1 itself represents the sum. Otherwise, we split the variables X into two halves $L = \{x_1, \ldots, x_{\lfloor \frac{n}{2} \rfloor}\}$ and $R = \{x_{\lfloor \frac{n}{2} \rfloor+1}, \ldots, x_n\}$. Let $U_L = (l_1, \ldots, l_{n_L})$ and $U_R = (r_1, \ldots, r_{n_R})$ be the lists of variables representing the sums of L and R, respectively, computed by the recursive calls. Then, by using a subroutine described in Algorithm 9, we introduce a list of variables $U = (u_1, \ldots, u_{n'})$ representing the sum of X as follows. First, for each $i \in [n_L]$, we create a clause $\bar{l_i} \vee u_i$, which is equivalent to $l_i \to u_i$. This clause expresses the constraint that if the sum of L is at least i, then the sum of X is also at least i. Similarly, we create clauses $\bar{r_j} \vee u_j$. Then, for each $i \in [n_L]$ and $j \in [n_R]$, we create a clause $\bar{l_i} \vee \bar{r_j} \vee u_{i+j}$, which is equivalent to $l_i \wedge r_j \to u_{i+j}$. This clause expresses the constraint that if the sum of L is at least i and the sum of R is at least j then the total is at least $i + j$.

Finally, by adding a clause $(\overline{u_k})$, we can express the constraint $x_1+\ldots+x_n < k$. Here, we note that we can update the right-hand side value k only by setting the

Algorithm 8 Totalizer [13]

INPUT: variables $X = \{x_1, \ldots, x_n\}$
OUTPUT: a list of variables representing the sum

```
1: procedure Totalizer(X)
2:     if n = 1 then return (∅, (x_1))
3:     split X into two halves L ∪ R
4:     U_L ← Totalizer(L)
5:     U_R ← Totalizer(R)
6:     return UnaryAdder(U_L, U_R)
```

Algorithm 9 Unary Adder

INPUT: two lists of variables $U_L = (l_1, \ldots, l_{n_L})$ and $U_R = (r_1, \ldots, r_{n_L})$
OUTPUT: a list of variables representing the sum

```
1: procedure UnaryAdder(U_L, U_R)
2:     n' ← min(n_L + n_R, k)
3:     introduce new variables U = (u_1, ..., u_{n'})
4:     for i ∈ [n_L] do
5:         introduce a clause (¬l_i ∨ u_i)                 ▷ l_i → u_i
6:     for j ∈ [n_R] do
7:         introduce a clause (¬r_j ∨ u_j)                 ▷ r_j → u_j
8:     for i ∈ [n_L] do
9:         for j ∈ [min(n_R, n' − n_L)] do
10:            introduce a clause (¬l_i ∨ ¬r_j ∨ u_{i+j})  ▷ (l_i ∧ r_j) → u_{i+j}
11:    return U
```

corresponding variable to false. The encoding introduces $O(n \log n)$ variables and $O(n^2)$ clauses. Since we only need to determine whether the sum is smaller than k or not, it suffices to introduce $\min(n, k)$ variables to represent a sum of n variables. By this improvement, the number of introduced variables and clauses are bounded by $O(n \log k)$ and $O(nk)$, respectively.

8.2 Decomposition-based Reduction

Now, we apply the idea of the decomposition-based reduction to the reduction from Partial Max SAT to SAT. Although the basic idea is almost the same as the tree-decomposition-based reduction from Max 2-SAT to SAT presented in Subsection 6.3.1, for practical performance, we need to modify it as follows.

First, in the presented reduction, we created a new variable x_i for each bag i and each variable $x \in X_i$. However, this is only for describing the proofs in a uniform format, and actually without replicating variables, we can obtain the same tree-width bound.

Second, we used the binary representation of integers, which may not be practical. Thus, we use the unary representation used in the Totalizer encoding. Since the unary representation requires k variables to express the number of unsatisfied clauses, where k is the initial upper bound of the number of unsatisfied clauses, we cannot obtain $\mathbf{tw} + O(\log n)$ bound of the tree-width. However, we note that the dynamic programming algorithm can still solve the reduced instance in $O^*(2^{\mathbf{tw}})$ time because there are only $O(k)$ possible satisfying assignments for these unary representation variables. Thus, we can assume that the reduced instance *virtually* has a tree-width $\mathbf{tw} + O(\log n)$.

Algorithm 10 Decomposition-based Reduction

INPUT: a node i of the nice tree-decomposition of the input CNF
OUTPUT: a unary representation of the number of unsatisfied clauses in $\mathcal{S}(i)$

```
 1: procedure REDUCE(i)
 2:     if i is an Introduce(S_i) node for a soft clause S_i then
 3:         introduce a new variable b_i and replace S_i with S_i ∨ b_i.
 4:         if S(c(i)) = ∅ then return (b_i)
 5:         else return UNARYADDER((b_i), REDUCE(c(i)))
 6:     else if i is a Join node then
 7:         if both of S(l(i)) and S(r(i)) are non-empty then
 8:             return UNARYADDER(REDUCE(l(i)), REDUCE(r(i)))
 9:         else
10:             return REDUCE(c), where c is the child satisfying S(c) ≠ ∅
11:     else
12:         return REDUCE(c(i))
```

Now, we explain our tree-decomposition-based reduction from Partial Max SAT to SAT. First, we compute a nice tree-decomposition of the primal graph of the input CNF. For a node i, let $\mathcal{S}(i)$ denote the set of soft clauses contained in the subtree rooted at i. If i is a Join node, we denote the set of two children of i by $\{l(i), r(i)\}$, and if i is neither a Join node nor a Leaf node, we denote the unique child of i by $c(i)$.

Starting from the root node, we recursively apply Algorithm 10. If the current node i is an Introduce(S_i) node for a soft clause S_i, we create a blocking variable b_i and relax the clause. If no other soft clauses are contained in the subtree, (b_i) is the unary representation of the number of unsatisfied clauses. Otherwise, we recursively compute a unary representation for the child node $c(i)$ and return the sum of them by calling the subroutine UnaryAdder (Algorithm 9). If the current node i is a Join node such that both of $\mathcal{S}(l(i))$ and $\mathcal{S}(r(i))$ are not empty, we recursively compute unary representations for both children and then return the unary representation of their sum. If none of the above cases are met, we move to the child without doing anything.

Let $(u_1, \dots, u_k)$ be the obtained unary representation. By adding a clause $(\overline{u_k})$, we can encode the constraint that the number of unsatisfied soft clauses is at most k. The proposed decomposition-based reduction can be seen as a variant of satisfiability-based method with Totalizer encoding; instead of splitting blocking variables into two halves, we split them according to the tree-decomposition. There is one drawback in our reduction; since Totalizer encoding splits a variable set into two sets of similar size, the number of introduced variables is bounded by $O(m \log k)$, where m is the number of soft clauses; on the other hand, in our reduction, a variable set of size t might be split into two sets of size 1 and $t-1$, and thus it may introduce $O(mk)$ variables.

In order to overcome the drawback, we propose the following improvement called *Lazy Addition*. Instead of using a single unary representation U_i to count the number of unsatisfied clauses in $\mathcal{S}(i)$, we use a *set* of unary representations $\mathcal{U}_i = \{U_i^1, \dots, U_i^{k'}\}$. We ensure that every set $\mathcal{U}_i$ satisfies the following three conditions: 1) the size (i.e., the number of variables) of each unary representation in $\mathcal{U}_i$ is a power of 2 or exactly k, 2) all the unary representations in $\mathcal{U}_i$ have distinct size, and 3) the sum of all the unary representations in $\mathcal{U}_i$ is equal to the number of unsatisfied soft clauses in $\mathcal{S}(i)$. Then, the UnaryAdder in Algorithm 10 is

Algorithm 11 Lazy Unary Adder

INPUT: two sets of unary representations $\mathcal{U}_L$ and $\mathcal{U}_R$
OUTPUT: a set of unary representations

1: **procedure** LAZYUNARYADDER($\mathcal{U}_L$,$\mathcal{U}_R$)
2: $\quad \mathcal{U} \leftarrow \mathcal{U}_L \cup \mathcal{U}_R$
3: $\quad$ **while** $\exists A, B \in \mathcal{U}$ with $|A| = |B|$ **do**
4: $\quad\quad \mathcal{U} \leftarrow \mathcal{U} \setminus \{A, B\} \cup \{\text{UNARYADDER}(A, B)\}$
5: $\quad$ **return** $\mathcal{U}$

replaced with LAZYUNARYADDER described in Algorithm 11. When computing a sum of $\mathcal{U}_L$, $\mathcal{U}_R$, we first take the union $\mathcal{U} = \mathcal{U}_L \cup \mathcal{U}_R$. Let $A, B \in \mathcal{U}$ be two unary representations of the same size. If there exists no such pair, $\mathcal{U}$ is a set of unary representations satisfying the above conditions. Otherwise, we remove them from the set and insert the sum UNARYADDER(A, B) to the set. We repeat the process while such a pair exists.

We note that with this improvement, it takes $O^*(2^{\mathbf{tw}} k^{\log k})$ time to solve the reduced CNF by the dynamic programming because in each bag, there are $O(\prod_{i=0}^{\lceil \log k \rceil} 2^i) = O(k^{\log k})$ possible satisfying assignments for the set of unary representations. Thus it might be possible that the simple version would perform well in practice. We will compare the two versions in the experiments.

Lemma 8.1. *The decomposition-based reduction with the lazy addition introduces $O(m \log k)$ variables and $O(mk)$ clauses.*

Proof. We construct a binary tree T, called an *addition tree*, as follows; for each unary representation U_i created by the reduction, we create a corresponding node i; if U_i was introduced as a sum of two unary representations U_l and U_r, we set $\{l, r\}$ as the children of the node i. We define the size of a node i as the size of the corresponding unary representation U_i. Let T_k be the subtree of T consisting of nodes whose corresponding unary representations have size k.

First, we bound the number of introduced variables and clauses inside T_k. Since each unary representation corresponding to a leaf node of T_k counts the number of unsatisfied clauses in a distinct set of soft clauses, the number of the leaves of T_k is bounded by $O(m/k)$. Therefore, the number of nodes in T_k is also bounded by $O(m/k)$. Thus, the number of introduced variables is $O(m/k \cdot k) = O(m)$ and the number of introduced clauses is $O(m/k \cdot k^2) = O(mk)$.

Finally, we bound the number of introduced variables and clauses outside T_k. Let r be a leaf of T_k and $T(r)$ be a subtree of T rooted at the node r. Each leaf node of $T(r)$ has size 1 and each non-leaf node i has two children of size $|U_i|/2$. Therefore, the subtree $T(r)$ forms a complete binary tree and the number of the leaves is bounded by $O(k)$. Thus, the number of introduced variables inside $T(r)$ is $\sum_{i=0}^{\lceil \log_2 k \rceil} O(k/2^i \cdot 2^i) = O(k \log k)$ and the number of introduced clauses inside $T(r)$ is $\sum_{i=0}^{\lceil \log_2 k \rceil - 1} O((k/2^i)^2 \cdot 2^i) = O(k^2)$. Since the number of leaves of T_k is $O(m/k)$, the total number of introduced variables and clauses outside T_k is $O(m/k \cdot k \log k) = O(m \log k)$ and $O(m/k \cdot k^2) = O(mk)$, respectively, $\square$

8.3 Experiments

8.3.1 Setup

Environment

Experiments were conducted on a machine with Intel Xeon X5670 (2.93 GHz) and 48GB of main memory. All the methods were implemented in C++ by extending the MiniSAT[3] version 2.2.0. All the timing results were sequential. We set the time limit for each execution as 30 minutes (= 1,800 seconds).

Datasets

We used benchmark datasets that were used in the Industrial Partial Max-SAT category of MaxSAT Evaluation 2015[4]. The benchmark consists of 14 sets of instances came from different types of applications: aes, atcoss, bcp, circuit-trace-compaction, close_solutions, des, haplotype-assembly, hs-timetabling, mbd, packup-pms, pbo, protein_ins, tpr, and treewidth-computation. The number of variables, the number of clauses, and the number of soft clauses of each instance are shown in Figures 8.1 and 8.2, where each point corresponds to a single instance.

Methods

We implemented the following four reductions.

Totalizer: This is a satisfiability-based method with Totalizer encoding. This combination was used by QMaxSAT [66], which won the first place in MaxSAT Evaluations 2010–2012. When splitting blocking variables into two halves at the line 3 of the Algorithm 8, we use the input ordering; i.e., we number blocking variables from 1 to m in the order in which the corresponding soft clauses appear in the input, and then at each recursive call, we split the variables into the first half and the second half.

Totalizer(Shuffle): Instead of using the input ordering of the blocking variables, in this version, we use the random ordering for investigating the impact of the ordering.

Decomposition-based: This is a simple version of the decomposition-based reduction that does not use the lazy addition. In order to use our tree-decomposition-based reductions, we need to compute a tree-decomposition of the primal graph of the input CNF. Since computing the optimal tree-decomposition is NP-hard, we use a *min-degree heuristic* [16] with a *star-based representation* [74], which was developed by Maehara, Akiba, Iwata, and Kawarabayashi for computing tree-decompositions of very large real-world graphs. In our experiments, the time required to compute a primal graph and its tree-decomposition is also included to the running time.

Decomposition-based(Lazy): This is an improved version of the proposed decomposition-based reduction that uses the lazy addition.

[3] `http://minisat.se/`
[4] `http://maxsat.ia.udl.cat/`

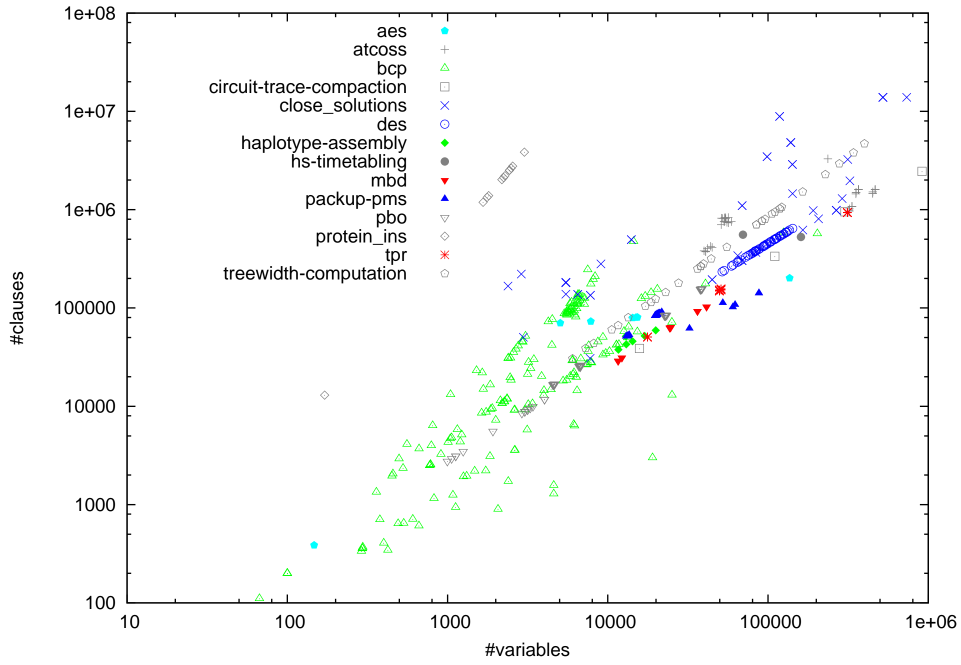

Figure 8.1: The number of variables and the number of clauses

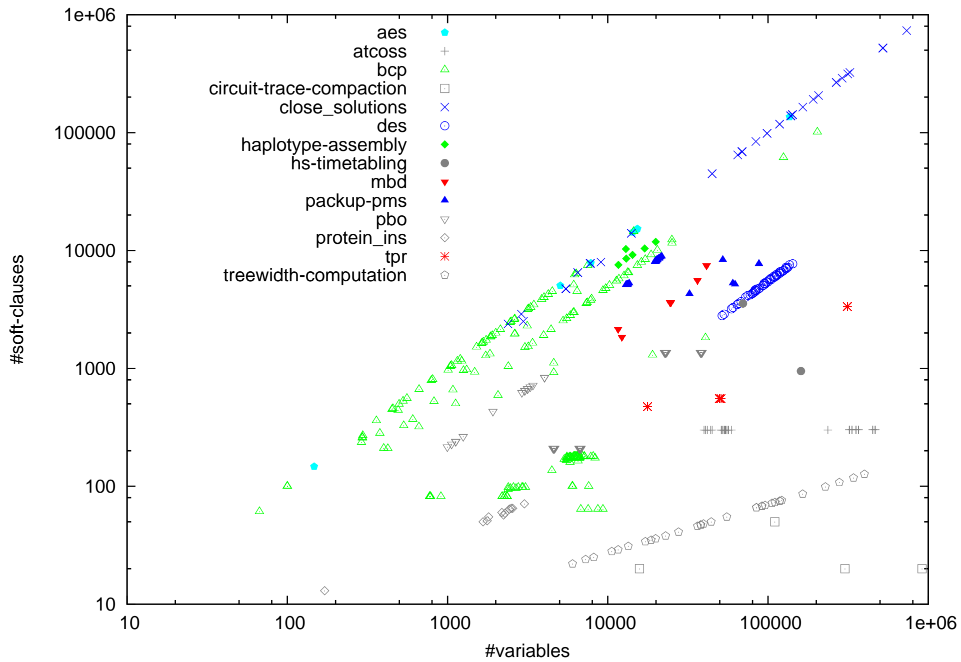

Figure 8.2: The number of variables and the number of soft clauses

8.3.2 Comparison and Analysis

We compare the performance of the four reductions. Table 8.1 shows the number of solved instances within the time limit and the average running time. Here, when taking an average, a running time for an unsolved instance is regarded as 30 minutes. From the table, we can see that the difference of the datasets strongly affects the performance.

Table 8.1: Comparison of the four reductions. #I denotes the number of instances in a dataset, #S denotes the number of solved instances, and T denotes the average running time (in seconds).

Dataset	#I	Totalizer #S	Totalizer T	+Shuffle #S	+Shuffle T	Dec.-based #S	Dec.-based T	+Lazy #S	+Lazy T
aes	7	0	1800.0	0	1800.0	0	1800.0	0	1800.0
atcoss	37	23	805.4	23	798.7	23	813.4	23	806.5
bcp	188	118	695.6	68	1184.2	116	730.6	121	688.9
circuit-trace-compaction	4	4	35.7	4	40.2	4	43.2	4	34.7
close_solutions	50	24	1028.9	18	1286.0	7	1660.5	17	1289.9
des	50	44	497.9	2	1760.1	19	1189.6	21	1102.6
haplotype-assembly	6	5	349.5	0	1800.0	5	385.9	5	351.2
hs-timetabling	2	1	1051.4	1	1010.0	1	1036.3	1	1043.2
mbd	46	35	703.6	0	1800.0	45	104.2	45	100.6
packup-pms	40	40	9.8	5	1576.3	40	125.4	39	147.3
pbo	65	65	54.6	64	110.4	58	247.8	64	103.1
protein_ins	12	12	437.2	11	736.4	9	950.3	12	206.1
tpr	61	61	247.3	56	615.1	61	90.9	61	77.7
treewidth-computation	33	26	464.7	26	426.9	26	461.2	26	479.2
Total	601	458	542.1	278	1070.8	414	640.9	439	559.3

Practical Importance of Preserving Tree-width

Although the decomposition-based reduction could not always outperform Totalizer, we can still show the practical importance of preserving the tree-width. We arrange datasets other than aes, of which all the reductions could not solve any instances, into the following four groups.

A. The group *A* consists of six datasets atcoss, circuit-trace-compaction, hs-timetabling, pbo, protein_ins, and treewidth-computation, on which Totalizer, Totalizer(Shuffle), and Decomposition-based(Lazy) show similar performance. In figures, these datasets will be colored in gray.

B. The group *B* consists of two datasets bcp and haplotype-assembly, on which Totalizer and Decomposition-based(Lazy) show similar performance, and moreover they outperform Totalizer(Shuffle). In figures, these datasets will be colored in green.

C. The group *C* consists of three datasets close_solutions, des, and packup-pms, on which Totalizer outperforms Decomposition-based(Lazy). In figures, these datasets will be colored in blue.

D. The group *D* consists of two datasets mbd and tpr, on which Decomposition-based(Lazy) outperforms Totalizer. In figures, these datasets will be colored in red.

We expect the reason why there exists such a difference as follows.

A. All the datasets other than pbo in this group have relatively small number of soft clauses (see Figure 8.2). Thus a reduction modifies only a small part of an instance, which does not seriously affect the performance.

B. On these datasets, the input ordering may unintentionally preserve the tree-width. For example, if the primal graph of the input CNF is a $\sqrt{n} \times \sqrt{n}$ grid (whose tree-width is $O(\sqrt{n})$) and the clauses are ordered from top to bottom (which is a very natural ordering), then the reduced CNF by Totalizer using the input ordering also has tree-width $O(\sqrt{n})$. If so, both

Totalizer and Decomposition-based(Lazy) can preserve the tree-width and therefore they outperform Totalizer(Shuffle) which may not preserve the tree-width.

C. On these datasets, instances may admit more important (unknown) structures than the tree-width (i.e., tree-like-ness). Since the datasets are created by encoding other problems, the input ordering might be determined by using a knowledge of the original problem. Thus, by using the input ordering, such nice structures might be preserved. Actually, on these datasets, Totalizer strongly outperforms Totalizer(Shuffle). Thus, our expectation is confirmed. On the datasets des and packup-pms, Decomposition-based(Lazy) outperforms Totalizer(Shuffle) because Totalizer(Shuffle) may not preserve neither such nice structures nor tree-width.

D. On these datasets, the input ordering may not preserve the tree-width, and moreover, instances may not admit other nice structures (or do admit nice structures but the input ordering may not preserve them). Since Totalizer outperforms Totalizer(Shuffle), the input ordering may somewhat preserve the tree-width. On tpr, the difference of the performance is small because it has relatively small number of soft clauses.

In order to confirm the above expectations, we will precisely analyze the performance. Figure 8.3 shows the detailed comparison of the running times of Decomposition-based(Lazy) and Totalizer, where a point with a coordinates (x, y) means that there is an instance for which Decomposition-based(Lazy) takes x seconds and the Totalizer takes y seconds. Here, only instances solved by at least one of the reductions are plotted. We note that no instances of aes dataset are plotted because our reductions solved none of them. If one of the reductions could not solve an instance within the time limit, we plot it with a random coordinate between 3000 and 6000 to avoid overlapping. From the figure, we can see that not only when taking an average, but also when focusing on each instance separately, the performance strongly rely on a dataset the instance belongs to; for most of the instances in the group C (blue), Totalizer outperforms Decomposition-based(Lazy), and for most of the instances in the group D (red), Decomposition-based(Lazy) outperforms Totalizer.

Figure 8.4 shows the comparison of the running times of Decomposition-based(Lazy) and Totalizer(Shuffle). In contrast to the previous comparison, when using a random ordering, Decomposition-based(Lazy) outperforms Totalizer(Shuffle) on most of the instances in the group B (green), C (blue), and D (red).

We will now analyze the effect of tree-width on the performance. As we discussed in Section 8.2, when using the unary representations, the tree-width is not preserved even if we use the decomposition-based reduction; however, the difficulty might be preserved. Moreover, by updating the upper-bound k, the length of each unary representation becomes smaller, and therefore the tree-width also becomes smaller. Thus, instead of investigating the tree-width of the actual CNFs obtained by the reduction, we compute tree-width of CNFs obtained when setting the upper-bound k to 1. In this setting, the decomposition-based reduction can preserve tree-width.

Figure 8.5 shows the tree-width of the CNFs obtained by Totalizer, where a point with a coordinate (x, y) means that there is an instance for which the original tree-width is x and the tree-width of the reduced CNF is y. Here, the shown values are not exact but are heuristically computed by the min-degree

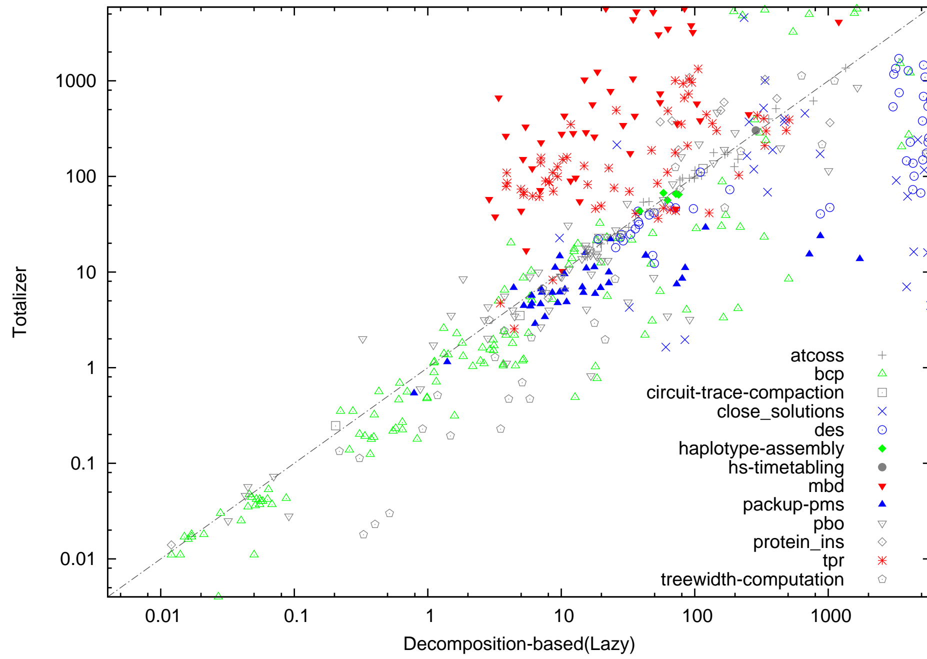

Figure 8.3: Running times of Decomposition-based(Lazy) and Totalizer

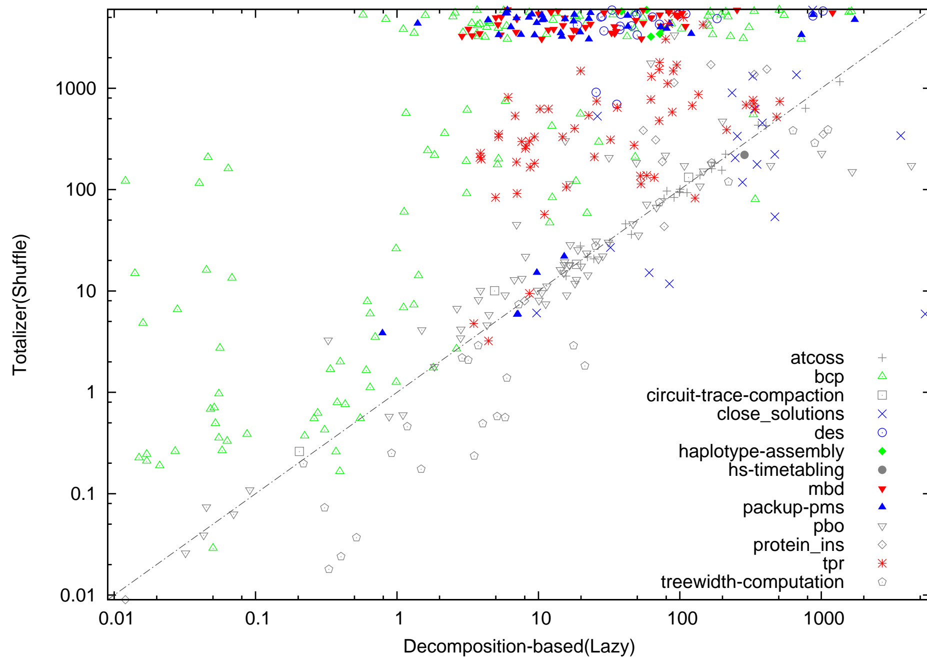

Figure 8.4: Running times of Decomposition-based(Lazy) and Totalizer(Shuffle)

heuristic with the star-based representation, which is the reason why there are several instances whose tree-width decreased through the reduction. Since we are using the heuristic method for computing a tree-width, the decomposition-based reduction might increase the heuristically computed tree-width. For a fair comparison, in Figure 8.6, we show the tree-width of the CNFs obtained by Decomposition-based(Lazy).

From the figures, we can confirm that on the groups A (gray) B (green), on which Totalizer and Decomposition-based(Lazy) showed similar performance, the increase of the tree-width by Totalizer is relatively smaller than the one on the

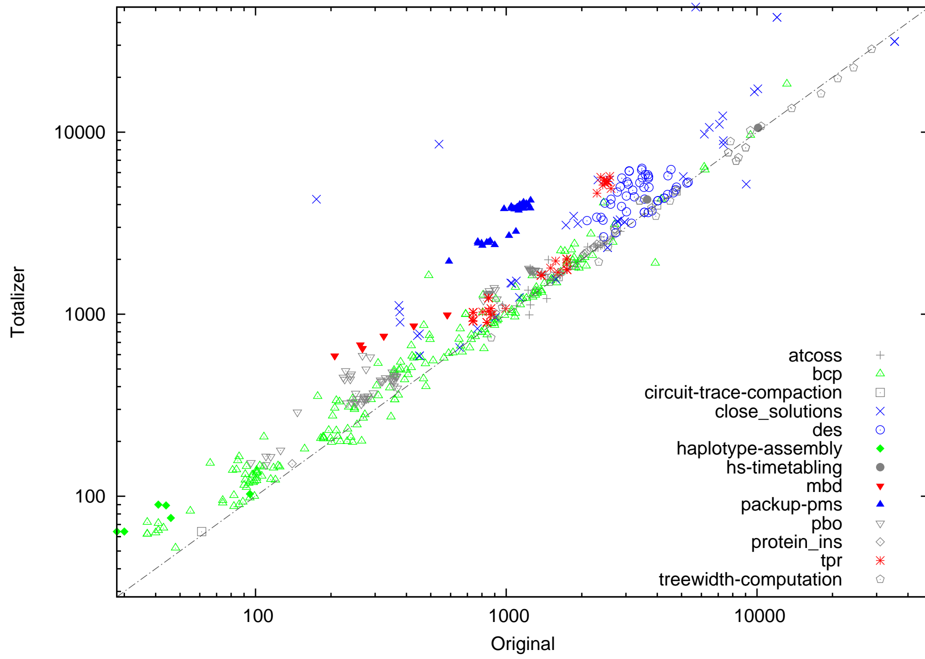

Figure 8.5: Tree-width of the CNFs obtained by Totalizer

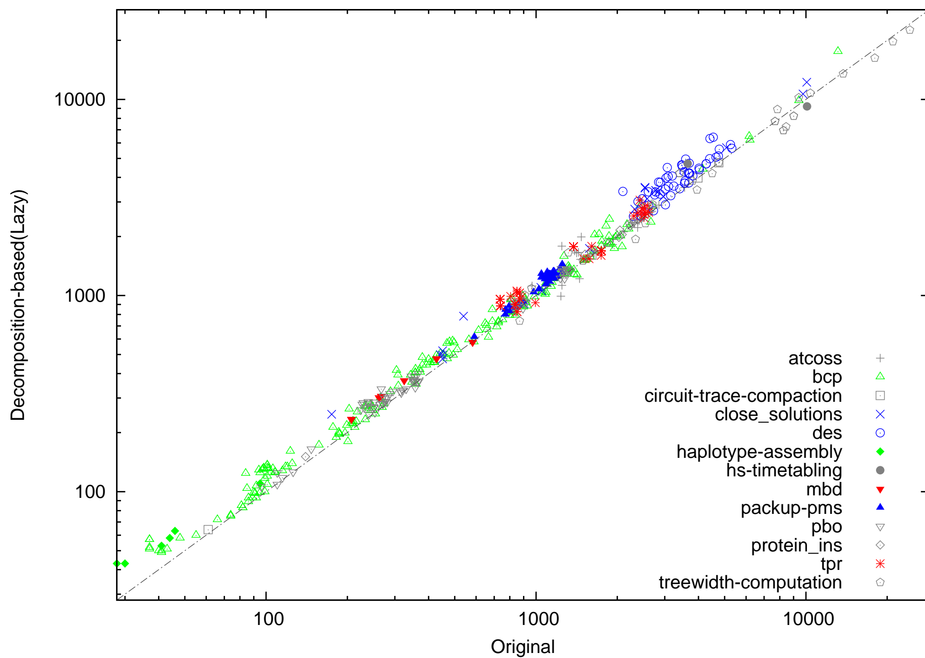

Figure 8.6: Tree-width of the CNFs obtained by Decomposition-based(Lazy)

other groups C (blue) and D (red).

Figure 8.7 shows the tree-width of the CNFs obtained by Totalizer(Shuffle). As we expected, the tree-width increases for most of the instances in the groups B (green), C (blue), and D (red), and moreover the amount of the increase is much larger than the one by Totalizer. The dataset close_solution is the only exception on which the tree-width strongly increases but Totalizer(Shuffle) is competitive to Decomposition-based(Lazy). Since instances in the close_solution dataset contain a huge number of soft clauses (see Figure 8.2), the height of the addition tree (in the proof of the Lemma 8.1), which is always $O(\log m)$ when

using Totalizer, might become too large when using Decomposition-based(Lazy), which may cause a delay of constraint propagation. We predict that this is the reason why Decomposition-based(Lazy) could not outperform Totalizer(Shuffle) on this dataset.

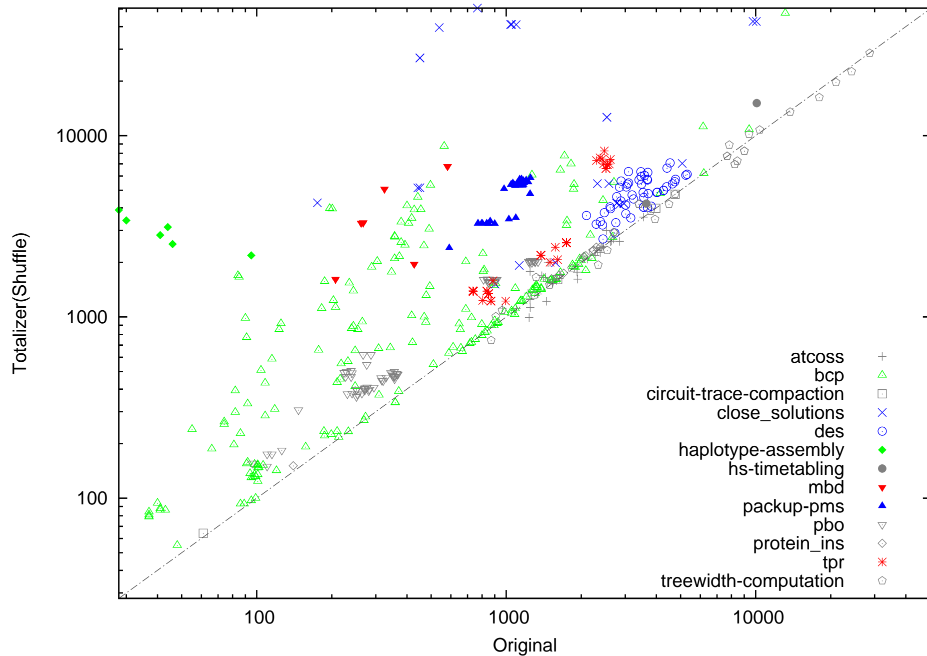

Figure 8.7: Tree-width of the CNFs obtained by Totalizer(Shuffle)

The Power of Lazy Addition

Finally, we analyze the effect of the lazy addition by comparing the performance of Decomposition-based and Decomposition-based(Lazy). Figure 8.8 shows the comparison of the running times of Decomposition-based(Lazy) and Decomposition-based, and Figures 8.9 and 8.10 show the comparison of the number of introduced variables and clauses. Although Decomposition-based introduces much more variables and clauses than Decomposition-based(Lazy) does, its performance is not so bad. This would be because, as we discussed in Section 8.2, a CNF obtained by Decomposition-based has a smaller (virtual) tree-width than a CNF obtained by Decomposition-based(Lazy) has.

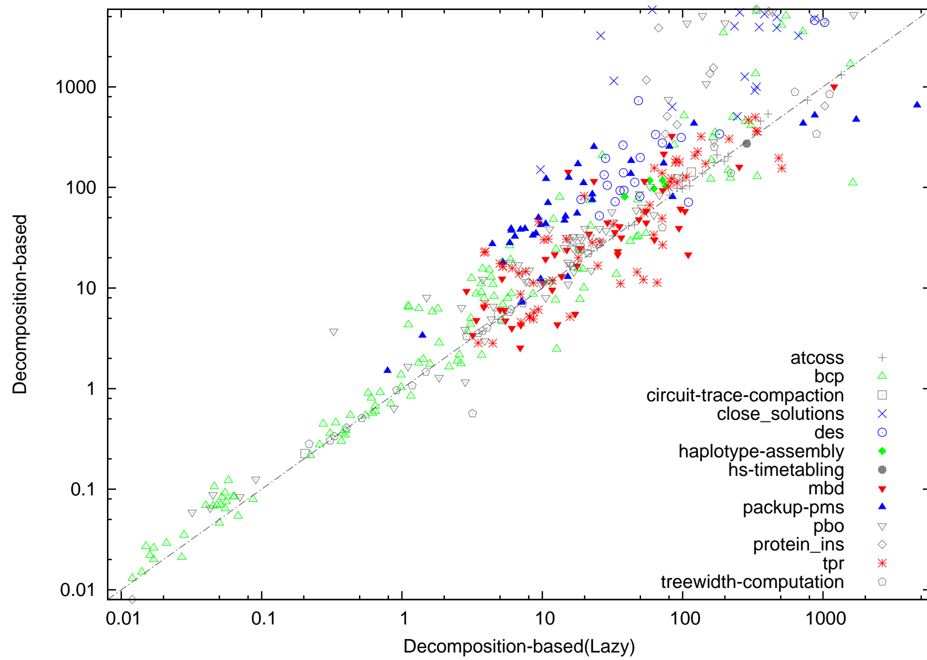

Figure 8.8: Running times of Decomposition-based(Lazy) and Decomposition-based

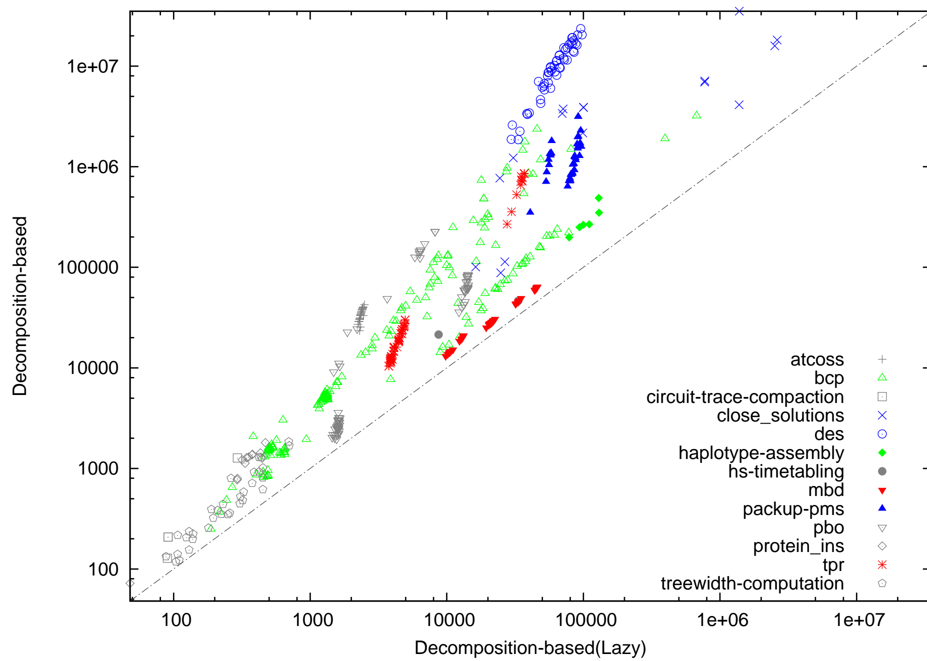

Figure 8.9: The number of introduced variables by Decomposition-based(Lazy) and Decomposition-based

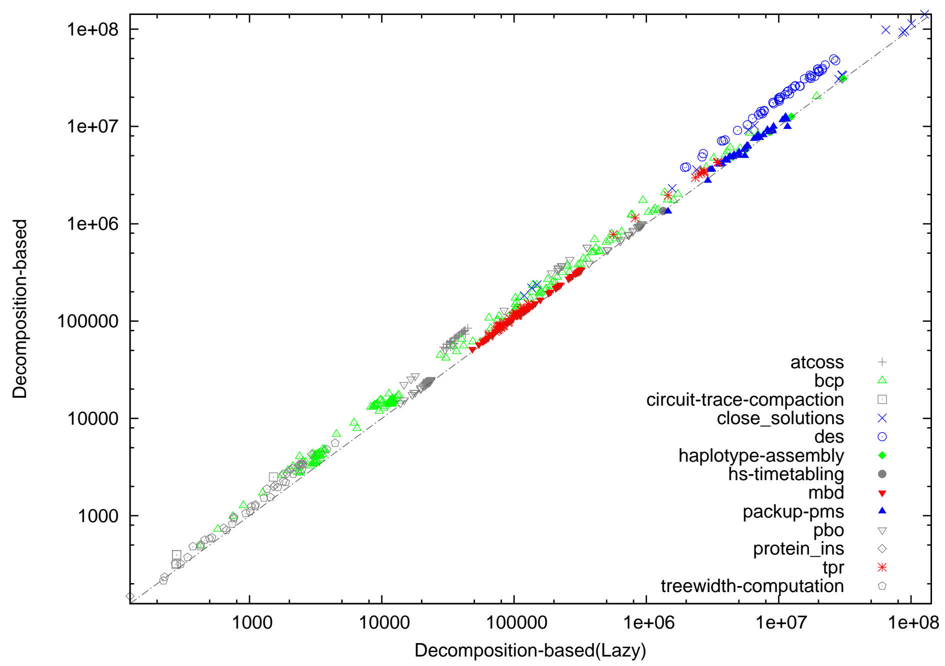

Figure 8.10: The number of introduced clauses by Decomposition-based(Lazy) and Decomposition-based

Chapter 9

Conclusions

In this thesis, we aimed to close the gap between theory and practice of exact algorithms for NP-hard problems from two directions.

From Practice to Theory

In Chapters 3 and 5, we showed that the practical branch-and-bound methods can be used to obtain theoretically fast FPT algorithms in terms of both $f(k)$ and $n^{O(1)}$. By introducing the discrete relaxations, we widened the range of applications of FPT branch-and-bound methods and obtained an improved FPT algorithms for Unique Label Cover. In the joint version [59], the results are further generalized to cover Group Feedback Vertex Set which includes Subset Feedback Vertex Set as a special case. However, compared to the wide range of practical applications of branch-and-bound methods, the range of applications to FPT algorithms is still narrow. Applying the methods to much wider range of problems is remained as future work. Especially, we are interested in directed problems, such as Directed Feedback Vertex Set and Directed Multiway Cut.

In Chapter 4, we investigated two lower bounds, clique cover and cycle cover, and showed that Vertex Cover is still NP-hard even if they provide the tight lower bound. Thus, we could not theoretically explain the power of these lower bounds. In a very recent paper, Garg and Philip [45] gave a stronger lower bound above which Vertex Cover is still FPT. Can we explain this result in the framework of discrete relaxations?

As for linear-time FPT algorithms, by generalizing the algorithm developed in Chapter 3, we obtained the linear-time FPT algorithm for Unique Label Cover in Chapter 5. However, we could not generalize it to cover vertex-deletion problems and Group Feedback Vertex Set. In a follow-up work by Lokshtanov, Ramanujan, and Saurabh [72], a linear-time FPT algorithm for Subset Feedback Vertex Set was obtained via very different approach. However, linear-time FPT algorithms for Group Feedback Vertex Set and vertex-deletion Unique Label Cover are still remained open.

In Chapter 6, we developed a new reduction technique, called decomposition-based reductions, and showed that reductions can preserve various structures, which theoretically explains why reductions are useful in practice; real-world instances admit some nice structures and such structures can be preserved through reductions; thus, even if the reduced instances becomes much larger than the original instances, by exploiting the structures, problems can be solved as efficiently as when they are directly solved without reductions. Since reductions used in practice are not decomposition-based reductions, they might break such nice structures. However, as we saw in Chapter 8, they often unintentionally

preserve structures. This would be the reason why SAT solvers can efficiently solve very huge instances reduced from real-world problems. Our new technique was applied to prove robust exponential-time hardness of problems parameterized by path-width in Chapter 7. Can we show a similar robust hardness for other problems?

From Theory to Practice

In Chapter 4, we investigated the practical impact of techniques developed in theoretical research on branching algorithms. Our experimental results indicated that, as well as theoretical importance, development of these techniques indeed leads to empirical efficiency. In a follow-up work by Lamm, Sanders, Schulz, Strash, and Werneck [67], these techniques were combined with *local search* methods to compute approximate solutions, and the practical importance of these techniques was reconfirmed.

In Chapter 8, we investigated the power of decomposition-based reductions. Although the proposed reductions could not outperform the existing reduction that uses the input ordering, we found that the existing reduction (unintentionally) preserves tree-width or some unknown more important structures. When using the random ordering, the tree-width exploded and the performance got worse. Thus, we confirmed that preserving tree-width is important for practical performance. Applying decomposition-based reductions to unsatisfiability-based approach or other problems is remained as future work. Analyzing structures hidden in the datasets for which our decomposition-based reduction did not work well is also an important task. We believe that by revealing such structures and intentionally designing the reduction to preserve the structures, the performance will be improved.

References

[1] Akiba, T. and Iwata, Y.: Branch-and-Reduce Exponential/FPT Algorithms in Practice: A Case Study of Vertex Cover, *Proceedings of the Seventeenth Workshop on Algorithm Engineering and Experiments, ALENEX 2015, San Diego, CA, USA, January 5, 2015*, pp. 70–81 (2015).

[2] Akiba, T. and Iwata, Y.: Branch-and-reduce exponential/FPT algorithms in practice: A case study of vertex cover, *Theoretical Computer Science*, Vol. 609, Part 1, pp. 211–225 (2016).

[3] Allender, E., Chen, S., Lou, T., Papakonstantinou, P. A. and Tang, B.: Time-space tradeoffs for width-parameterized SAT: Algorithms and lower bounds, *Electronic Colloquium on Computational Complexity (ECCC)*, Vol. 19, p. 27 (2012).

[4] Ansótegui, C., Bonet, M. L. and Levy, J.: Solving (Weighted) Partial MaxSAT through Satisfiability Testing, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, pp. 427–440 (2009).

[5] Ansótegui, C., Didier, F. and Gabàs, J.: Exploiting the Structure of Unsatisfiable Cores in MaxSAT, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pp. 283–289 (2015).

[6] Ansótegui, C., Malitsky, Y. and Sellmann, M.: MaxSAT by Improved Instance-Specific Algorithm Configuration, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pp. 2594–2600 (2014).

[7] Arnborg, S.: Efficient Algorithms for Combinatorial Problems with Bounded Decomposability - A Survey, *BIT Numerical Mathematics*, Vol. 25, No. 1, pp. 2–23 (1985).

[8] Arnborg, S., Corneil, D. G. and Proskurowski, A.: Complexity of Finding Embeddings in a k-Tree, *SIAM Journal on Algebraic Discrete Methods*, Vol. 8, No. 2, pp. 277–284 (1987).

[9] Arora, S. and Barak, B.: *Computational Complexity - A Modern Approach*, Cambridge University Press (2009).

[10] Asín, R., Nieuwenhuis, R., Oliveras, A. and Rodríguez-Carbonell, E.: Cardinality Networks: a theoretical and empirical study, *Constraints*, Vol. 16, No. 2, pp. 195–221 (2011).

[11] Atserias, A., Fichte, J. K. and Thurley, M.: Clause-Learning Algorithms with Many Restarts and Bounded-Width Resolution, *J. Artif. Intell. Res. (JAIR)*, Vol. 40, pp. 353–373 (2011).

[12] Backurs, A. and Indyk, P.: Edit Distance Cannot Be Computed in Strongly Subquadratic Time (unless SETH is false), *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pp. 51–58 (2015).

[13] Bailleux, O. and Boufkhad, Y.: Efficient CNF Encoding of Boolean Cardinality Constraints, *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, pp. 108–122 (2003).

[14] Bellman, R.: Dynamic programming treatment of the travelling salesman problem, *Journal of the ACM*, Vol. 9, No. 1, pp. 61–63 (1962).

[15] Berre, D. L. and Parrain, A.: The Sat4j library, release 2.2, *JSAT*, Vol. 7, No. 2-3, pp. 59–6 (2010).

[16] Berry, A., Heggernes, P. and Simonet, G.: The Minimum Degree Heuristic and the Minimal Triangulation Process, *Graph-Theoretic Concepts in Computer Science, 29th International Workshop, WG 2003, Elspeet, The Netherlands, June 19-21, 2003, Revised Papers*, pp. 58–70 (2003).

[17] Björklund, A.: Determinant Sums for Undirected Hamiltonicity, *SIAM J. Comput.*, Vol. 43, No. 1, pp. 280–299 (2014).

[18] Björklund, A., Husfeldt, T., Kaski, P. and Koivisto, M.: Fourier meets möbius: fast subset convolution, *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007*, pp. 67–74 (2007).

[19] Bodlaender, H. L.: A Linear-Time Algorithm for Finding Tree-Decompositions of Small Treewidth, *SIAM J. Comput.*, Vol. 25, No. 6, pp. 1305–1317 (1996).

[20] Bodlaender, H. L., Drange, P. G., Dregi, M. S., Fomin, F. V., Lokshtanov, D. and Pilipczuk, M.: An O(c^k n) 5-Approximation Algorithm for Treewidth, *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pp. 499–508 (2013).

[21] Bodlaender, H. L., Fomin, F. V., Koster, A. M. C. A., Kratsch, D. and Thilikos, D. M.: A Note on Exact Algorithms for Vertex Ordering Problems on Graphs, *Theory Comput. Syst.*, Vol. 50, No. 3, pp. 420–432 (2012).

[22] Bodlaender, H. L., van Leeuwen, E. J., van Rooij, J. M. M. and Vatshelle, M.: Faster Algorithms on Branch and Clique Decompositions, *Mathematical Foundations of Computer Science 2010, 35th International Symposium, MFCS 2010, Brno, Czech Republic, August 23-27, 2010. Proceedings*, pp. 174–185 (2010).

[23] Boldi, P., Rosa, M., Santini, M. and Vigna, S.: Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks, *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*, pp. 587–596 (2011).

[24] Boldi, P. and Vigna, S.: The webgraph framework I: compression techniques, *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*, pp. 595–602 (2004).

[25] Bouchet, A.: Coverings and Delta-Coverings, *Integer Programming and Combinatorial Optimization, 4th International IPCO Conference, Copenhagen, Denmark, May 29-31, 1995, Proceedings*, pp. 228–243 (1995).

[26] Bourgeois, N., Escoffier, B., Paschos, V. T. and van Rooij, J. M. M.: Fast Algorithms for max independent set, *Algorithmica*, Vol. 62, No. 1-2, pp. 382–415 (2012).

[27] Chen, J., Kanj, I. A. and Xia, G.: Improved upper bounds for vertex cover, *Theoretical Computer Science*, Vol. 411, No. 40-42, pp. 3736–3756 (2010).

[28] Cheng, J., Ke, Y., Chu, S. and Cheng, C.: Efficient processing of distance queries in large graphs: a vertex cover approach, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pp. 457–468 (2012).

[29] Chitnis, R. H., Cygan, M., Hajiaghayi, M., Pilipczuk, M. and Pilipczuk, M.: Designing FPT Algorithms for Cut Problems Using Randomized Contractions, *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012*, pp. 460–469 (2012).

[30] Corneil, D. G. and Rotics, U.: On the Relationship Between Clique-Width and Treewidth, *SIAM J. Comput.*, Vol. 34, No. 4, pp. 825–847 (2005).

[31] Courcelle, B., Makowsky, J. A. and Rotics, U.: Linear time solvable optimization problems on graphs of bounded clique-width, *Theor. Comput. Syst.*, Vol. 33, No. 2, pp. 125–150 (2000).

[32] Cygan, M., Dell, H., Lokshtanov, D., Marx, D., Nederlof, J., Okamoto, Y., Paturi, R., Saurabh, S. and Wahlström, M.: On Problems as Hard as CNF-SAT, *Proceedings of the 27th Conference on Computational Complexity, CCC 2012, Porto, Portugal, June 26-29, 2012*, pp. 74–84 (2012).

[33] Cygan, M., Nederlof, J., Pilipczuk, M., Pilipczuk, M., van Rooij, J. M. M. and Wojtaszczyk, J. O.: Solving Connectivity Problems Parameterized by Treewidth in Single Exponential Time, *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*, pp. 150–159 (2011).

[34] Cygan, M., Pilipczuk, M., Pilipczuk, M. and Wojtaszczyk, J. O.: On multiway cut parameterized above lower bounds, *TOCT*, Vol. 5, No. 1, p. 3 (2013).

[35] Davis, M., Logemann, G. and Loveland, D. W.: A machine program for theorem-proving, *Commun. ACM*, Vol. 5, No. 7, pp. 394–397 (1962).

[36] Dorn, F.: Dynamic Programming and Fast Matrix Multiplication, *Algorithms - ESA 2006, 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006, Proceedings*, pp. 280–291 (2006).

[37] Feige, U., Hajiaghayi, M. and Lee, J. R.: Improved Approximation Algorithms for Minimum Weight Vertex Separators, *SIAM J. Comput.*, Vol. 38, No. 2, pp. 629–657 (2008).

[38] Fiorini, S., Hardy, N., Reed, B. A. and Vetta, A.: Planar graph bipartization in linear time, *Discrete Applied Mathematics*, Vol. 156, No. 7, pp. 1175–1180 (2008).

[39] Flum, J. and Grohe, M.: Describing parameterized complexity classes, *Inf. Comput.*, Vol. 187, No. 2, pp. 291–319 (2003).

[40] Fomin, F. V., Grandoni, F. and Kratsch, D.: A Measure & Conquer Approach for the Analysis of Exact Algorithms, *J. ACM*, Vol. 56, No. 5, pp. 25:1–25:32 (2009).

[41] Fu, Z. and Malik, S.: On Solving the Partial MAX-SAT Problem, *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, pp. 252–265 (2006).

[42] Fujishige, S. and Iwata, S.: Bisubmodular Function Minimization, *SIAM J. Discrete Math.*, Vol. 19, No. 4, pp. 1065–1073 (2005).

[43] Funke, S., Nusser, A. and Storandt, S.: On k-Path Covers and their Applications, *PVLDB*, Vol. 7, No. 10, pp. 893–902 (2014).

[44] Garg, N., Vazirani, V. V. and Yannakakis, M.: Multiway Cuts in Directed and Node Weighted Graphs, *Automata, Languages and Programming, 21st International Colloquium, ICALP94, Jerusalem, Israel, July 11-14, 1994, Proceedings*, pp. 487–498 (1994).

[45] Garg, S. and Philip, G.: Raising The Bar For Vertex Cover: Fixed-parameter Tractability Above A Higher Guarantee, *CoRR*, Vol. abs/1509.03990 (2015).

[46] Guillemot, S.: FPT algorithms for path-transversal and cycle-transversal problems, *Discrete Optimization*, Vol. 8, No. 1, pp. 61–71 (2011).

[47] Held, M. and Karp, R. M.: A dynamic programming approach to sequencing problems, *Journal of the SIAM*, Vol. 10, pp. 196–210 (1962).

[48] Heras, F., Morgado, A. and Marques-Silva, J.: Core-Guided Binary Search Algorithms for Maximum Satisfiability, *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011* (2011).

[49] Hertli, T.: 3-SAT Faster and Simpler - Unique-SAT Bounds for PPSZ Hold in General, *SIAM J. Comput.*, Vol. 43, No. 2, pp. 718–729 (2014).

[50] Hopcroft, J. E. and Karp, R. M.: An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs, *SIAM J. Comput.*, Vol. 2, No. 4, pp. 225–231 (1973).

[51] Huber, A. and Kolmogorov, V.: Towards Minimizing k-Submodular Functions, *Combinatorial Optimization - Second International Symposium, ISCO 2012, Athens, Greece, April 19-21, 2012, Revised Selected Papers*, pp. 451–462 (2012).

[52] Hüffner, F.: Algorithm Engineering for Optimal Graph Bipartization, *Journal of Graph Algorithms and Applications*, Vol. 13, No. 2, pp. 77–98 (2009).

[53] Impagliazzo, R. and Paturi, R.: On the Complexity of k-SAT, *J. Comput. System Sci.*, Vol. 62, No. 2, pp. 367–375 (2001).

[54] Impagliazzo, R., Paturi, R. and Zane, F.: Which Problems Have Strongly Exponential Complexity?, *J. Comput. Syst. Sci.*, Vol. 63, No. 4, pp. 512–530 (2001).

[55] Iwata, S., Fleischer, L. and Fujishige, S.: A combinatorial strongly polynomial algorithm for minimizing submodular functions, *J. ACM*, Vol. 48, No. 4, pp. 761–777 (2001).

[56] Iwata, S. and Nagano, K.: Submodular Function Minimization under Covering Constraints, *50th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2009, October 25-27, 2009, Atlanta, Georgia, USA*, pp. 671–680 (2009).

[57] Iwata, Y.: A Faster Algorithm for Dominating Set Analyzed by the Potential Method, *Parameterized and Exact Computation - 6th International Symposium, IPEC 2011, Saarbrücken, Germany, September 6-8, 2011. Revised Selected Papers*, pp. 41–54 (2011).

[58] Iwata, Y., Oka, K. and Yoshida, Y.: Linear-Time FPT Algorithms via Network Flow, *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pp. 1749–1761 (2014).

[59] Iwata, Y., Wahlström, M. and Yoshida, Y.: Half-integrality, LP-branching and FPT Algorithms, *CoRR*, Vol. abs/1310.2841 (2014).

[60] Iwata, Y. and Yoshida, Y.: On the Equivalence among Problems of Bounded Width, *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, pp. 754–765 (2015).

[61] Johnson, D. J. and Trick, M. A.(eds.): *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11-13, 1993*, American Mathematical Society (1996).

[62] Kawarabayashi, K. and Reed, B. A.: An (almost) Linear Time Algorithm for Odd Cyles Transversal, *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pp. 365–378 (2010).

[63] Khot, S.: On the power of unique 2-prover 1-round games, *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada*, pp. 767–775 (2002).

[64] Kneis, J., Langer, A. and Rossmanith, P.: A Fine-grained Analysis of a Simple Independent Set Algorithm, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2009, December 15-17, 2009, IIT Kanpur, India*, pp. 287–298 (2009).

[65] Kolmogorov, V.: The power of linear programming for valued CSPs: a constructive characterization, *CoRR*, Vol. abs/1207.7213 (2012).

[66] Koshimura, M., Zhang, T., Fujita, H. and Hasegawa, R.: QMaxSAT: A Partial Max-SAT Solver, *JSAT*, Vol. 8, No. 1/2, pp. 95–100 (2012).

[67] Lamm, S., Sanders, P., Schulz, C., Strash, D. and Werneck, R. F.: Finding Near-Optimal Independent Sets at Scale, *CoRR*, Vol. abs/1509.00764 (2015).

[68] Le Gall, F.: Powers of tensors and fast matrix multiplication, *International Symposium on Symbolic and Algebraic Computation, ISSAC '14, Kobe, Japan, July 23-25, 2014*, pp. 296–303 (2014).

[69] Lipton, R. J. and Tarjan, R. E.: A Separator Theorem for Planar Graphs, *SIAM Journal on Applied Mathematics*, Vol. 36, No. 2, pp. 177–189 (1979).

[70] Lokshtanov, D., Marx, D. and Saurabh, S.: Known Algorithms on Graphs on Bounded Treewidth are Probably Optimal, *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San Francisco, California, USA, January 23-25, 2011*, pp. 777–789 (2011).

[71] Lokshtanov, D., Narayanaswamy, N. S., Raman, V., Ramanujan, M. S. and Saurabh, S.: Faster Parameterized Algorithms Using Linear Programming, *ACM Transactions on Algorithms*, Vol. 11, No. 2, pp. 15:1–15:31 (2014).

[72] Lokshtanov, D., Ramanujan, M. S. and Saurabh, S.: Linear Time Parameterized Algorithms for Subset Feedback Vertex Set, *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*, pp. 935–946 (2015).

[73] Lovász, L.: *Mathematical Programming The State of the Art: Bonn 1982*, chapter Submodular functions and convexity, pp. 235–257, Springer Berlin Heidelberg (1983).

[74] Maehara, T., Akiba, T., Iwata, Y. and Kawarabayashi, K.: Computing Personalized PageRank Quickly by Exploiting Graph Structures, *PVLDB*, Vol. 7, No. 12, pp. 1023–1034 (2014).

[75] Marques-Silva, J. and Manquinho, V. M.: Towards More Effective Unsatisfiability-Based Maximum Satisfiability Algorithms, *Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*, pp. 225–230 (2008).

[76] Marques-Silva, J. and Planes, J.: On Using Unsatisfiability for Solving Maximum Satisfiability, *CoRR*, Vol. abs/0712.1097 (2007).

[77] Marques-Silva, J. and Planes, J.: Algorithms for Maximum Satisfiability using Unsatisfiable Cores, *Design, Automation and Test in Europe, DATE 2008, Munich, Germany, March 10-14, 2008*, pp. 408–413 (2008).

[78] Martins, R., Manquinho, V. M. and Lynce, I.: Open-WBO: A Modular MaxSAT Solver,, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, pp. 438–445 (2014).

[79] Monien, B. and Speckenmeyer, E.: Solving satisfiability in less than 2^n steps, *Discrete Applied Mathematics*, Vol. 10, No. 3, pp. 287–295 (1985).

[80] Morgado, A., Heras, F. and Marques-Silva, J.: Improvements to Core-Guided Binary Search for MaxSAT, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, pp. 284–297 (2012).

[81] Nemhauser, G. and Trotter, L.: Vertex packing: structural properties and algorithms, *Mathematical Programming*, Vol. 8, pp. 232–248 (1975).

[82] Niedermeier, R.: *Invitation to Fixed-Parameter Algorithms*, Oxford University Press (2006).

[83] Ogawa, T., Liu, Y., Hasegawa, R., Koshimura, M. and Fujita, H.: Modulo Based CNF Encoding of Cardinality Constraints and Its Application to MaxSAT Solvers, *2013 IEEE 25th International Conference on Tools with Artificial Intelligence, Herndon, VA, USA, November 4-6, 2013*, pp. 9–17 (2013).

[84] Patrascu, M. and Williams, R.: On the Possibility of Faster SAT Algorithms, *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pp. 1065–1075 (2010).

[85] Picard, J.-C. and Queyranne, M.: On the structure of all minimum cuts in a network and applications, *Combinatorial Optimization II*, Mathematical Programming Studies, Vol. 13, Springer Berlin Heidelberg, pp. 8–16 (1980).

[86] Raman, V., Ramanujan, M. S. and Saurabh, S.: Paths, Flowers and Vertex Cover, *Algorithms - ESA 2011 - 19th Annual European Symposium, Saarbrücken, Germany, September 5-9, 2011. Proceedings*, pp. 382–393 (2011).

[87] Ramanujan, M. S. and Saurabh, S.: Linear Time Parameterized Algorithms via Skew-Symmetric Multicuts, *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pp. 1739–1748 (2014).

[88] Razgon, I.: Computing Minimum Directed Feedback Vertex Set in $\mathrm{O}(1.9977^{\mathrm{n}})$, *Theoretical Computer Science, 10th Italian Conference, ICTCS 2007, Rome, Italy, October 3-5, 2007, Proceedings*, pp. 70–81 (2007).

[89] Razgon, I.: Faster computation of maximum independent set and parameterized vertex cover for graphs with maximum degree 3, *J. Discrete Algorithms*, Vol. 7, No. 2, pp. 191–212 (2009).

[90] Razgon, I. and O'Sullivan, B.: Almost 2-SAT is fixed-parameter tractable, *Journal of computer and system sciences*, Vol. 75, No. 8, pp. 435–450 (2009).

[91] Reed, B., Smith, K. and Vetta, A.: Finding odd cycle transversals, *Operations Research Letters*, Vol. 32, No. 4, pp. 299–301 (2004).

[92] Robertson, N. and Seymour, P. D.: Graph Minors. II. Algorithmic Aspects of Tree-Width, *J. Algorithms*, Vol. 7, No. 3, pp. 309–322 (1986).

[93] Robertson, N. and Seymour, P. D.: Graph minors. X. Obstructions to tree-decomposition, *J. Comb. Theory B*, Vol. 52, No. 2, pp. 153–190 (1991).

[94] Schrijver, A.: A Combinatorial Algorithm Minimizing Submodular Functions in Strongly Polynomial Time, *J. Comb. Theory, Ser. B*, Vol. 80, No. 2, pp. 346–355 (2000).

[95] Silva, J. P. M. and Sakallah, K. A.: GRASP: A Search Algorithm for Propositional Satisfiability, *IEEE Trans. Computers*, Vol. 48, No. 5, pp. 506–521 (1999).

[96] Tarjan, R. E.: Depth-First Search and Linear Graph Algorithms, *SIAM J. Comput.*, Vol. 1, No. 2, pp. 146–160 (1972).

[97] Thapper, J. and Zivny, S.: The Power of Linear Programming for Valued CSPs, *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012*, pp. 669–678 (2012).

[98] Thapper, J. and Zivny, S.: The complexity of finite-valued CSPs, *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pp. 695–704 (2013).

[99] Tomita, E., Sutani, Y., Higashi, T., Takahashi, S. and Wakatsuki, M.: A Simple and Faster Branch-and-Bound Algorithm for Finding a Maximum Clique, *WALCOM: Algorithms and Computation, 4th International Workshop, WALCOM 2010, Dhaka, Bangladesh, February 10-12, 2010. Proceedings*, pp. 191–203 (2010).

[100] van Rooij, J. M. M., Bodlaender, H. L. and Rossmanith, P.: Dynamic Programming on Tree Decompositions Using Generalised Fast Subset Convolution, *Algorithms - ESA 2009, 17th Annual European Symposium, Copenhagen, Denmark, September 7-9, 2009. Proceedings*, pp. 566–577 (2009).

[101] Wahlström, M.: Half-integrality, LP-branching and FPT Algorithms, *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pp. 1762–1781 (2014).

[102] Williams, R.: A new algorithm for optimal 2-constraint satisfaction and its implications, *Theor. Comput. Sci.*, Vol. 348, No. 2-3, pp. 357–365 (2005).

[103] Woeginger, G. J.: Space and Time Complexity of Exact Algorithms: Some Open Problems (Invited Talk), *Parameterized and Exact Computation, First International Workshop, IWPEC 2004, Bergen, Norway, September 14-17, 2004, Proceedings*, pp. 281–290 (2004).

[104] Woeginger, G. J.: Open problems around exact algorithms, *Discrete Appl. Math*, Vol. 156, No. 3, pp. 397–405 (2008).

[105] Xiao, M. and Nagamochi, H.: Confining sets and avoiding bottleneck cases: A simple maximum independent set algorithm in degree-3 graphs, *Theoretical Computer Science*, Vol. 469, No. 0, pp. 92 – 104 (2013).

[106] Xiao, M. and Nagamochi, H.: Exact Algorithms for Maximum Independent Set, *Algorithms and Computation - 24th International Symposium, ISAAC*

2013, Hong Kong, China, December 16-18, 2013, Proceedings, pp. 328–338 (2013).