MEASURE-DRIVEN ALGORITHM DESIGN AND ANALYSIS: A NEW APPROACH FOR SOLVING NP-HARD PROBLEMS

A Dissertation

by

YANG LIU

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

August 2009

Major Subject: Computer Science

MEASURE-DRIVEN ALGORITHM DESIGN AND ANALYSIS:
A NEW APPROACH FOR SOLVING NP-HARD PROBLEMS

A Dissertation

by

YANG LIU

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	Jianer Chen
Committee Members,	Donald K. Friesen
	Weiping Shi
	Jennifer L. Welch
Head of Department,	Valerie E. Taylor

August 2009

Major Subject: Computer Science

ABSTRACT

Measure-Driven Algorithm Design and Analysis:
A New Approach for Solving NP-hard Problems. (August 2009)
Yang Liu, B.S., Zhejiang University;
M.S., Rose-Hulman Institute of Technology
Chair of Advisory Committee: Dr. Jianer Chen

NP-hard problems have numerous applications in various fields such as networks, computer systems, circuit design, etc. However, no efficient algorithms have been found for NP-hard problems. It has been commonly believed that no efficient algorithms for NP-hard problems exist, i.e., that P≠NP. Recently, it has been observed that there are parameters much smaller than input sizes in many instances of NP-hard problems in the real world. In the last twenty years, researchers have been interested in developing efficient algorithms, i.e., fixed-parameter tractable algorithms, for those instances with small parameters. Fixed-parameter tractable algorithms can practically find exact solutions to problem instances with small parameters, though those problems are considered intractable in traditional computational theory.

In this dissertation, we propose a new approach of algorithm design and analysis: discovering better measures for problems. In particular we use two measures instead of the traditional single measure—input size to design algorithms and analyze their time complexity. For several classical NP-hard problems, we present improved algorithms designed and analyzed with this new approach,

First we show that the new approach is extremely powerful for designing fixed-parameter tractable algorithms by presenting improved fixed-parameter tractable algorithms for the 3D-MATCHING and 3D-PACKING problems, the MULTIWAY CUT problem, the FEEDBACK VERTEX SET problems on both directed and undirected graph and the MAX-LEAF problems on both directed and undirected graphs. Most of our algorithms are practical for problem instances with small parameters.

Moreover, we show that this new approach is also good for designing exact algorithms (with no parameters) for NP-hard problems by presenting an improved exact algorithm for the well-known SATISFIABILITY problem.

Our results demonstrate the power of this new approach to algorithm design and analysis for NP-hard problems. In the end, we discuss possible future directions on this new approach and other approaches to algorithm design and analysis.

To my family

ACKNOWLEDGMENTS

I would like to give my greatest gratitude to my advisor, Dr. Jianer Chen. He aroused my interests in algorithm design and analysis and encouraged me to go further and do better in research. His support and confidence helped me to succeed after failure. He gave numerous examples himself on how to express ideas clearly in both writing and talking. I hope that I will be an advisor to my students as Dr. Chen has been to me in my future academic career.

Dr. Donald K. Friesen has always been there to listen and to give advice. He was my mentor when I was in the Graduate Teaching Academy program. He has given advice on research and teaching. His opinion on research encouraged me to go further in research. He carefully corrected my dissertation, and gave suggestions on the writing. I am grateful to Dr. Friesen for his help.

Dr. Jennifer L. Welch has helped me whenever I asked for help. She suggested new directions for my research, and showed me the interesting area of distributed algorithms. She also helped me to correct this dissertation. I am thankful to Dr. Welch for her help.

Dr. Weiping Shi discussed problems with me even when he was very busy. He opened the door for me to the fascinating area of CAD algorithms and gave suggestions on my research to make my research more interesting. I am grateful to Dr. Shi for his help.

I would like to thank Songjian Lu, Fenghui Zhang, Qilong Feng, Jie Meng, Jiahao Fan and Yixin Cao for their valuable discussions. In particular, Songjian Lu has collaborated with me successfully since I started my Ph.D. study. Our collaboration has been very productive during my Ph.D. study.

I would like to thank the Department of Computer Science at Texas A&M University for financial support during my Ph.D. study. Their support has helped me to be able to concentrate on my studies.

Finally, I would like to thank my family, especially my wife Ping. They have eased my worries and encouraged me to continue my education with both emotional and financial support. Without their love and support, this work would not have been possible.

TABLE OF CONTENTS

LIST OF TABLES

TABLE Page

LIST OF FIGURES

CHAPTER I

INTRODUCTION

There are abundant applications of NP-hard problems. These applications have attracted intensive studies of NP-hard problems. There are many methods to tackle NP-hard problems: exact algorithms, approximation algorithms, probabilistic algorithms, heuristics, and parameterized algorithms. In this dissertation, we are interested in design and analysis of exact algorithms and parameterized algorithms for NP-hard problems. In particular, we are interested in fpt-algorithms which are a special class of parameterized algorithms. The rest of this chapter is organized as follows: in section A, we discuss why we study exact algorithms. In section B, we discuss why we study fixed-parameter tractable algorithms. In section C, we introduce the branch-and-search tree technique which will be widely applied later in this dissertation. In section D, we give an outline of this dissertation.

A. Why Study Exact Algorithms

First, exact algorithms can find optimal solutions, which minimize the cost or maximize the benefits in applications. This is the major reason why we prefer exact algorithms to approximation algorithms and heuristics. Moreover, only exact algorithms can find meaningful and correct answers for decision problems. Solutions from approximation algorithms make no sense at all, and heuristics can be wrong for some inputs. For example, the SATISFIABILITY problem should be answered with either ”satisfiable” or ”not satisfiable”. Approximation algorithms can not help at all. Therefore, we need to study exact algorithms for NP-hard problems.

The journal model is *IEEE Transactions on Automatic Control.*

Second, exact algorithms also can be used as a subroutine by other heuristics. For example, optimal solutions to small instances are found and stored in a table. Then heuristics break a large instance into small instances, and use the solutions in the table to find a solution (not necessary optimal) for the large input instance. This technique helps to find better solutions [26] for the RECTILINEAR STEINER MINIMAL TREE problem. So studying exact algorithms can help to develop heuristics with better solutions.

Moreover, faster exact algorithms for NP-hard problem may increase significantly the size of instances which can be handled, since those algorithms are expected to be of exponential running time according to the widely accepted belief that P$\neq$NP. For example, if an exact algorithm of time $O(2^{n/2})$ is developed and the previous best algorithm is of time $O(2^n)$, then the size of instances which can be handled by the new exact algorithm is twice that of instances which can be handled by the previous best algorithm. On the other hand, the speedup of processors can not increase linearly the size of instances which can be handled as exact algorithms. With the amazing progress of technology, the processors are almost $9,000$ times faster in 2004 than the processors in 1978 [60]. But the size of instances which can be handled by the $9,000$ times faster processors are only $\log_2 9,000$ larger than that of instances which can be handled by the old processors, for an $O(2^n)$ algorithm. The increase in instance size is less than 13. Therefore, it is still desirable to study exact algorithms, despite the incredible speedup of processors.

Current progress in exact algorithms for NP-hard problems appeals to further studies of exact algorithms. A notable example is the RECTILINEAR STEINER TREES problem. There is an exact algorithm which finds an optimal Rectilinear Steiner Tree in 38 CPU hours for instances of size up to 1000 [100]. Another exciting example is the TRAVELING SALESMAN problem. This problem now can be solved for inputs of size $13,509$, though 48 workstations had run for around 10 years [4].

B. Why Study Fpt-algorithms

Though studying exact algorithms is necessary for NP-hard problems, it is notoriously difficult to develop fast algorithms such that we can solve instances with considerable sizes. However, it has been observed that there are parameters in many instances of NP-hard problems, which are small compared to their input sizes. Given such an instance, let n be its input size and k be its parameter. Then k may be much smaller than n. Moreover, the running time of algorithms for that instance is a function of both n and k. *Parameterized algorithms* are those algorithms whose running time is a function of n and k. In last decade, many researchers have studied parameterized algorithms extensively.

Not all parameterized algorithms are practical. For example, an algorithm of running time $O(n^k)$ is not practical even for $k = 10$ when n is moderately large. In particular, people are interested in efficient parameterized algorithms whose running time are $O(f(k)n^c)$ where (1) c is a constant independent of input size n and the parameter k, and (2) $f(k)$ is independent of n. Those efficient parameterized algorithms are *fixed parameter tractable* (fpt-) algorithms. A problem is *fixed parameter tractable* if there exists an fpt-algorithm for the problem.

For some instances of NP-hard problems in real world, there are natural parameters that are much smaller than the sizes of inputs. We give two examples.

The first example is the problem Type Checking in ML [38]. One of the tasks of a compiler is to check the compatibility of type declarations. This problem is exponential-time complete [59]. There is an fpt-algorithm of running time $O(2^k n)$ where n is the size of the program and k is the maximum nesting depth of the type declarations [77]. Normally k is not greater than 6. So this algorithm works well in practice.

Another example is a recent fpt-algorithm for the Individual Haplotyping problem [103]. The fpt-algorithm is of time complexity $O(nk_2 2^{k_2} + m \log m + mk_1)$ where m is the number of fragments (the number of 0/1/2 sequences), n is the number of SNP sites (the length of each 0/1/2 sequence), k_1 is the maximum number of SNP sites that a fragment covers (the maximum number of 0/1's in a sequences), and k_2 is the maximum number of the fragments covering an SNP site (the number of sequences which have 0/1 in a particular position in the sequence). The parameter k_1 is bounded by n. But normally k_1 is smaller than 10. The parameter k_2 is usually not more than 19 [63]. Thus this algorithm is practical. Moreover, it is more accurate in haplotype reconstruction than other known algorithms.

From these examples, we can see that fpt-algorithms are practical for instances with small parameters. So we need to study fpt-algorithms for NP-hard problems. We will investigate a new approach for designing fpt-algorithms in this dissertation.

C. Branch-and-Search Tree

There are many algorithmic approaches to design exact algorithms (without parameters) and fpt-algorithms. Algorithmic approaches for exact algorithms (without parameters) include dynamic programming, pruning the search tree, preprocessing the data and local search [101], measure and conquer [49, 50]. Algorithmic approaches for fpt-algorithms include kernelization [39], greedy localization [19, 42], iterative compression [90, 32], coloring coding [3], divide and conquer [25], divide-and-color [70], and branch-and-search tree [20, 73]. Among these approaches, branch-and-search tree is the most widely used approach, and can be applied to design both exact algorithms (without parameters) and fpt-algorithms.

Given an instance I of a problem, the branch-and-search tree approach can be applied to design a recursive algorithm $\mathcal{A}$ to solve I. If the instance I satisfies some conditions, $\mathcal{A}$ solves the problem for the instance I directly. Otherwise, $\mathcal{A}$ reduces the instance I into some smaller instances $I_1, \cdots, I_p$, calls $\mathcal{A}$ recursively on each of $I_1, \cdots, I_p$ to find solutions to $I_1, \cdots, I_p$, and then finds solutions to I from solutions to $I_1, \cdots, I_p$.

To study the time complexity of such an algorithm, we consider the algorithm $\mathcal{A}$ as a tree $\mathcal{T}$. Each node represents an instance. Let $instance(N)$ be the instance represented by node N. If N is the root of $\mathcal{T}$, then $instance(N)$ is the original instance I. If N is a leaf of $\mathcal{T}$, then $instance(N)$ is an instance that can be solved directly without recursive call of $\mathcal{A}$. A node N and its children $N_1, \cdots, N_q$ represent a call of $\mathcal{A}$ on $instance(N)$ during which $instance(N)$ is reduced to $instance(N_1), \cdots, instance(N_q)$.

Now we can distribute the running time of $\mathcal{A}$ on instance I in the tree $\mathcal{T}$ as follows: the time for a leaf L is the time to solve directly $instance(L)$. The time for an internal node N with children $N_1, \cdots, N_q$ is the time to reduce $instance(N)$ to $instance(N_1), \cdots, instance(N_q)$ plus the time to find solution to $instance(N)$ from solutions to $instance(N_1)$, $\cdots$, $instance(N_q)$. It is clear that the running time of $\mathcal{A}$ on I is the summation of the time for nodes in the tree $\mathcal{T}$.

Normally it is required that the time for each node be polynomial in the input size n of I, and the height of $\mathcal{T}$ be also polynomial in n. Let T_{path} be the summation of the time for all nodes in a path from the root to a leaf in $\mathcal{T}$. Then T_{path} is polynomial in n. Let T_{max} be the maximum among all possible T_{path}. The running time of $\mathcal{A}$ is then bounded by the product of T_{max} and the number of leaves in $\mathcal{T}$.

In general, it is easier to give an upper bound of the number of leaves in $\mathcal{T}$ than to calculate the precise number of leaves in $\mathcal{T}$. To bound the number of leaves in $\mathcal{T}$, we assign a measure to every node in $\mathcal{T}$. Let m be the measure of the root in $\mathcal{T}$, and $m_1, \cdots, m_q$ be the measures corresponding to the children of the root in $\mathcal{T}$. A *characteristic polynomial* for $m, m_1, \cdots, c_q$ is $\sum x^{d_i} = 1$ where $d_i = m_i - m$ for $i = 1, \cdots, q$. We also say that the root has a characteristic polynomial $\sum x^{d_i} = 1$. There is only one positive root for any characteristic polynomial. Let α be the root of a characteristic polynomial. It can be proved that $f(m) \leq \alpha^m$ when all internal nodes have the same characteristic polynomial. If there are multiple characteristic polynomials $\mathcal{P}_1, \cdots, \mathcal{P}_q$, let α_i be the positive root of the characteristic polynomial $\mathcal{P}_i$, and $\alpha = (\max\{\alpha_1, \cdots, \alpha_q\})$. Then we have $f(m) \leq \alpha^m$. More explanations and proofs of this approach can be found in [73].

From the arguments above, the total running time of algorithm $\mathcal{A}$ is bounded by $O(\alpha^m poly(n))$. For simplicity, we use notation $O^*(\alpha^m)$[1] to ignore the part of $poly(n)$ in $O(\alpha^m poly(n))$.

Now we give examples to illustrate the approach of branch-and-search tree. Consider the MINIMUM VERTEX COVER problem, which is NP-hard. Given a graph G, a *vertex cover* C of G is a subset of vertices such that every edge has at least one endpoint in C. A *minimum vertex cover* is a vertex cover of the minimum number of vertices among all vertex covers. The MINIMUM VERTEX COVER problem is to find a minimum vertex cover for the input graph G. We design an exact algorithm for the MINIMUM VERTEX COVER problem with the branch-and-search tree approach.

Normally, we choose the number of vertices n to be the measure. Let C be a minimum vertex cover. If there are no edges in the input graph G, then return ϕ as a minimum vertex cover. Otherwise, pick an edge xy and consider two cases: either

---

[1] Formally, $O^*(f)$ refers to $O(fn^{O(1)})$ where n is the input size.

x or y is in the minimum vertex cover C. If x is in C, let G_x be the remaining graph after deleting x and all incident edges of x. Then we only need to find a minimum vertex cover $C - x$ for G_x. Note that $|V[G_x]| = |V[G]| - 1$, i.e., the measure decreases by 1. If x is not in C, then y must be in C. Let G_y be the graph after deleting y. Then we only need to find a minimum vertex cover $C - y$ for G_y. The measure of G_y also decreases by 1. The characteristic polynomial for these measures is $x^{-1} + x^{-1} = 1$. Solve it and find its positive root $\alpha = 2$. So $f(n) \leq 2^n$. That is, the MINIMUM VERTEX COVER problem can be solved in time $O^*(2^n)$.

Next we show how to design fpt-algorithms with branch-and-search tree. The parameter for the VERTEX COVER problem is the size of a vertex cover to find. Let k be the size of a vertex cover to find. Formally, the k-VERTEX COVER problem is to: either find a vertex cover of k vertices if such a vertex cover exists, or report 'NO' if none exists. The same algorithm above can be applied to solve the k-VERTEX COVER problem. However, we should redefine C to be the vertex cover of k vertices to search, and calculate the measure changes accordingly: (1) for G_x, the measure decreases by 1 since we only need to find $C - x$ in G_x, and (2) for G_y, the measure decreases by 1 since we only need to find $C - y$ in G_y. The characteristic polynomial for these measures is still $x^{-1} + x^{-1} = 1$, whose positive root $\alpha = 2$. Therefore, the k-vertex cover problem can be solved in time $O^*(2^k)$.

It is difficult to design better algorithm of time $O^*(c^n)$ or $O^*(c^k)$ with small c. The best algorithm for the MINIMUM VERTEX COVER problem has running time of $O^*(1.1893^n)$ [91]. The best algorithm for the k-VERTEX COVER problem has running time of $O^*(1.2738^k)$ [21]. But these algorithms are much more complicated than the simple algorithm above.

Before our works [78, 23, 18, 24, 22], researchers had taken only one parameter when using the approach of branch-and-search tree. This common usage has made it difficult to design fast algorithms. For example, let us consider the k FEEDBACK VERTEX SET problem. This problem either finds a set S of k vertices such that every cycle contains at least one vertex in S, or reports no such set exists. To apply the traditional approach of branch search tree, we naturally pick a cycle C and have p choices to put a vertex in S, where p is the length of C. Then we have a characteristic polynomial $f(k) = pf(k-1)$, whose positive root is p. Thus any algorithm by this way has time complexity of $O^*(p^k)$. It is hard to bound the length of cycles by some constant in graphs. Thus it is difficult to design algorithms of time $O^*(c^k)$, where c is a constant not related to k, by this traditional approach. In this dissertation, we show that using two measures, instead of single measure, is much more powerful than the traditional approach of branch-and-search tree because more properties can be applied to design algorithms. With this new approach, we present an algorithm of time $O^*(5^k)$ for the k FEEDBACK VERTEX SET problem on undirected graphs in Chapter IV.

D. Outline of This Dissertation

In Chapters II to VI, we apply our new approach to design better fpt-algorithms for several problems: 3D-MATCHING and 3D-PACKING problems, MULTIWAY-CUT problem, FEEDBACK VERTEX SET problems on undirected graphs and directed graphs, and MAX LEAF problem. Detailed analyses are given. The results in these chapters illustrate the power of our new approach for fpt-algorithm design and analysis.

Then in Chapter VII, we illustrate how to design an improved algorithm for the well-known SATISFIABILITY problem using two measures. The result in this chapter demonstrates that our new approach also works well for exact algorithm design and analysis.

In Chapter VIII, we give a summary of our work and directions for future research.

CHAPTER II

3D-MATCHING AND 3-SET PACKING

In this chapter, we give improved randomized and deterministic fpt-algorithms for the 3-D MATCHING and 3-SET PACKING problems. Our randomized algorithm for the 3-D MATCHING problem has running time of $O^*(2.32^{3k})$, which improves the previous best randomized algorithm of running time $O^*(2.52^{3k})$. Our deterministic algorithm for the 3-D MATCHING is of running time $O^*(2.77^{3k})$, which improves the previous best deterministic algorithms of running time $O^*(12.8^{3k})$. Our deterministic algorithm for the 3-SET PACKING problems are of running time $O^*(4.61^{3k})$, which improves the previous best deterministic algorithms of running time $O^*(12.8^{3k})$.

The previous algorithms for the 3-D MATCHING and 3-SET PACKING problems focus on the parameter k as a measure, which makes it difficult to design faster algorithms. We still study these problems with the greedy localization approach [19]: start from a matching (packing) of size p and to find a matching (packing) of size k where $k > p$. However, instead of focusing on the parameter k only, we consider two measures: the number of colors used for coloring and the number of elements used in dynamic programming. With this new approach, we discover that only $p+3$ colors are needed, and the number of elements used in dynamic programming is $4p+3$, if we try to find a matching (packing) of size $p+1$. This reduces the number of colors from $2k$ to $p+3$, and the number of elements for dynamic programming from $4k+2$ to $4p+3$, thus improving the time complexity significantly to $O^*(4.61^{3k})$ for the 3-D MATCHING and 3-SET PACKING problems. Moreover, we can do better for the 3-D MATCHING PROBLEM with this approach. It is observed that if we keep only two columns of the known matching, then the number of elements for dynamic programming is only $8p/3+2$, while the number of colors is only $2p/3+2$. This results in a randomized algorithm of running time $O^*(2.32^{3k})$ and a deterministic algorithm of running time $O^*(2.77^{3k})$ for the 3-D MATCHING problem.

A. Introduction

Matching and packing problems have formed an important class of NP-hard problems. In particular, the 3-D MATCHING problem is one of the six "basic" NP-complete problems in terms of Garey and Johnson [53], and the 3-SET PACKING problem is a natural extension of the 3-D MATCHING problem. There has been a remarkable line of research in the study of parameterized algorithms for 3-D MATCHING and 3-SET PACKING problems.

Downey and Fellows [37] proved that the 3-D MATCHING problem is fixed-parameter tractable and gave an algorithm of time $O^*((3k)!(3k)^{9k+1})$Chen et al. [19] improved the time complexity for 3-D MATCHING to $O^*((5.7k)^k)$, and Jia, Zhang, and Chen [19] improved the time complexity for 3-SET PACKING to $O^*((5.7k)^k)$.

More progress has been made recently. For the 3-SET PACKING problem, Koutis [71] developed a randomized algorithm of time $O^*(10.88^{3k})$ and a deterministic algorithm of time $O^*(2^{O(k)})$. Koutis [71] did not give the exact constant factor in the exponent of the time complexity $O^*(2^{O(k)})$ for his deterministic algorithm. He used the perfect hashing families proposed by Schmidt and Siegel [92], in which the number of hashing functions to hash n elements into $3k$ colors is larger than $2^{\log \log n + 12k}$. It can be derived that his deterministic algorithm has time complexity of at least $O^*(32000^{3k})$. These algorithms can be applied to the 3-D MATCHING problem without any changes. Fellows et al. [44] studied the complexity of matching and packing problems. They first showed that the 3-D MATCHING problem has a kernel of size $O(k^3)$, and then presented an algorithm of time $O^*(2^{O(k)})$ for the problem, where the term $O(k)$ was also not specified in detail. It can be deduced that the running time of the algorithm given in [44] for 3-D MATCHING is at least $O^*(12.67^{3k}T(k))$, where $T(k)$ is the running time of a dynamic programming algorithm that, on a set of triples whose symbols are colored with $13k$ colors, searches for a matching of k triples in which all symbols are colored with distinct colors ($T(k)$ is at least $O^*(10.4^{3k})$ using currently known techniques). The chapter also discussed how these techniques are applied to solve 3-SET PACKING and various graph packing problems.

There are at least two very recent works that give further improved algorithms for 3-D MATCHING and 3-SET PACKING problems. Chen et al. [25] proposed a new technique based on divide-and-conquer that leads to randomized algorithms of time $O^*(2.52^{3k})$ for 3-D MATCHING and 3-SET PACKING problems. Moreover, they proposed a color-coding scheme of $O^*(6.1^k)$ k-colorings which, when combined with standard dynamic programming techniques, gives deterministic algorithms of running time $O^*(12.8^{3k})$ for 3-D MATCHING and 3-SET PACKING problems. We point out that using this new color-coding scheme, the time complexity of the algorithms by Coitus [71] for 3-D MATCHING and 3-SET PACKING can be improved to $O^*(25.6^{3k})$, and the time complexity of the algorithms by Fellows et al. [44] can be improved to $O^*(13.78^{3k})$. In a work performed independently of that in [25], Kneis et al. [70] developed a divide-and-conquer method that leads to randomized algorithms for 3-D MATCHING and 3-SET PACKING problems with time complexity similar to that in [25]. Moreover, a different de-randomization method was proposed in [70] based on the work of [84], which leads to deterministic algorithms of running time $O^*(16^{3k})$ for 3-D MATCHING and 3-SET PACKING problems.

The known parameterized algorithms for 3-D MATCHING and 3-SET PACKING have used either the technique of *greedy localization* [19, 32, 64], the technique of *color-coding* [3] plus dynamic programming [25, 44, 71], or the *divide-and-conquer* method

[25, 70]. In this chapter, we show how a combination of these techniques and new techniques will yield further improved algorithms for these problems. We start with the 3-SET PACKING problem. In difference from the approach used in [19, 64] that constructs a packing of k 3-sets directly from a maximal packing, we concentrate on the construction of a packing of $k+1$ 3-sets based on a packing of k 3-sets. This slight modification enables us to derive a property for packing that is much stronger than the one given in [19]. Moreover, instead of coloring all elements in an instance of 3-SET PACKING, we color only part of the elements and use either ordering or pre-selected elements to reduce the complexity of the coloring stage in the algorithms. Using these new techniques, we are able to develop a parameterized algorithm of running time $O^*(4.61^{3k})$ for the 3-SET PACKING problem, significantly improving the previous best algorithm of running time $O^*(12.8^{3k})$ for the problem [25]. For the 3-D MATCHING problem, we further show that the complexity of the dynamic programming stage in the algorithms, which seems to have been largely neglected in the previous research, can also be improved using a pre-ordering technique. Combining this new technique and those developed for 3-SET PACKING, we achieve further improved algorithms for the 3-D MATCHING problem. More specifically, our new randomized algorithm for 3-D MATCHING runs in time $O^*(2.32^{3k})$, and our new deterministic algorithm for 3-D MATCHING runs in time $O^*(2.77^{3k})$, both significantly improving the previous best algorithms for the problem.

We would like to point out that all previous parameterized algorithms for 3-D MATCHING and 3-SET PACKING have the same time complexity for both problems, although it is obvious that 3-SET PACKING is a nontrivial generalization of 3-D MATCHING. The results in the current chapter seem to give faster algorithms for 3-D MATCHING than for 3-SET PACKING. We also mention that the difference in complexity between our deterministic algorithm (i.e., $O^*(2.77^{3k})$) and our random-

Table I. Comparison of algorithms for 3-D MATCHING

References	Randomized algorithm	Deterministic algorithm
Downey and Fellows [37]		$O^*((3k)!(3k)^{9k+1})$
Chen et al. [19]		$O^*((5.7k)^k)$
Koutis [71]	$O^*(10.88^{3k})$	$> O^*(32000^{3k})$
Fellows et al. [44]*		$> O^*(12.67^{3k}T(k))$
Kneis et al. [70]	$O^*(2.52^{3k})$	$O^*(16^{3k})$
Chen et al. [25]	$O^*(2.52^{3k})$	$O^*(12.8^{3k})$
Our new result	$O^*(2.32^{3k})$	$O^*(2.77^{3k})$

*$T(k)$ is the running time of a dynamic programming process that, on a set of triples whose symbols are colored with $13k$ colors, searches for a matching of k triples in which all symbols are colored with distinct colors. Based on currently known techniques, $T(k)$ is at least $O^*(10.4^{3k})$.

ized algorithm (i.e., $O^*(2.32^{3k})$, which is also currently the best upper bound) for 3-D MATCHING has been significantly narrowed down, which is remarkable considering the fact that in the previous research on the problem, the difference between these two kinds of algorithms is in general very significant. Table I gives a specific comparison of our new algorithms and the previous algorithms for the 3-D MATCHING problem.

B. Preliminaries and Reformulations

Let X, Y, and Z be three pairwise disjoint symbol sets, and let $U = X \times Y \times Z$ be the product set of X, Y, and Z. Each element $t = (x, y, z)$ in U, where $x \in X$, $y \in Y$, and $z \in Z$, is called a *triple*. For a triple $t = (x, y, z)$ in U, denote by $\mathrm{Val}(t)$ the set

$\{x, y, z\}$, and let $\mathrm{Val}^1(t) = \{x\}$, $\mathrm{Val}^2(t) = \{y\}$, $\mathrm{Val}^3(t) = \{z\}$. We say that a triple t_1 *conflicts* with another triple t_2 if $t_1 \neq t_2$ and $\mathrm{Val}(t_1) \cap \mathrm{Val}(t_2) \neq \emptyset$. Let S be a set of triples in U. Denote $\mathrm{Val}(S) = \bigcup_{t \in S} \mathrm{Val}(t)$, and $\mathrm{Val}^i(S) = \bigcup_{t \in S} \mathrm{Val}^i(t)$ for $i = 1, 2, 3$. A *matching* in S is a subset M of triples in S such that no two triples in M conflict with each other. A matching M in S is a *k-matching* if M contains exactly k triples.

Packing problems are a generalization of matching problems. We say that a set ρ_1 *conflicts* with another set ρ_2 if $\rho_1 \neq \rho_2$ and $\rho_1 \cap \rho_2 \neq \emptyset$. Let S be a collection of sets. Denote $\mathrm{Val}(S) = \cup_{\rho \in S} \rho$. A *packing* in S is a sub-collection P of S such that no two sets in P conflict with each other. A packing P in S is a *k-packing* if P contains exactly k sets.

The main problems we study in this chapter are formally defined as follows.

> (PARAMETERIZED) 3-D MATCHING:
>
> Given a pair (S, k), where S is a set of n triples, and k is an integer, either construct a k-matching in S or report that no such matching exists.

> (PARAMETERIZED) 3-SET PACKING:
>
> Given a pair (S, k), where S is a collection of n sets, each containing at most three elements, and k is an integer, either construct a k-packing in S or report that no such packing exists.

A set is a *3-set* if it contains exactly three elements. For an instance (S, k) of 3-SET PACKING, we can assume, without loss of generality, that all sets in S are 3-sets (otherwise, we can add new elements, i.e., elements not in S, to convert each set with fewer than three elements to a 3-set). Instead of working on the above problems, we will concentrate on the following related problems.

3-d matching augmentation:

Given a pair (S, M_k), where S is a set of triples, and M_k is a k-matching in S, either construct a $(k+1)$-matching M_{k+1} in S, or report that no such matching exists.

3-set packing augmentation:

Given a pair (S, P_k), where S is a collection of n 3-sets, and P_k is a k-packing in S, either construct a $(k+1)$-packing P_{k+1} in S, or report that no such packing exists.

Lemma B.1 *For any constant $c > 1$, the* 3-d matching augmentation *problem can be solved in time $O^*(c^k)$ if and only if the* 3-d matching *problem can be solved in time $O^*(c^k)$. Similarly, the* 3-set packing augmentation *problem can be solved in time $O^*(c^k)$ if and only if the* 3-set packing *problem can be solved in time $O^*(c^k)$.*

According to Lemma C.1, we only need to concentrate on the 3-d matching augmentation and 3-set packing augmentation problems.

C. Improved Packing Algorithms

The method of *greedy localization* has been heavily used in early algorithms for matching and packing problems [19, 64]. The method takes advantage of the fact that information for a larger matching/packing can be obtained from a given smaller matching/packing, which narrows down the size of the search space during the construction of the larger matching/packing. We show that this property can be significantly enhanced and more effectively used to develop algorithms for the 3-set packing augmentation problem.

Lemma C.1 *Let* (S, P_k) *be an instance of* 3-SET PACKING AUGMENTATION, *where* P_k *is a* k*-packing in* S*. If* S *also has* $(k{+}1)$*-packings, then there exists a* $(k{+}1)$*-packing* P_{k+1} *in* S *such that every set in* P_k *contains at least two elements in Val*(P_{k+1})*.*

PROOF. We prove the lemma by contradiction. Suppose that the lemma does not hold. Then there is a k-packing P_k such that for every $(k+1)$-packing P in S, there is a set in P_k that contains at most one element in $\text{Val}(P)$. Let P_{k+1} be a $(k+1)$-packing in S such that the number of common sets in P_k and P_{k+1} is maximized over all $(k+1)$-packings in S. By our assumption, there is a set ρ in P_k that contains at most one element in $\text{Val}(P_{k+1})$.

Case 1. Exactly one element a in the set ρ is in $\text{Val}(P_{k+1})$. Then let ρ' be the set in P_{k+1} that contains the element a. Since no other element in ρ is in $\text{Val}(P_{k+1})$, if we replace ρ' in P_{k+1} by ρ, we get a new $(k+1)$-packing that has one more common set (i.e., ρ) with the k-packing P_k (note that ρ' cannot be in P_k because ρ' and ρ share a common element a while ρ contains another two elements not in $\text{Val}(P_{k+1})$). This contradicts our assumption that the $(k+1)$-packing P_{k+1} maximizes the number of common sets with P_k.

Case 2. No element in ρ is in $\text{Val}(P_{k+1})$. Since P_k contains k sets while P_{k+1} contains $k+1$ sets, there must be a set ρ'' in P_{k+1} that is not in P_k. Since ρ contains no element in $\text{Val}(P_{k+1})$, replacing ρ'' in P_{k+1} by ρ gives a new $(k+1)$-packing that has one more common set (i.e., ρ) with P_k, again contradicting the assumption that the $(k+1)$-packing P_{k+1} maximizes the number of common sets with P_k.

This contradiction shows that the set ρ in P_k that contains at most one element in $\text{Val}(P_{k+1})$ cannot exist. $\square$

According to Lemma C.1, to construct a $(k+1)$-packing from a given instance (S, P_k) of 3-SET PACKING AUGMENTATION, we can aim at the $(k+1)$-packing P_{k+1} with the property described in the lemma. The advantage of this $(k+1)$-packing P_{k+1} is that at least $2k$ elements in P_{k+1} are already present in the k-packing P_k, and we only need to identify at most $k+3$ other elements in P_{k+1}. We use the technique of *color-coding*, first introduced by Alon, Yuster, and Zwick [3], to search for these elements that are in P_{k+1} but not in P_k.

Let B be a set of elements. A *coloring* of B is a function mapping B to the natural numbers $\{1, 2, \ldots\}$, and an *h-coloring* of B is a function mapping B to $\{1, 2, \ldots, h\}$. A subset B' of B is *colored properly* by a coloring f if no two elements in B' are colored with the same color under f. A collection $\mathcal{C}$ of h-colorings of a set B is an *h-color coding scheme* if for every subset B' of h elements in B, there is an h-coloring in $\mathcal{C}$ that colors B' properly. The following proposition has been proved in [25].

Proposition C.2 [25] *For any finite set B and any integer h, there is an h-color coding scheme $\mathcal{C}$ of $O^*(6.1^h)$ h-colorings of the set B. Moreover, the h-colorings in $\mathcal{C}$ can be constructed and enumerated in time $O^*(6.1^h)$.*

Let S be a collection of 3-sets and let f be a coloring of the set $\mathrm{Val}(S)$. We say that a packing P in S is *colored properly* if the set $\mathrm{Val}(P)$ is colored properly under the coloring f. Let (S, P_k) be an instance of 3-SET PACKING AUGMENTATION. Since the set of elements that are in $\mathrm{Val}(P_{k+1})$ but not in $\mathrm{Val}(P_k)$ contains at most $k+3$ elements, by introducing $3k$ new colors to properly color the $3k$ elements in P_k, P_{k+1} can be colored properly with at most $4k+3$ colors.

Lemma C.3 *Let (S, P_k) be an instance of* 3-SET PACKING AUGMENTATION*, and let P_{k+1} be a $(k+1)$-packing in S such that each 3-set in P_k contains at least two elements in Val(P_{k+1}). Then there is a collection $\mathcal{C}_0$ of $O^*(6.1^k)$ $(4k+3)$-colorings of the set*

Val(S) *in which at least one properly colors* P_{k+1}*. Moreover, the collection* $\mathcal{C}_0$ *can be constructed in time* $O^*(6.1^k)$*.*

Now we turn to the problem of constructing a properly colored $(k+1)$-packing P_{k+1}. Alon, Yuster, and Zwick [3] in their seminal work on color-coding suggested a general principle in which a $(3k+3)$-coloring that properly colors the $3k+3$ elements is first constructed in Val(P_{k+1}), then a dynamic programming process is applied to find the properly colored $(k+1)$-packing P_{k+1}. Koutis [71] proposed an algebraic formulation to find the properly colored $(k+1)$-packing P_{k+1}. Fellows et al. [44] considered a more general approach that first uses g colors to properly color the $(k+1)$-packing P_{k+1}, where $g \geq 3k+3$, then perform a dynamic programming algorithm. For completeness, we present such a generalized dynamic programming algorithm in detail, as given in Fig. 1, verify its correctness, and analyze its precise complexity.

Lemma C.4 *The algorithm* **3SetPack**(S, k, f, g) *runs in time* $O^*(\sum_{j=0}^{k} \binom{g}{3j})$*, and constructs a properly colored* k*-packing in* S *if such* k*-packings exist.*

Proof. From steps 4.1-4.2 of the algorithm, it can be seen that every collection P of 3-sets added to the super-collection $\mathcal{Q}$ in step 4.4 is a properly colored packing. Therefore, if the algorithm returns a packing in step 5, the packing must be a properly colored k-packing.

For each i, let $S_i = \{\rho_1, \ldots, \rho_i\}$. We prove by induction on i that for all $j \leq k$, if S_i has a properly colored j-packing P_j, then after the i-th execution of the **for**-loop in step 4 of the algorithm, the super-collection $\mathcal{Q}$ contains a properly colored j-packing P'_j such that P_j and P'_j use exactly the same $3j$ colors.

```
Algorithm 3SetPack($S, k, f, g$)
input: A collection $S$ of 3-sets, an integer $k$, a $g$-coloring $f$ of Val($S$)
output: A properly colored $k$-packing if such a packing exists

1.    remove all 3-sets in $S$ in which any two elements have the same color;
2.    Let the remaining 3-sets in $S$ be $\rho_1, \rho_2, \ldots, \rho_n$;
3.    $\mathcal{Q} = \{\emptyset\}$;
4.    for $i = 1$ to $n$ do
4.1.      for each packing $P$ in $\mathcal{Q}$ such that no element in $P$ is colored
           with the same color as an element in $\rho_i$ do
4.2.          $P' = P \cup \{\rho_i\}$;
4.3.          if $P'$ is a $j$-packing with $j \leq k$ and $\mathcal{Q}$ contains no packing that
               uses exactly the same colors as that used by $P'$
4.4.          then add $P'$ to $\mathcal{Q}$;
5.    return a $k$-packing in $\mathcal{Q}$ if such a packing exists.
```

Fig. 1. Dynamic programming for 3-SET PACKING

The initial case $i = 0$ is trivial since $\mathcal{Q} = \{\emptyset\}$. Consider $i \geq 1$. Suppose that the collection S_i has a properly colored j-packing $P_j = \{\rho_{i_1}, \rho_{i_2}, \ldots, \rho_{i_j}\}$, where $1 \leq i_1 < i_2 < \cdots < i_j \leq i$. Then the collection $S_{i_j - 1}$ contains the properly colored $(j-1)$-packing $P_{j-1} = \{\rho_{i_1}, \rho_{i_2}, \ldots, \rho_{i_{j-1}}\}$. By the inductive hypothesis, after the $(i_j - 1)$-st execution of the **for**-loop in step 4, the super-collection $\mathcal{Q}$ contains a properly colored $(j-1)$-packing P'_{j-1} such that the $(j-1)$-packings P_{j-1} and P'_{j-1} use exactly the same $3(j-1)$ colors. Since $P_j = \{\rho_{i_1}, \rho_{i_2}, \ldots, \rho_{i_j}\}$ is a properly colored j-packing, and $P_{j-1} = \{\rho_{i_1}, \rho_{i_2}, \ldots, \rho_{i_{j-1}}\}$ and P'_{j-1} use exactly the same $3(j-1)$ colors, no element in Val(P'_{j-1}) is colored with the same color as an element in the set ρ_{i_j}. Therefore, in the i_j-th execution of the **for**-loop in step 4, a properly colored j-packing $P'_{j-1} \cup \{\rho_{i_j}\}$ will be added to the super-collection $\mathcal{Q}$ if no properly colored j-packing that uses exactly the same $3j$ colors exists in $\mathcal{Q}$ yet. Note that the j-packing $P'_{j-1} \cup \{\rho_{i_j}\}$ and the j-packing P_j use exactly the same $3j$ colors. Therefore, after the i_j-th execution of the **for**-loop in step 4, a j-packing that uses exactly the same $3j$ colors as P_j will exist in the super-collection $\mathcal{Q}$. Finally, since packings in $\mathcal{Q}$ are never removed from $\mathcal{Q}$ and $i_j \leq i$, we conclude that after the i-th execution of the **for**-loop in step 4, a j-packing that uses exactly the same $3j$ colors as P_j will exist in the super-collection $\mathcal{Q}$. This completes the inductive proof.

Now if we let $i = n$, for any $j \leq k$, if the original collection S contains a properly colored j-packing P_j, then the super-collection $\mathcal{Q}$ contains a j-packing that uses exactly the same $3j$ colors as P_j. In particular, if the collection S contains properly colored k-packings, then the algorithm **3SetPack**(S, k, f, g) must return a properly colored k-packing.

Finally, we analyze the complexity of the algorithm. For each $0 \leq j \leq k$ and for each set of $3j$ colors, the super-collection $\mathcal{Q}$ keeps at most one properly colored j-packing that uses exactly these $3j$ colors. Since there are $\binom{g}{3j}$ different subsets of $3j$ colors over a total of g colors, the total number of packings recorded in $\mathcal{Q}$ is bounded by $\sum_{j=0}^{k} \binom{g}{3j}$. For each i, $1 \leq i \leq k$, we examine each packing P in $\mathcal{Q}$ in step 4.1 and check if we can construct a larger packing by adding the set ρ_i to the packing P. This can be done for each packing P in time $O(k)$. In consequence, the algorithm **3SetPack**(S, k, f, g) runs in time $O(nk \sum_{j=0}^{k} \binom{g}{3j}) = O^*(\sum_{j=0}^{k} \binom{g}{3j})$. □

Combining Lemmas C.1, C.3, and C.4, the 3-SET PACKING AUGMENTATION problem can be solved in time $O^*(6.1^k 2^{4k}) = O^*(4.61^{3k})$.

Theorem C.5 *The* 3-SET PACKING AUGMENTATION *problem can be solved in time* $O^*(4.61^{3k})$.

Corollary C.6 *The* 3-SET PACKING *problem can be solved in time* $O^*(4.61^{3k})$.

D. Matching Algorithms Further Improved

All the previous results are applicable to the 3-D MATCHING problem. In fact, if we regard each triple as a 3-set, then each instance S_M of 3-D MATCHING is also an instance S_P of 3-SET PACKING, and a triple set is a matching in S_M if and only if it is a packing in S_P.

As shown in the previous section, a dynamic programming algorithm is used as a second stage in parameterized algorithms for 3-SET PACKING/3-D MATCHING. In this section, we develop a new technique for the dynamic programming stage for the 3-D MATCHING problem so that fewer colors will be needed. This technique has two advantages. First, the use of fewer colors will significantly reduce the time complexity of the coloring stage. Second, since fewer colors are used, the number of different color sets is reduced, which will reduce the time complexity of the dynamic programming stage remarkably.

Let the universal triple set be $U = X \times Y \times Z$, where X, Y, and Z are three pairwise disjoint symbol sets. The symbols in the sets X, Y, and Z will be called the *symbols in column-1*, *column-2*, and *column-3*, respectively.

Definition Let p and q be any two indices in the index set $\{1, 2, 3\}$, and let S be a set of triples in U. A matching M in the set S is (p, q)*-properly colored* by a coloring f of $\mathrm{Val}^p(M) \cup \mathrm{Val}^q(M)$ if no two symbols in $\mathrm{Val}^p(M) \cup \mathrm{Val}^q(M)$ are colored with the same color under f.

Theorem D.1 *Let p and q be any two indices in the index set $\{1, 2, 3\}$. There is an algorithm of time $O^*(\sum_{i=0}^{k} \binom{g}{2i})$ that, on an integer k and a set S of triples in which the symbols in $Val^p(S) \cup Val^q(S)$ are colored by a g-coloring f, constructs a*

```
Algorithm 3DMatch($S, k, f, g; p, q$)
input: A set $S$ of triples, an integer $k$, a $g$-coloring $f$ of the symbols in
        $\mathrm{Val}^p(S) \cup \mathrm{Val}^q(S)$
output: A $(p, q)$-properly colored $k$-matching in $S$ if such a matching exists

1.   remove any triples in $S$ in which any two symbols have the same color
     under $f$;
2.   let the set of remaining triples be $S'$;
3.   $r = \{1, 2, 3\} - \{p, q\}$;
4.   let the symbols in $\mathrm{Val}^r(S')$ be $x_1$, $x_2$, ..., $x_m$;
5.   $\mathcal{Q}_{old} = \{\emptyset\}$; $\mathcal{Q}_{new} = \{\emptyset\}$;
6.   for $i = 1$ to $m$ do
6.1.   for each set $C$ of symbol pairs in $\mathcal{Q}_{old}$ do
6.2.     for each $t \in S'$ with $\mathrm{Val}^r(t) = x_i$ do
6.3.       if no symbol in $C$ is of the same color as a symbol in
           $\mathrm{Val}^p(t) \cup \mathrm{Val}^q(t)$
6.4.       then  $C' = C \cup \{(\mathrm{Val}^p(t), \mathrm{Val}^q(t))\}$;
6.5.             if $C'$ contains no more than $k$ symbol pairs and $\mathcal{Q}_{new}$
                 contains no set of symbol pairs that uses exactly the same
                 colors as that used by $C'$
6.6.                 then add $C'$ to $\mathcal{Q}_{new}$;
6.7.   $\mathcal{Q}_{old} = \mathcal{Q}_{new}$;
7.   return a set $C$ of $k$ symbol pairs in $\mathcal{Q}_{old}$ if such a set exists.
```

Fig. 2. Dynamic programming for 3-D MATCHING

(p, q)-properly colored k-matching in S when such matchings exist in S.

PROOF. Consider the algorithm in Fig. 11. By steps 6.3-6.6, for every set C in the collection $\mathcal{Q}_{old}$, all symbols in C are from $\mathrm{Val}^p(S) \cup \mathrm{Val}^q(S)$, and no two symbols in C are of the same color. The algorithm **3DMatch**$(S, k, f, g; p, q)$ either outputs a set of k symbol pairs in the collection $\mathcal{Q}_{old}$ or reports that no (p, q)-properly colored k-matchings exist in S. We say that a set $C = \{w_1, \ldots, w_i\}$ of i symbol pairs is *extendable to an i-matching* in S if there is an i-matching $M = \{t_1, \ldots, t_i\}$ in S such that for all j, the pair $(\mathrm{Val}^p(t_j), \mathrm{Val}^q(t_j))$ is identical to the symbol pair w_j. For each i, let S'_i be the set of triples in S' whose symbols in column-r are among $\{x_1, x_2, \ldots, x_i\}$. For a matching M, we will denote by $cl(M) = \{f(y) \mid y \in \mathrm{Val}^p(M) \cup \mathrm{Val}^q(M)\}$ the set of colors used by the symbols in $\mathrm{Val}^p(M) \cup \mathrm{Val}^q(M)$.

We prove the following claim by induction on i:

> **Claim.** For each i, $0 \leq i \leq m$, and for all $h \leq k$, there is a (p, q)-properly colored h-matching M_h in S'_i if and only if after the i-th execution of the loop 6.1-6.7 of algorithm **3DMatch**$(S, k, f, g; p, q)$, the collection $\mathcal{Q}_{old}$ contains a set C_h of h symbol pairs such that the set of colors used for the symbols in C_h is exactly $cl(M_h)$. Moreover, each set C_h of h symbol pairs in the collection $\mathcal{Q}_{old}$ after the i-th execution of the loop 6.1-6.7 is extendable to an h-matching in S'_i.

The case $i = 0$ is obvious because we initially set $\mathcal{Q}_{old}$ to $\{\emptyset\}$. Consider $i \geq 1$. First note that the claim is always true for $h = 0$ because the collection $\mathcal{Q}_{old}$ always contains the empty set $\emptyset$ while the set S'_i always contains a 0-matching (which by the definition is (p, q)-properly colored).

Suppose that after the i-th execution of the loop 6.1-6.7, the collection $\mathcal{Q}_{old}$ contains a set C_h of h symbol pairs, where $h \geq 1$. Suppose that the set C_h was created during the j-th execution of the loop 6.1-6.7, where $j \leq i$, by adding a symbol pair $(\mathrm{Val}^p(t), \mathrm{Val}^q(t))$ to a set C_{h-1} of $h - 1$ symbol pairs, where t is a triple with $\mathrm{Val}^r(t) = x_j$ and the set C_{h-1} is contained in $\mathcal{Q}_{old}$ after the $(j-1)$-st execution of the loop 6.1-6.7. By the inductive hypothesis, the set C_{h-1} is extendable to an $(h-1)$-matching M_{h-1} in S'_{j-1}, which is obviously (p, q)-properly colored. Since no symbol in C_{h-1} uses the same color as a symbol in $\mathrm{Val}^p(t) \cup \mathrm{Val}^q(t)$, and the matching M_{h-1}

does not contain the symbol x_j, the set $M_h = M_{h-1} \cup \{t\}$ makes a (p,q)-properly colored h-matching in S'_j. Since $j \leq i$ and $S'_j \subseteq S'_i$, we conclude that the set S'_i contains a (p,q)-properly colored h-matching M_h such that the symbols in the set C_h use exactly the color set $cl(M_h)$. Moreover, it is obvious that the symbol set C_h is extendable to the h-matching M_h.

To prove the other direction, suppose that the set S'_i contains a (p,q)-properly colored h-matching M_h.

Case 1. There is a (p,q)-properly colored h-matching M'_h in S'_j for some $j < i$ such that $cl(M'_h) = cl(M_h)$. By the inductive hypothesis, after the j-th execution of the loop 6.1-6.7, the collection $\mathcal{Q}_{old}$ contains a set C_h of h symbol pairs such that (1) the set of colors used for the symbols of C_h is exactly $cl(M'_h)$; and (2) C_h is extendable to an h-matching in S'_j. Since $j < i$, $S'_j \subseteq S'_i$, and we never remove symbol pairs from $\mathcal{Q}_{old}$, we conclude that in this case, after the i-th execution of the loop 6.1-6.7, the set C_h is still contained in the collection $\mathcal{Q}_{old}$ such that (1) the set of colors used for the symbols of C_h is exactly $cl(M'_h) = cl(M_h)$; and (2) C_h is extendable to an h-matching in $S'_j \subseteq S'_i$.

Case 2. There is no (p,q)-properly colored h-matching M'_h in S'_j for any $j < i$ such that $cl(M'_h) = cl(M_h)$. Then by the inductive hypothesis, after the j-th execution of the loop 6.1-6.7 for any $j < i$, the collection $\mathcal{Q}_{old}$ contains no set C of symbol pairs such that the symbols in C use exactly the color set $cl(M_h)$. Let the (p,q)-properly colored h-matching M_h be $M_h = \{t_1, \cdots, t_h\}$, where for each j, $\mathrm{Val}^r(t_j) = x_{d_j}$, with $d_1 < \cdots < d_{h-1} < d_h$. In this case, we must have $d_h = i$ and $\mathrm{Val}^r(t_h) = x_i$. Let $y = \mathrm{Val}^p(t_h)$ and $z = \mathrm{Val}^q(t_h)$. Since $d_{h-1} < d_h = i$, the triple set $M_{h-1} = M_h - \{t_h\}$ is a (p,q)-properly colored $(h-1)$-matching in S'_{i-1}. By the inductive hypothesis, after the $(i-1)$-st execution of the loop 6.1-6.7 in the algorithm, the collection $\mathcal{Q}_{old}$ contains a set C_{h-1} of $h-1$ symbol pairs such that the set of colors used for the symbols in C_{h-1} is exactly $cl(M_{h-1})$. Now in the i-th execution of the loop 6.1-6.7 when the set C_{h-1} and the triple t_h are examined in step 6.3, a set C of symbol pairs using the color set $cl(M_{h-1}) \cup \{f(y), f(z)\} = cl(M_h)$ will be created. Therefore, after the i-th execution of the loop 6.1-6.7, a set C_h of h symbol pairs using the color set $cl(M_h)$ must be contained in the collection $\mathcal{Q}_{old}$. Suppose that the set C_h was created during the i-th execution by adding a symbol pair $(\mathrm{Val}^p(t'_h), \mathrm{Val}^q(t'_h))$ to a set C'_{h-1} of $h-1$ symbol pairs, where t'_h satisfies $\mathrm{Val}^r(t'_h) = x_i$ and C'_{h-1} is contained in the collection $\mathcal{Q}_{old}$ after the $(i-1)$-st execution of the loop 6.1-6.7 (note that t'_h and C'_{h-1} are not necessarily t_h and C_{h-1}, respectively). By the inductive hypothesis, the set C'_{h-1} is extendable to a (p,q)-properly colored $(h-1)$-matching M'_{h-1} in S'_{i-1}. In consequence, the set C_h is extendable to the (p,q)-properly colored h-matching $M'_{h-1} \cup \{t'_h\}$ in S'_i. This completes the proof of the claim.

By the claim and let $i = m$, the algorithm **3DMatch**$(S, k, f, g; p, q)$ returns a set C_k of k symbol pairs if and only if the triple set S contains a (p,q)-properly colored k-matching, and the set C_k is extendable to a k-matching in S. To construct such a k-matching from C_k, we can use the graph matching technique suggested in [19]. Formally, from the set C_k of symbol pairs, we construct a bipartite graph $B_k = (V_L \cup V_R, E)$, where V_L contains k vertices, corresponding to the k symbol pairs in C_k, and V_R is the set of all symbols in $\mathrm{Val}^r(S)$. There is an edge in B_k from a vertex (y,z) in V_L to a vertex x in V_R if and only if the symbols y, z, and x form a triple in S. It is easy to see that a (p,q)-properly colored k-matching M_k in S can be obtained by constructing a graph matching of k edges in the bipartite graph B_k, which takes polynomial time [28].

In terms of the time complexity of the above algorithm, note that since for each set of $2i$ colors, we record at most one set of symbol pairs that uses exactly these $2i$ colors, the collection $\mathcal{Q}_{old}$ contains at most $\sum_{i=0}^{k} \binom{g}{2i}$ sets of symbol pairs.

For each set C of symbol pairs in $\mathcal{Q}_{old}$, steps 6.2-6.6 of the algorithm take time polynomial in n and k. Therefore, each execution of the loop 6.1-6.7 of the algorithm runs in time $O^*(\sum_{i=0}^{k} \binom{g}{2i})$. In consequence, the running time of the algorithm **3DMatch**$(S, k, f, g; p, q)$ is bounded by $O^*(\sum_{i=0}^{k} \binom{g}{2i})$. □

To solve the 3-D MATCHING AUGMENTATION problem (S, M_k), we only color two columns of a $(k+1)$-matching properly if it exists. In this case, by Lemma C.1, there is a $(k+1)$-matching M_{k+1} such that M_{k+1} has two columns that contain at least $4k/3$ symbols in M_k. Thus at most $2k/3+2$ symbols in these two columns in M_{k+1} are missing in M_k. By introducing $2k$ new colors for each symbol in these two columns in M_k and by Proposition C.2, in time $O^*(6.1^{2k/3})$, the two columns of M_{k+1} can be colored properly into $8k/3+2$ colors. By Theorem D.1, the 3-D MATCHING AUGMENTATION problem (S, M_k) can be solved in time $O^*(6.1^{2k/3}2^{8k/3}) = O^*(2.77^{3k})$.

Theorem D.2 *The* 3-D MATCHING *problem can be solved in time* $O^*(2.77^{3k})$.

If we use a randomized color-coding scheme that properly colors a subset of size k into k colors with high probability in time $O^*(e^k)$ [3], then the time complexity to solve 3-D MATCHING problem can be improved to $O^*(e^{2k/3}2^{8k/3}) = O^*(2.32^{3k})$.

Theorem D.3 *The* 3-D MATCHING *problem can be solved by a randomized algorithm of time* $O^*(2.32^{3k})$.

E. Final Remarks

Recently there has been much interest in parameterized algorithms for *graph packing* problems, i.e., algorithms for constructing k disjoint isomorphic subgraphs in a given graph [45, 70, 83, 86]. In particular, Fellows et al. [45] presented a parameterized algorithm of time $O^*(2^{2k\log k+1.869k})$ for packing k vertex-disjoint triangles in a given graph, and Mathieson, Prieto, and Shaw [83] proposed a parameterized algorithm of time $O^*(2^{4.5k \log k + 4.5k})$ for packing k edge-disjoint triangles in a given graph. Since these problems can be trivially reduced to the 3-SET PACKING problem, by Corollary C.6, they can be solved in time $O^*(4.61^{3k})$. This again gives significant improvements over the previous algorithms.

There are more new results by the time this dissertation is completed: the best randomized algorithm of $O^*(2^k)$ [72] for the 3-D MATCHING and 3-SET PACKING problems, the best deterministic algorithm of $O^*(4^{(r-1)k})$ for the weighted R-D MATCHING problem [46], and the best deterministic algorithm of $O(2^{(2r-1)k})$ for the weighted R-SET PACKING problem.

CHAPTER III

MULTIWAY CUT*

In this chapter, we present an fpt-algorithm of running time $O^*(4^k)$ for the MULTIWAY CUT problem, which significantly improves the previous best algorithm of running time $O^*(4^{k^3})$.

The previous algorithm uses only the parameter k, the size of a multiway cut to search, as the measure. Our algorithm considers both the parameter k and the minimum cut m between a vertex and the other vertices as measures. We find an operation which either decreases k by 1 or increases m by 1. Moreover, we uncover that either such operation is executed or the vertices for further consideration decreases. With these discoveries, we design an algorithm of running time $O^*(4^k)$.

A. Introduction

The MULTITERMINAL CUT problem is a well-known problem, and has been extensively studied ([16, 66, 85]). Applications of this problem are found in distributed computing [96], VLSI [27], computer vision [15], and many other fields. The problem is defined as follows: given an undirected graph $G = (V, E)$ and a set of l vertices $\{t_1, \ldots, t_l\}$ in G (the vertices t_i are called *terminals*), find an edge set E' of minimum size in G such that after the deletion of E', no two terminals are in the same connected component. This problem is NP-hard for general graphs for any fixed integer $l \geq 3$, and is also NP-hard for planar graphs when l is not fixed [30].

A generalization of the MULTITERMINAL CUT problem is the MINIMUM NODE

---

*Reprinted with permission from "An Improved Parameterized Algorithm for the Minimum Node Multiway Cut Problem", by J. Chen, Y. Liu, and S. Liu, 2009, Algorithmica, Volume 55, pages 1-13, Copyright [2009] by Springer.

MULTIWAY CUT problem, which, for a given graph and a given set of terminals, is to find a vertex set S of minimum size such that after the deletion of S, no two terminals are in the same connected component. The MINIMUM NODE MULTIWAY CUT problem is at least as hard as the MULTITERMINAL CUT problem, as the latter can be reduced to the former in time $O(|V| + |E|)$, if we require that no terminal be in the separator S [29]. Therefore, the MINIMUM NODE MULTIWAY CUT problem is also NP-hard if the number l of terminals is at least 3.

When there are only two terminals s and t, the MULTITERMINAL CUT problem and the MINIMUM NODE MULTIWAY CUT problem become the edge version and the vertex version of the MINIMUM s-t CUT problem, respectively. According to the max-flow min-cut theorem [74], the MINIMUM s-t CUT problem, for both the edge version and the vertex version, can be solved via algorithms for the MAXIMUM s-t FLOW problem. For an undirected graph G of n vertices and m edges, the MAXIMUM s-t FLOW problem can be solved in time $O(n^{7/6}m^{2/3})$ [67]. In consequence, the MULTITERMINAL CUT problem and the MINIMUM NODE MULTIWAY CUT problem can also be solved in time $O(n^{7/6}m^{2/3})$.

A natural extension of the MINIMUM NODE MULTIWAY CUT problem is to have a collection of terminal sets, instead of a collection of individual terminals. Formally, let $G = (V, E)$ be an undirected graph, and let $\{T_1, \ldots, T_l\}$ be a collection of *terminal sets* where each T_i is a subset of vertices in G. A *separator* S for $\{T_1, T_2, \ldots, T_l\}$ is a subset of vertices in G such that no vertex in S is in any terminal set, and after deleting S from the graph G, no connected component in the resulting graph contains vertices from more than one terminal set.

In certain real world applications, one may expect that the size of the separator be small. For example, suppose that we are given a network (i.e., a graph) $G = (V, E)$ and a collection of network node groups $\{T_1, \ldots, T_l\}$ in G, and we want to monitor the message communication among the node groups. A separator for $\{T_1, \ldots, T_l\}$ in the network G will well serve for this purpose: any communication path between any two node groups must pass through at least one node in the separator. Therefore, if we set up a monitor process in each of the nodes in the separator, then we can monitor all communications among the node groups. Naturally, we may want to limit the cost of this monitoring system by using only a small number of "monitor nodes" in the network G.

This motivates a parameterized version of the MINIMUM NODE MULTIWAY CUT problem, which will be called the PARAMETERIZED NODE MULTIWAY CUT problem and is defined as follows: given an undirected graph $G = (V, E)$, a collection of pairwise disjoint terminal sets $\{T_1, \ldots, T_l\}$ (where each T_i is a subset of vertices in G), and a parameter k, either construct a separator of at most k vertices in G, or report that no such a separator exists. Our goal is, for the PARAMETERIZED NODE MULTIWAY CUT problem, to develop a *fixed-parameter tractable algorithm* [37], i.e., an algorithm whose running time is of the form $f(k)n^c$ with a function f independent of the input size n and a constant c. In particular, when the parameter value k is small, such a fixed-parameter tractable algorithm will be practically effective. In fact, the study of fixed-parameter tractable algorithms for a variety of parameterized problems has drawn considerable attention recently and has direct impact on real word applications where the selected parameter varies in a small range [37].

It can be derived from the graph minor theory of Robertson and Seymour [37] that there is a fixed-parameter tractable algorithm for the PARAMETERIZED NODE MULTIWAY CUT problem. However, the proof is not constructive. An explicit constructive algorithm for the problem was given by Marx [82], who developed an algorithm of running time $O(n^5 4^{k^3})$ for the PARAMETERIZED NODE MULTIWAY CUT problem for its original version (i.e., in which each terminal set is restricted to contain a single terminal). To our knowledge, this is the only known constructive fixed-parameter tractable algorithm for the problem.

In this chapter, we present an algorithm of running time $O(n^3k4^k)$ for the PARAMETERIZED NODE MULTIWAY CUT problem, which significantly improves the algorithm given in [82]. In the real world of computing, this improvement makes it become possible to practically solve the problem for some reasonable values of the parameter k. For example, for the case of $k = 10$, our algorithm has running time $O(n^3 4^{10})$, which is practically feasible using the currently available computation power. On the other hand, the algorithm in [82] in this case has running time $O(n^5 4^{1000})$, which is totally infeasible from the practical point of view. Theoretically, our result gives the first polynomial time algorithm for the MINIMUM NODE MULTIWAY CUT problem when the size of the optimal separator is of order $O(\log n)$.

Finally, we remark that the techniques we developed in this chapter seem to be very powerful for solving various kinds of multiway cut problems. In particular, very recently the techniques have been extended to directed graphs, and led to a fixed parameter tractable algorithm for the FEEDBACK VERTEX SET problem on directed graphs [24], thus resolving an outstanding open problem in the area of parameterized computation and complexity [37, 34].

B. Minimum V-cuts between Two Terminal Sets

We start with some terminologies. All graphs in our discussion are supposed to be directed.

Let $G = (V, E)$ be a graph and let u and v be two vertices in G. A *path between u and v* is a simple path in G whose two ends are u and v, respectively. We say that there is a *path between a vertex u and a vertex subset V'* if there is a path between the vertex u and a vertex v in the subset V'. For two vertex subsets V_1 and V_2, we say that there is a *path between* V_1 *and* V_2 if there exist a vertex u in V_1 and a vertex v in V_2 such that there is a path between u and v. Two paths are *internally disjoint* if there is no vertex that is an internal vertex for both the paths.

Let G be a graph, and let $\{T_1, \ldots, T_l\}$ be a collection of pairwise disjoint terminal sets (each terminal set is a subset of vertices in G). A subset S of vertices in G is a *separator* for $\{T_1, \ldots, T_l\}$ if S contains no vertex in any of the sets T_1, …, T_l, and if after deleting all vertices in S from G, there is no path between any two different subsets T_i and T_j in the resulting graph. In particular, a separator S for two terminal sets T_1 and T_2 is also called a *V-cut* between the two sets T_1 and T_2.

Let T be a subset of vertices in the graph $G = (V, E)$. By *merging* T (*into a single vertex*), we mean the operation that first deletes all vertices in T then creates a new vertex w adjacent to each v of the vertices in $V - T$ where v is a neighbor of a vertex in T in the original graph G.

Finally, for a subset V' of vertices in the graph G, we will denote by $G(V')$ the subgraph of G that is induced by the vertex subset V'.

Proposition B.1 **[17]** (Menger's Theorem–Vertex Version) *Let u and v be two distinct and nonadjacent vertices in a graph G. Then the maximum number of internally disjoint paths between u and v in G is equal to the size of a minimum V-cut between u and v in G.*

Proposition B.1 can be generalized from the case for two vertices to the case of two vertex subsets, as given in the following lemma.

Lemma B.2 *Let T_1 and T_2 be two disjoint vertex subsets in a graph G such that no vertex in T_1 is adjacent to a vertex in T_2. Then the maximum number h of internally disjoint paths between T_1 and T_2 in G is equal to the size of a minimum V-cut between*

T_1 and T_2 in G. Moreover, for any set π of h internally disjoint paths between T_1 and T_2 in G, every minimum V-cut between T_1 and T_2 in G contains exact one vertex in each of the paths in π.

Proof. Let G' be the graph obtained from the graph G by merging the two vertex subsets T_1 and T_2 into two vertices t_1 and t_2, respectively. Note that t_1 and t_2 are not adjacent in G'.

By the definition of the merge operation, it is easy to verify that a vertex subset S is a V-cut between the vertex subsets T_1 and T_2 in the graph G if and only if S is a V-cut between the vertices t_1 and t_2 in the graph G'. In particular, the size of a minimum V-cut between T_1 and T_2 in G is equal to the size of a minimum V-cut between t_1 and t_2 in G'. Moreover, it is also easy to verify that for any integer h', from a set of h' internally disjoint paths between T_1 and T_2 in G, we can construct a set of h' internally disjoint paths between t_1 and t_2 in G', and *vice versa*. Therefore, the maximum number of internally disjoint paths between T_1 and T_2 in G is equal to the maximum number of internally disjoint paths between t_1 and t_2 in G'. Now the first part of the lemma follows by applying Proposition B.1 to the graph G'.

To prove the second part of the lemma, let S be a minimum V-cut, of size h, between T_1 and T_2 in G, and let π be a set of h internally disjoint paths between T_1 and T_2. The vertex set S must contain at least one vertex from each of the paths in π: otherwise there would be a path between T_1 and T_2 in $G - S$, contradicting the assumption that S is a V-cut between T_1 and T_2. Moreover, the set S cannot contain more than one vertex in any path in π: otherwise S would not be able to contain at least one vertex for each of the paths in π (note that the paths in π are internally disjoint). □

Lemma B.2 provides an efficient algorithm that constructs the maximum number of internally disjoint paths and a minimum-size V-cut between two given vertex subsets in a graph.

Lemma B.3 *Let T_1 and T_2 be two disjoint vertex subsets in a graph $G = (V, E)$ such that no vertex in T_1 is adjacent to a vertex in T_2. Then in time $O((|V| + |E|)k)$, we can decide if the size h of a minimum V-cut between T_1 and T_2 is bounded by k, and in case $h \leq k$, construct h internally disjoint paths between T_1 and T_2.*

PROOF. Let G' be the graph obtained from the graph G by merging the two vertex subsets T_1 and T_2 into two vertices t_1 and t_2, respectively. As discussed in the proof of Lemma B.2, it suffices to show how to decide if the size h of a minimum V-cut between t_1 and t_2 in G' is bounded by k, and in case $h \leq k$, how to construct h internally disjoint paths between t_1 and t_2.

This can be done based on the standard approach to the MAXIMUM t_1-t_2 FLOW problem [17]. For this, we first transform the undirected graph G' into a directed graph by replacing each edge by two reverse arcs. Then we modify the new directed graph by replacing each vertex u (except the vertices t_1 and t_2) by two vertices u_1 and u_2 with an arc from u_1 to u_2, connecting all u's incoming arcs to the vertex u_1 and connecting all u's outgoing arcs to the vertex u_2. Finally we set all edges to have capacity 1. Let the resulting flow graph be G''.

Applying Ford-Fulkerson's standard approach using augmenting paths, in time $O((|V| + |E|)k)$, we can either construct a t_1-t_2 flow of value larger than k in G'', or end up with a maximum t_1-t_2 flow of value h bounded by k. In the former case, we conclude that the size of a minimum V-cut between t_1 and t_2 in G' is larger than k, which implies that the size of a minimum V-cut between T_1 and T_2 in G is larger than

k. In the latter case, h internally disjoint paths between t_1 and t_2 in G' can be easily constructed from the maximum t_1-t_2 flow of value h in G'', from which h internally disjoint paths between T_1 and T_2 in G can be constructed. $\square$

C. The Main Algorithm

Now we return back to the PARAMETERIZED NODE MULTIWAY CUT problem. Formally, an instance $(G, \{T_1, \ldots, T_l\}, k)$ of the PARAMETERIZED NODE MULTIWAY CUT problem consists of an undirected graph G, a collection $\{T_1, \ldots, T_l\}$ of pairwise disjoint *terminal sets* (each terminal set is a vertex subset in G), and a parameter k. The objective is to either construct a separator of at most k vertices for $\{T_1, \ldots, T_l\}$, or conclude that no such a separator exists.

Before we formally present our algorithm, we give a less formal but intuitive explanation on the basic idea of the algorithm. Let the size of a minimum V-cut between T_1 and $\bigcup_{j \neq 1} T_j$ be m.

Pick a vertex u that is not in any terminal set and has a neighbor in T_1. If u also has a neighbor in another terminal set T_i, $i \neq 1$, then we can directly include u in the separator (this is necessary because the separator must separate T_1 and T_i), and recursively find a separator of size $k - 1$ in the remaining graph. On the other hand, if u has no neighbor in other terminal sets, then we compute the size m' of a minimum V-cut between the sets $T_1' = T_1 \cup \{u\}$ and $\bigcup_{i \neq 1} T_i$. It can be proved that we must have $m \leq m'$. Note that by Lemma B.3, the values m and m' can be computed in polynomial time.

In the case $m = m'$, we will show that the instance $(G, \{T_1, T_2, \ldots, T_l\}, k)$ has a separator of size bounded by k if and only if the instance $(G, \{T_1', T_2, \ldots, T_l\}, k)$ has a separator of size bounded by k. Then we recursively work on the new instance

$(G, \{T_1', T_2, \ldots, T_l\}, k)$. Thus, in the case of $m = m'$, we can reduce the number of vertices that are not in the separator by 1.

On the other hand, suppose $m < m'$. Then we branch on the vertex u in two cases, one includes u in the separator and the other excludes u from the separator. In the case of including the vertex u in the separator, we recursively work on the instance $(G - \{u\}, \{T_1, T_2, \ldots, T_l\}, k - 1)$, in which the parameter value is decreased by 1; and in the case of excluding the vertex u from the separator, we recursively work on the instance $(G, \{T_1', T_2, \ldots, T_l\}, k)$, in which the size of the minimum V-cut between T_1' and $\bigcup_{i \neq 1} T_i$ is increased by at least 1.

Therefore, for the given instance $(G, \{T_1, T_2, \ldots, T_l\}, k)$, we can either (1) apply a polynomial time process that either decreases the parameter value by 1 or reduces the number of vertices not in the separator by 1, or (2) branch into two cases, of which one decreases the parameter value by 1 and the other increases the value m by at least 1 (see the definition of m given in the second paragraph in this section). Note that all these generated new instances will be "simpler" than the original given instance: (i) reducing the number of vertices not in the separator will narrow down our search space for the separator; (ii) an instance of parameter value bounded by 1 can be solved in polynomial time; and (iii) an instance in which the value m is larger than the parameter value k obviously has no separator of size bounded by k.

To present our formal discussions, we fix an instance $(G, \{T_1, \ldots, T_l\}, k)$ of the PARAMETERIZED NODE MULTIWAY CUT problem, where $G = (V, E)$ is a graph, and $\{T_1, \ldots, T_l\}$ is a collection of terminal sets in G. Let the size of a minimum V-cut between T_1 and $\bigcup_{j \neq 1} T_j$ be m. Moreover, fix a vertex u that is not in any of the terminal sets but has a neighbor in the terminal set T_1. Let $T_1' = T_1 \cup \{u\}$.

Lemma C.1 *Let m be the size of a minimum V-cut between the two sets T_1 and*

$\bigcup_{j \neq 1} T_j$, and let m' be the size of a minimum V-cut between the two sets T_1' and $\bigcup_{j \neq 1} T_j$. Then $m' \geq m$.

Proof. The lemma follows from the observation that every V-cut between the sets T_1' and $\bigcup_{j \neq 1} T_j$ is also a V-cut between the sets T_1 and $\bigcup_{j \neq 1} T_j$. $\square$

The following theorem is the most crucial observation for our algorithm.

Theorem C.2 *If the minimum V-cuts between the sets T_1 and $\bigcup_{j \neq 1} T_j$ and the minimum V-cuts between the sets T_1' and $\bigcup_{j \neq 1} T_j$ have the same size, then the instance $(G, \{T_1, T_2, \ldots, T_l\}, k)$ has a separator of size bounded by k if and only if the instance $(G, \{T_1', T_2, \ldots, T_l\}, k)$ has a separator of size bounded by k.*

Proof. If the instance $(G, \{T_1', T_2, \ldots, T_l\}, k)$ has a separator S of size bounded by k, then it is obvious that S is also a separator for the instance $(G, \{T_1, T_2, \ldots, T_l\}, k)$. In consequence, the instance $(G, \{T_1, T_2, \ldots, T_l\}, k)$ also has a separator of size bounded by k.

Now we consider the other direction. Suppose that the instance $(G, \{T_1, T_2, \ldots, T_l\}, k)$ has a separator S_k of size bounded by k.

To simplify the discussion, denote by T_{other} the set $\bigcup_{j \neq 1} T_j$. Let S_m be a minimum V-cut between T_1' and T_{other} (note that S_m does not contain u). Then S_m is also a V-cut between T_1 and T_{other}. In fact, by the assumption of the theorem, S_m is also a minimum V-cut between T_1 and T_{other}. Let $C(T_1)$ be the set of vertices x such that either $x \in T_1$ or there is a path between x and T_1 in the subgraph $G - S_m$. In particular, since u is not in S_m and u is adjacent to T_1, we have $u \in C(T_1)$. Moreover, let $C(T_{other}) = V - C(T_1) - S_m$.

By Lemma B.2, there exist $|S_m|$ internally disjoint paths between T_1 and T_{other},

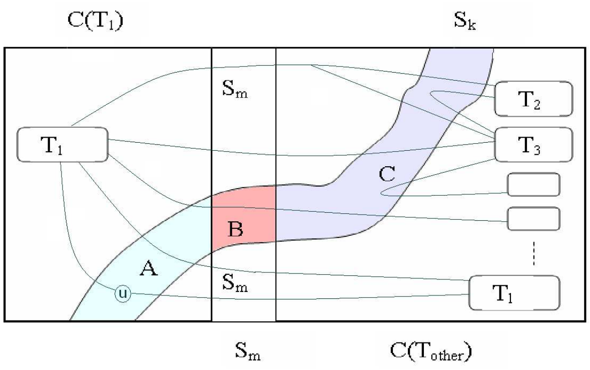

Fig. 3. Decomposition of separators

each contains exactly one vertex in the set S_m. Therefore, each of these $|S_m|$ paths is cut into two subpaths by a vertex in S_m, such that one subpath is in the induced subgraph $G(C(T_1))$ and the another subpath is in the induced subgraph $G(C(T_{other}))$. From this, we derive that there are $|S_m|$ internally disjoint paths between T_1 and S_m in the induced subgraph $G(C(T_1) \cup S_m)$, each contains a distinct vertex in the set S_m.

Define $A = S_k \cap C(T_1)$, $B = S_k \cap S_m$, and $C = S_k \cap C(T_{other})$. Finally, let S'_m be the set of vertices x in S_m such that there is a path between x and T_{other} in the induced subgraph $G(C(T_{other}) \cup S_m - S_k)$ (see Figure 3 for an intuitive illustration of these sets).

We first prove that $|A| \geq |S'_m|$.

From the fact that there are $|S_m|$ internally disjoint paths between T_1 and S_m in the induced subgraph $G(C(T_1) \cup S_m)$ in which each path contains a distinct vertex in the set S_m, we derive that there are $|S'_m|$ internally disjoint paths between T_1 and S'_m in the induced subgraph $G(C(T_1) \cup S'_m)$. If $|A| < |S'_m|$, then there must be a path P_1 between T_1 and a vertex v' in S'_m in the subgraph $G(C(T_1) \cup S'_m - A) = G(C(T_1) \cup S'_m - S_k)$. Moreover, by the definition of the set S'_m, there is also a path P_2 between v' and T_{other} in the induced subgraph $G(C(T_{other}) \cup S_m - S_k)$. The concatenation of the paths P_1 and P_2 would give a path between T_1 and T_{other} in the induced subgraph $G(V-S_k)$, which contradicts the assumption that S_k is a separator of the instance $(G, \{T_1, T_2, \ldots, T_l\}, k)$. Therefore, we must have $|A| \geq |S'_m|$.

Define a set $S'_k = S'_m \cup B \cup C$. We now prove that the set S'_k is a separator of the instance $(G, \{T'_1, T_2, \ldots, T_l\}, k)$. Suppose that the set S'_k is not a separator of the instance $(G, \{T'_1, T_2, \ldots, T_l\}, k)$, then there are two vertices v_1 and v_2 that are in two different terminal sets in $\{T'_1, T_2, \ldots, T_l\}$ and there exists a path P between v_1 and v_2 in the induced subgraph $G(V - S'_k)$. We discuss this in two possible cases.

Case 1: There is a vertex w in the path P such that $w \in C(T_1)$. Because (1) at least one of the vertices v_1 and v_2 is in the set T_{other}, (2) there is a path between T_1 and w in the induced subgraph $G(C(T_1))$, and (3) S_m is a V-cut between T_1 and T_{other}, we conclude that there must be a vertex $s \in S_m$ that is also on the path P. Without loss of generality, we can suppose that the vertex v_1 is in the set T_{other}, and that the subpath P' of P that begins from v_1 and ends at s has no vertices from $C(T_1)$ – for this we only have to pick the first vertex s in S_m when we traverse on the path P from v_1 to v_2. Then the path P' is in the induced subgraph $G(C(T_{other}) \cup S_m - S'_k)$, which is a subgraph of the induced subgraph $G(C(T_{other}) \cup S_m - S_k)$. Now by the definition of the set S'_m, the vertex s is in the set S'_m, thus in the set S'_k. But this is impossible because we assumed that the path P is in the induced subgraph $G(V - S'_k)$.

Case 2: All vertices of the path P are in $V - S'_k - C(T_1)$. Then neither of the vertices v_1 and v_2 can be from the set T_1. Moreover, since the induced subgraph $G(V - S'_k - C(T_1))$ is a subgraph of the induced subgraph $G(V - S_k)$, the path P, which is between two different terminal sets in $\{T_2, \ldots, T_l\}$, would contain no vertex in S_k. But this again contradicts the assumption that S_k is a separator of the instance $(G, \{T_1, T_2, \ldots, T_l\}, k)$.

Combining the discussions in Case 1 and Case 2, we conclude that the set S'_k is a separator for the instance $(G, \{T'_1, T_2, \ldots, T_l\}, k)$.

Since $|A| \geq |S'_m|$, $S_k = A \cup B \cup C$, and $S'_k = S'_m \cup B \cup C$, and A does not intersect $B \cup C$, we conclude that $|S_k| \geq |S'_k|$. In particular, if the instance $(G, \{T_1, T_2, \ldots, T_l\}, k)$ has the separator S_k of size bounded by k, then the instance $(G, \{T'_1, T_2, \ldots, T_l\}, k)$ has the separator S'_k of size also bounded by k.

This completes the proof of the theorem. □

The proof of Theorem C.4 becomes complicated partially because the vertex u may be included in a separator for the instance $(G, \{T_1, T_2, \ldots, T_l\}, k)$. If we restrict that the vertex u is not in the separators for the instance $(G, \{T_1, T_2, \ldots, T_l\}, k)$, then a result similar to Theorem C.4 can be obtained much more easily, even without the need of the condition that the minimum V-cuts between T_1 and $\bigcup_{j \neq 1} T_j$ and the minimum V-cuts between T'_1 and $\bigcup_{j \neq 1} T_j$ have the same size. This is given in the following lemma. This result will also be needed in our algorithm.

Lemma C.3 *Let S be a vertex subset in the graph G such that S does not include the vertex u. Then S is a separator for the instance $(G, \{T_1, T_2, \ldots, T_l\}, k)$ if and only if S is a separator for the instance $(G, \{T'_1, T_2, \ldots, T_l\}, k)$.*

Proof. If S is a separator for the instance $(G, \{T'_1, T_2, \ldots, T_l\}, k)$, then as explained in Theorem C.4, S is also a separator for the instance $(G, \{T_1, T_2, \ldots, T_l\}, k)$.

We are done once we show the other direction. Suppose that S is a separator for the instance $(G, \{T_1, T_2, \ldots, T_l\}, k)$. We show that S is also a separator for the instance $(G, \{T'_1, T_2, \ldots, T_l\}, k)$. Suppose that S is not a separator for the instance $(G, \{T'_1, T_2, \ldots, T_l\}, k)$. Then there is a path P in $G - S$ between two different terminal sets in $\{T'_1, T_2, \ldots, T_l\}$. Let one of these two terminal sets in $\{T'_1, T_2, \ldots, T_l\}$ be T_i, where $i \neq 1$. The path P must contain the vertex u (recall that S does not contain u) – otherwise the path P in $G - S$ would be between two different terminal sets in $\{T_1, T_2, \ldots, T_l\}$, contradicting the assumption that S is a separator for $(G, \{T_1, T_2, \ldots, T_l\}, k)$. However, this would imply that the path from T_1 to u (recall that u has a neighbor in T_1) then following the path P to the terminal set T_i would give a path in $G - S$ between T_1 and T_i, again contradicting the assumption that S is a separator for $(G, \{T_1, T_2, \ldots, T_l\}, k)$. This contradiction shows that the set S must be also a separator for the instance $(G, \{T_1', T_2, \ldots, T_l\}, k)$. $\square$

Now, we are ready to present our algorithm. For an instance $(G, \{T_1, \ldots, T_l\}, k)$ of the PARAMETERIZED NODE MULTIWAY CUT problem, a vertex in the graph G that does not belong to any terminal sets will be called a "non-terminal".

The algorithm is given in Figure 10.

Theorem C.4 *The algorithm* **NMC**$(G, \{T_1, T_2, \ldots, T_l\}, k)$ *in Figure 10 solves the* PARAMETERIZED NODE MULTIWAY CUT *problem in time* $O(n^3 k 4^k)$.

PROOF. We first prove the correctness of the algorithm. Let $(G, \{T_1, T_2, \ldots, T_l\}, k)$ be an input to the algorithm, which is an instance of the PARAMETERIZED NODE MULTIWAY CUT problem, where $G = (V, E)$ is a graph, $\{T_1, T_2, \ldots, T_l\}$ is a collection of terminal sets, and k is the upper bound of the size of the separator we are looking for.

If there is an edge whose two ends are in two different terminal sets, then we have no way to separate these two terminal sets since all vertices in a separator are supposed to be non-terminals. Step 1 handles this case correctly.

If a non-terminal w has two neighbors that are in two different terminal sets, then w must be in the separator because otherwise the two terminal sets will not be separated. Thus, we can simply include the vertex w in the separator, and recursively find a separator of size bounded by $k - 1$ for the same collection of terminal sets

```
Algorithm NMC($G, \{T_1, T_2, \ldots, T_l\}, k$)
input: an instance $(G, \{T_1, T_2, \ldots, T_l\}, k)$ of the PARAMETERIZED NODE
        MULTIWAY CUT problem $(l \geq 2)$
output: a separator of size bounded by $k$ for $(G, \{T_1, T_2, \ldots, T_l\}, k)$,
        or report "No" (i.e., no such a separator)
1.  if an edge has its two ends in two different terminal sets
      then return "No";
2.  if a non-terminal $w$ has two neighbors in two different terminal sets
      then return $w + \mathbf{NMC}(G - w, \{T_1, \ldots, T_l\}, k - 1)$;
3.  find the size $m_1$ of a minimum V-cut between $T_1$ and $\bigcup_{j=2}^{l} T_j$;
4.  if $m_1 > k$ then return "No";
5.  if ($m_1 = 0$ and $l = 2$) then return $\emptyset$;
5.1 if ($m_1 = 0$ and $l > 2$) then return $\mathbf{NMC}(G, \{T_2, \ldots, T_l\}, k)$;
6.  else pick a non-terminal $u$ that has a neighbor in $T_1$; let $T_1' = T_1 + u$;
6.1     if the size of a minimum V-cut between $T_1'$ and $\bigcup_{j=2}^{l} T_j$ is equal to $m_1$
          then return $\mathbf{NMC}(G, \{T_1', T_2, \ldots, T_l\}, k)$;
6.2     else $S = u + \mathbf{NMC}(G - u, \{T_1, T_2, \ldots, T_l\}, k - 1)$;
            if $S$ is not "No" then return $S$;
6.3         else return $\mathbf{NMC}(G, \{T_1', T_2, \ldots, T_l\}, k)$.
```

Fig. 4. An algorithm for the PARAMETERIZED NODE MULTIWAY CUT problem

$\{T_1, T_2, \ldots, T_l\}$ in the remaining graph $G - w$. This case is correctly handled by step 2.[2]

Step 3 computes the size m_1 of a minimum V-cut between the sets T_1 and $\bigcup_{j=2}^{l} T_j$. By Lemma B.3, the value m_1 can be computed in time $O((|V| + |E|)k)$.

If $m_1 > k$, then the size of a minimum V-cut between T_1 and $\bigcup_{j=2}^{l} T_j$ is larger than k, which means that even separating the set T_1 from the other sets $\bigcup_{j=2}^{l} T_j$

---

[2] To simplify the expression, we suppose that "No" plus any vertex set gives a "No". Therefore, step 2 will return a "No" if $\mathbf{NMC}(G - w, \{T_1, \ldots, T_l\}, k - 1)$ returns a "No".

requires more than k vertices. Thus, no separator of size bounded by k can exist for the terminal sets T_1, T_2, …, T_l. This is handled by step 4.

In step 5 we handle the case $m_1 = 0$ and $l = 2$, which we do not need to remove any vertex to separate T_1 and T_2, i.e. the problem is solved. So we return an empty set $\emptyset$ as a separator of size 0 (note that because of step 4, here we must have $k \geq 0$). In step 5.1, $m_1 = 0$ and $l > 2$, which means that T_1 is already separated from $T_2, \ldots, T_l$. Hence we only need to find a separator to separate $T_2, \ldots, T_l$. Therefore, step 5.1 handles this case correctly.

When the algorithm reaches step 6, the following conditions hold true: (1) no edge has its two ends in two different terminal sets (because of step 1); (2) no non-terminal has two neighbors in two different terminal sets (because of step 2); (3) $0 < m_1 \leq k$ (because of steps 4-5). In particular, by condition (3), there must be a non-terminal u that has a neighbor in T_1.

Let m' be the size of a minimum V-cut between the sets T_1' and $\bigcup_{j=2}^{l} T_j$. If $m' = m_1$, then by Theorem C.4, the instance $(G, \{T_1, T_2, \ldots, T_l\}, k)$ has a separator of size bounded by k if and only if the instance $(G, \{T_1', T_2, \ldots, T_l\}, k)$ has a separator of size bounded by k. In particular, as shown in the proof of Theorem C.4, a separator of size bounded by k for the instance $(G, \{T_1', T_2, \ldots, T_l\}, k)$ is actually also a separator for the instance $(G, \{T_1, T_2, \ldots, T_l\}, k)$. Therefore, in this case, we can recursively work on the instance $(G, \{T_1', T_2, \ldots, T_l\}, k)$, as given in step 6.1. On the other hand, if $m' \neq m_1$, which means $m' > m_1$, then we simply branch on the vertex u in two cases: (1) including u in the separator and recursively working on the remaining graph for a separator of size bounded by $k-1$, as given by step 6.2; and (2) excluding u from the separator thus looking for a separator that does not include u and is of size bounded by k for the instance $(G, \{T_1, T_2, \ldots, T_l\}, k)$. By Lemma C.1, the second case is equivalent to finding a separator of size bounded by k for the instance

$(G, \{T_1', T_2, \ldots, T_l\}, k)$. This case is thus handled by step 6.3.

This completes the proof of the correctness of the algorithm. Now we analyze the complexity of the algorithm.

The recursive execution of the algorithm can be described as a search tree $\mathcal{T}$. We first count the number of leaves in the search tree $\mathcal{T}$. Note that only steps 6.2-6.3 of the algorithm correspond to branches in the search tree $\mathcal{T}$. Let $D(k, m_1)$ be the total number of leaves in the search tree $\mathcal{T}$ for the algorithm $\mathbf{NMC}(G, \{T_1, T_2, \ldots, T_l\}, k)$, where m_1 is the size of a minimum V-cut between the sets T_1 and $\bigcup_{j=2}^{l} T_j$. Then steps 6.2-6.3 induce the following recurrence relation:

$$D(k, m_1) \leq D(k-1, m'') + D(k, m''') \tag{3.1}$$

where m'' is the size of a minimum V-cut between T_1 and $\bigcup_{j=2}^{l} T_j$ *in the graph* $G - u$ as given in step 6.2, and m''' is the size of a minimum V-cut between T_1' and $\bigcup_{j=2}^{l} T_j$ *in the graph* G as given in step 6.3. Note that $m_1 - 1 \leq m'' \leq m_1$ because removing the vertex u from G cannot increase the size of a minimum V-cut between two sets, and can decrease the size of a minimum V-cut between the two sets by at most 1. Moreover, by Lemma C.1 and because of step 6.1, the size m''' of a minimum V-cut between T_1' and $\bigcup_{j=2}^{l} T_j$ in step 6.3 is at least $m_1 + 1$. Summarizing these, we have

$$m_1 - 1 \leq m'' \leq m_1 \qquad \text{and} \qquad m''' \geq m_1 + 1 \tag{3.2}$$

Introduce a new function D' such that $D'(2k - m_1) = D(k, m_1)$, and let $t = 2k - m_1$. Then by Inequalities (5.1) and (5.2), the branch in step 6.2-6.3 in the algorithm becomes

$$D'(t) \leq D'(t_1) + D'(t_2)$$

where when $t = 2k - m_1$ then $t_1 = 2(k-1) - m'' \leq t - 1$, and $t_2 = 2k - m''' \leq t - 1$.

We also point out that certain non-branching steps (i.e., steps 2, 5.1, and 6.1) may also change the values of k and m_1, thus changing the value $t = 2k - m_1$. However, none of these steps increases the value $t = 2k - m_1$: (1) step 2 decreases the value k by 1 and the value m_1 by at most 1, which as a total will decrease the value $t = 2k - m_1$ by at least 1; (2) step 5.1 keeps the value k unchanged and, since we have $m_1 = 0$ before the execution of this step, the new value m_1 is at least as large as the old value m_1. As a consequence, the value $t = 2k - m_1$ is not increased; (3) finally, step 6.1 does not change the values k and m_1, thus neither changes the value $t = 2k - m_1$. In summary, the value $t = 2k - m_1$ after a branching step to the next branching step can never be increased.

Our initial instance starts with $t = 2k - m_1 \leq 2k$. In the case $t = 2k - m_1 = 0$, because we also have the conditions $k \geq m_1 \geq 0$, we must have $m_1 = 0$ and $k = 0$, in this case the algorithm can solve the instance without further branching. Therefore, we have $D'(0) = 1$. Combining all these, we derive

$$D(k, m_1) = D'(2k - m_1) \leq 2^{2k},$$

and the search tree $\mathcal{T}$ has at most 2^{2k} leaves.

Finally, it is easy to verify that along each root-leaf path in the search tree $\mathcal{T}$, the running time of the algorithm is bounded by $O(n^3 k)$, where n is the number of vertices in the graph. In conclusion, the running time of the algorithm $\mathbf{NMC}(G, \{T_1, T_2, \ldots, T_l\}, k)$ is bounded by $O(n^3 k 4^k)$.

This completes the proof of the theorem. □

D. Final Remarks

We developed new and powerful techniques that lead to an algorithm of running time $O(n^3k4^k)$ for a generalized version of the PARAMETERIZED NODE MULTIWAY CUT problem. The algorithm significantly improves previous algorithms for the problem. More recently, our techniques have been extended to directed graphs that lead to a fixed parameter tractable algorithm for the FEEDBACK VERTEX SET problem [24], thus resolving an outstanding open problem in parameterized computation and complexity.

Our algorithm finds a separator that has no vertices in any terminal set. We call such a separator a *restricted separator*. If a separator is allowed to include vertices from terminal sets, the separator is called an *unrestricted separator*. It can be verified easily that the instance $(G, \{T_1, \ldots, T_l\}, k)$ has an unrestricted separator of size k if and only if the instance $(G', \{\{x_1\}, \ldots, \{x_l\}\}, k)$ has a restricted separator of size k, where the graph G' is obtained from the graph G by adding l new vertices $x_1, \ldots, x_l$ and connecting x_i to each vertex in T_i for all $1 \leq i \leq l$. Therefore, our algorithm can also be used to construct unrestricted separators for undirected graphs.

One related problem is the PARAMETERIZED NODE MULTICUT problem [82], where we look for a separator of size k to separate each of the l given pairs of terminals. When both k and l are used as parameters, based on the techniques developed in the current chapter, the fixed parameter tractable algorithm presented in [82] for the PARAMETERIZED NODE MULTICUT problem can be improved. On the other hand, if only k is used as the parameter, or if the graph G is a directed graph (or even just a directed acyclic graph), it is currently unknown whether the PARAMETERIZED NODE MULTICUT problem has fixed parameter tractable algorithms, which seem very interesting topics for further research.

CHAPTER IV

UNDIRECTED FEEDBACK VERTEX SET*

In this chapter, we give an fpt-algorithm of running time $O^*(5^k)$ for the FEEDBACK VERTEX SET problem in weighted graphs. We first present an fpt-algorithm for the FEEDBACK VERTEX SET problem in unweighted graphs, then extend that algorithm for the FEEDBACK VERTEX SET problem in weighted graphs.

The previous algorithms have been focused on the parameter k, the size of a feedback vertex set to search, as the only measure. The previous algorithm takes the approach based on the iterative compression method [90]. Our algorithm still takes the same approach. But we consider both the parameter k and the number of connected components when we apply the iterative compression method. We discover an operation which either decreases k by 1 or decreases the number of connected components by 1. When either k become 0 or the number of connected component is 1, the FEEDBACK VERTEX SET problem can be solved in polynomial time. With these interesting properties, we design an algorithm of running time $O^*(5^k)$ for the FEEDBACK VERTEX SET problem in weighted graphs.

A. Introduction

Let G be an undirected graph. A *feedback vertex set* (FVS) F in G is a set of vertices in G whose removal results in an acyclic graph (or equivalently, every cycle in G contains at least one vertex in F). The problem of finding a minimum feedback vertex set in a graph is one of the classical NP-complete problems [68] and has many

---

*Reprinted with permission from "Improved algorithms for feedback vertex set problems", by J. Chen, F. V. Fomin, Y. Liu, S. Liu, and Y. Villanger, 2008, Journal of Computer and System Sciences, volume 74, pages 1188-1198, Copyright [2008] by Elsevier Inc.

applications. The history of the problem can be traced back to early '60s. For several decades, many different algorithmic approaches were tried on this problem, including approximation algorithms, linear programming, local search, polyhedral combinatorics, and probabilistic algorithms (see the survey of Festa et al. [47]). There are also exact algorithms finding a minimum FVS in a graph of n vertices in time $\mathcal{O}(1.9053^n)$ [89] and in time $\mathcal{O}(1.7548^n)$ [48].

An important application of the FVS problem is *deadlock recovery* in operating systems [94], in which a deadlock is presented by a cycle in a *system resource-allocation graph* G. Therefore, in order to recover from deadlocks, we need to abort a set of processes in the system, i.e., to remove a set of vertices in the graph G, so that all cycles in G are broken. Equivalently, we need to find an FVS in G. The problem also has a version on weighted graphs, where the weight of a vertex can be interpreted as the cost of aborting the corresponding process. In this case, we are looking for an FVS in G whose weight is minimized.

In a practical system resource-allocation graph G, it can be expected that the size k of the minimum FVS in G, i.e., the number of vertices in the FVS, is fairly small. This motivated the study of *parameterized algorithms* for the FVS problem that find an FVS of k vertices in a graph of n vertices (where the weight of the FVS is minimized, in the case of weighted graphs), and run in time $f(k)n^{\mathcal{O}(1)}$ for a fixed function f (thus, the algorithms become practically efficient when the value k is small).

This line of research has received considerable attention, mostly on the unweighted version of the problem. The first group of parameterized algorithms of running time $f(k)n^{\mathcal{O}(1)}$ for the FVS problem on unweighted graphs was given by Bodlaender [10] and by Downey and Fellows [35]. Since then a chain of dramatic improvements was obtained by different researchers (see table II for references).

Table II. History of parameterized algorithms for the UNWEIGHTED FEEDBACK VERTEX SET problem

Bodlaender, Fellows [10, 35]	$\mathcal{O}(17(k^4)!n^{\mathcal{O}(1)})$
Downey and Fellows [37]	$\mathcal{O}((2k+1)^k n^2)$
Raman et al.[87]	$\mathcal{O}(\max\{12^k, (4\log k)^k\}n^{2.376})$
Kanj et al.[65]	$\mathcal{O}((2\log k + 2\log\log k + 18)^k n^2)$
Raman et al.[88]	$\mathcal{O}((12\log k/\log\log k + 6)^k n^{2.376})$
Guo et al.[57]	$\mathcal{O}((37.7)^k n^2)$
Dehne et al.[33]	$\mathcal{O}((10.6)^k n^3)$

Randomized parameterized algorithms have also been studied in the literature for the FVS problem, for both unweighted and weighted graphs. The best known randomized parameterized algorithms for the FVS problems are due to Becker et al. [6], who developed a randomized algorithm of running time $\mathcal{O}(4^k kn^2)$ for the FVS problem on unweighted graphs, and a randomized algorithm of running time $\mathcal{O}(6^k kn^2)$ for the FVS problem on weighted graphs. To our knowledge, no deterministic algorithm of running time $f(k)n^{O(1)}$ for any function f was known prior to our results for the weighted FVS problem.

The main result of this chapter is an algorithm that for a given integer k and a weighted graph G, either finds a minimum weight FVS in G of at most k vertices, or correctly reports that G contains no FVS of at most k vertices. The running time of our algorithm is $\mathcal{O}(5^k kn^2)$. This improves and generalizes a long chain of results in parameterized algorithms. Let us remark that the running time of our (deterministic) algorithm comes close to that of the best randomized algorithm for the FVS problem on unweighted graphs and is better than the running time of the previous best randomized algorithm for the FVS problem on weighted graphs.

The general approach of our algorithm is based on the *iterative compression* method [90], which has been successfully used recently for improved algorithms for the FVS and other problems [33, 57, 90]. The method starts with an FVS of k vertices for a small subgraph of the given graph, and iteratively grows the small subgraph while keeping an FVS of k vertices in the grown subgraph until the subgraph becomes the original input graph. This method makes it possible to reduce the original FVS problem on general graphs to the FVS problem on graphs with a special decomposition structure. The main contribution of the current chapter is the development of a general algorithmic technique that identifies a dual parameter in problem instances that limits the number of times where the original parameter k cannot be effectively reduced during a branch and search process. In particular, for the FVS problem on graphs of the above special decomposition structure, a measure is introduced that nicely combines the original parameter and the dual parameter and bounds effectively the running time of a branch and search algorithm for the FVS problem. This technique leads to a simpler but significantly more efficient parameterized algorithm for the FVS problem on unweighted graphs. Moreover, the introduction of the measure greatly simplifies the process of degree-2 vertices in a weighted graph, and enables us to solve the FVS problem on weighted graphs in the same complexity as that for the problem on unweighted graphs. Note that this is significant because no previous algorithms for the FVS problem on unweighted graphs can be extended to weighted graphs mainly because of the lack of effective method for handling degree-2 vertices. Finally, the technique of dual parameters seems to be of general usefulness for the development of parameterized algorithms, and has been used more recently in solving other parameterized problems [23, 24].

The remaining part of this chapter is organized as follows. In Section B, we provide in full details a simpler algorithm and its analysis for unweighted graphs. This is done for clearer demonstration of our approach. We also indicate why this simpler algorithm does not work for weighted graphs. In Section C, we obtain the main result of the chapter , the algorithm for the weighted FVS problem. This generalization of the results from Section B is not straightforward and requires a number of new structures and techniques.

B. Feedback Vertex Set in Unweighted Graphs

In this section, we consider the FVS problem on unweighted graphs. We start with some terminologies. A *forest* is a graph that contains no cycles. A *tree* is a forest that is connected (therefore, a forest can be equivalently defined as a collection of disjoint trees). Let W be a subset of vertices in a graph $G = (V, E)$. We will denote by $G[W]$ the subgraph of G that is induced by the vertex set W. For simplicity we will use the notation $G - w$ and $G - W$ for respectively $G[V \setminus \{w\}]$ and $G[V \setminus W]$ where $w \in V$ and $W \subseteq V$. A pair (V_1, V_2) of vertex subsets in a graph $G = (V, E)$ is a *forest bipartition* of G if $V_1 \cup V_2 = V$, $V_1 \cap V_2 = \emptyset$, and both induced subgraphs $G[V_1]$ and $G[V_2]$ are forests. For a vertex $u \in V$ the degree of u will be the number of edges incident to u.

Let G be a graph and let F be a subset of vertices in G. The set F is a *feedback vertex set* (shortly, FVS) of G if $G - F$ is a forest. The *size* of an FVS F is the number of vertices in F.

Our main problem is formally defined as follows.

> FEEDBACK VERTEX SET: given a graph G and an integer k, either find an FVS of size at most k in G, or report that no such an FVS exists.

Before we present our algorithm for the FEEDBACK VERTEX SET problem, we first consider a special version of the problem, defined as follows:

> F-BIPARTITION FVS: given a graph G, a forest bipartition (V_1, V_2) of G, and an integer k, either find an FVS of size at most k for the graph G *in the subset* V_1, or report that no such an FVS exists.

Note that the main difference between the F-BIPARTITION FVS problem and the original FEEDBACK VERTEX SET problem is that we require that the FVS in the F-BIPARTITION FVS is contained in the given subset V_1.

Observe that certain structures in the input graph G can be easily processed and then removed from G. For example, if a vertex v has a self-loop (i.e., an edge whose both ends are incident to v), then the vertex v is necessarily contained in every FVS in G. Thus, we can directly include v in the objective FVS. If two vertices v and w are connected by multiple edges (i.e., there are more than one edge whose one end is v and the other end is w), then one of v and w must be contained in the objective FVS. Thus, we can branch into two recursive calls, one includes v, and the other includes w, in the objective FVS. All these operations are more efficient than the algorithm of running time $\mathcal{O}(5^k kn^2)$ developed in the current chapter. Therefore, for a given input graph G, we always first apply a preprocessing that applies the above operations and remove all self-loops and multiple edges in the graph G. In consequence, we can assume, without loss of generality, that the input graph G contains neither self-loops nor multiple edges.

The algorithm, **Feedback**(G, V_1, V_2, k), for the F-BIPARTITION FVS problem is given in Figure 5. We first discuss the correctness of the algorithm. The correctness of step 1 and step 2 of the algorithm is obvious. Now consider step 3. Let w be a vertex in V_1 that has at least two neighbors in V_2.

```
**Algorithm-1 Feedback**$(G, V_1, V_2, k)$
**Input:** $G = (V, E)$ is a graph with a forest bipartition $(V_1, V_2)$, $k$ is an
        integer.
**Output:** An FVS $F$ of $G$ such that $|F| \le k$ and $F \subseteq V_1$; or report "No"
        (i.e., no such an FVS exists).

1. if $(k < 0)$ or $(k = 0$ and $G$ is not a forest$)$ then return "No";
2. if $(k \ge 0)$ and $G$ is a forest then return $\emptyset$;
3. if a vertex $w$ in $V_1$ has at least two neighbors in $V_2$ then
3.1. if two neighbors of $w$ in $V_2$ is in the same tree in $G[V_2]$ then
        $F_1 = $ Feedback$(G - w, V_1 \setminus \{w\}, V_2, k - 1)$;
        if $F_1 = $ "No" then return "No"
        else return $F_1 \cup \{w\}$;
3.2. else
        $F_1 = $ Feedback$(G - w, V_1 \setminus \{w\}, V_2, k - 1)$;
        $F_2 = $ Feedback$(G, V_1 \setminus \{w\}, V_2 \cup \{w\}, k)$;
        if $F_1 \ne $ "No" then return $F_1 \cup \{w\}$
        else return $F_2$;
4. else pick any vertex $w$ that has degree $\le 1$ in $G[V_1]$;
4.1. if $w$ has degree $\le 1$ in the original graph $G$ then
        return Feedback$(G - w, V_1 \setminus \{w\}, V_2, k)$;
4.2. else return Feedback$(G, V_1 \setminus \{w\}, V_2 \cup \{w\}, k)$.
```

Fig. 5. Algorithm for the UNWEIGHTED FEEDBACK VERTEX SET problem.

If the vertex w has two neighbors in V_2 that belong to the same tree T in the induced subgraph $G[V_2]$, then the tree T plus the vertex w contains at least one cycle. Since our search for an FVS is restricted to V_1, the only way to break the cycles in $T \cup \{w\}$ is to include the vertex w in the objective FVS. Moreover, the objective FVS of size at most k exists in G if and only if the remaining graph $G - w$ has an FVS of size at most $k - 1$ in the subset $V_1 \setminus \{w\}$ (note that $(V_1 \setminus \{w\}, V_2)$ is a valid forest bipartition of the graph $G - w$). Therefore, step 3.1 correctly handles this case.

If no two neighbors of the vertex w belong to the same tree in the induced subgraph $G[V_2]$, then the vertex w is either in the objective FVS or not in the objective FVS. If w is in the objective FVS, then we should be able to find an FVS F_1 in the graph $G - w$ such that $|F_1| \leq k - 1$ and $F_1 \subseteq V_1 \setminus \{w\}$ (again note that $(V_1 \setminus \{w\}, V_2)$ is a valid forest bipartition of the graph $G - w$). On the other hand, if w is not in the objective FVS, then the objective FVS for G must be contained in the subset $V_1 \setminus \{w\}$. Also note that in this case, the subgraph $G[V_2 \cup \{w\}]$ induced by the subset $V_2 \cup \{w\}$ is still a forest since no two neighbors of w in V_2 belong to the same tree in $G[V_2]$. In consequence, $(V_1 \setminus \{w\}, V_2 \cup \{w\})$ still makes a valid forest bipartition for the graph G. Therefore, step 3.2 handles this case correctly.

Now we consider step 4. At this point, every vertex in V_1 has at most one neighbor in V_2. Moreover, since the induced subgraph $G[V_1]$ is a forest, there must be a vertex w in V_1 that has degree at most 1 in $G[V_1]$ (note that V_1 cannot be empty at this point since otherwise the algorithm would have stopped at step 2). If the vertex w also has degree at most 1 in the original graph G, then removing w does not help breaking any cycles in G. Therefore, the vertex w can be discarded. This case is correctly handled by step 4.1. Otherwise, the vertex w has degree at most 1 in the induced subgraph $G[V_1]$ but has degree larger than 1 in the original graph G. Observing that w has at most one neighbor in V_2, we can derive that the degree of w in the original graph G must be exactly 2. Moreover, w has exactly two neighbors u and v such that v is in V_1 and u is in V_2.

Since the vertex w has degree 2 in the original graph G, and the vertex v is adjacent to w, we have that every cycle in G that contains w has to contain v. In consequence, if w is contained in the objective FVS, then we can simply replace it by v. Therefore, in this case, we can safely assume that the vertex w is not in the objective FVS. This can be easily implemented by moving the vertex w from the set V_1 to the set V_2, and recursively working on the modified instance, as given in step

4.2 of the algorithm (note that $(V_1 \setminus \{w\}, V_2 \cup \{w\})$ is a valid forest bipartition of the graph G, because by our assumption, the vertex w will be a degree-1 vertex in the induced subgraph $G[V_2 \cup \{w\}]$).

Now we are ready to present the following lemma.

Lemma B.1 *The algorithm* **Feedback**(G, V_1, V_2, k) *correctly solves the* F-BIPARTITION FVS *problem. The running time of the algorithm is* $\mathcal{O}(2^{k+l}n^2)$, *where* n *is the number of vertices in* G, *and* l *is the number of connected components in the induced subgraph* $G[V_2]$.

PROOF. The correctness of the algorithm has been verified by the above discussion. Now we consider the complexity of the algorithm.

The recursive execution of the algorithm can be described as a search tree $\mathcal{T}$. We first count the number of leaves in the search tree $\mathcal{T}$. Note that only step 3.2 of the algorithm corresponds to branches in the search tree $\mathcal{T}$. Let $T(k, l)$ be the total number of leaves in the search tree $\mathcal{T}$ for the algorithm **Feedback**(G, V_1, V_2, k), where l is the number of connected components (i.e., trees) in the forest $G[V_2]$. Inductively, the number of leaves in the search tree $\mathcal{T}_1$ corresponding to the recursive call **Feedback**$(G - w, V_1 \setminus \{w\}, V_2, k - 1)$ is at most $T(k - 1, l)$. Moreover, we assumed at step 3.2 that w has at least two neighbors in V_2 and that no two neighbors of w in V_2 belong to the same tree in $G[V_2]$. Therefore, the vertex w "merges" at least two trees in $G[V_2]$ into a single tree in $G[V_2 \cup \{w\}]$. Hence, the number of trees in $G[V_2 \cup \{w\}]$ is at most $l - 1$. In consequence, the number of leaves in the search tree $\mathcal{T}_2$ corresponding to the recursive call **Feedback**$(G, V_1 \setminus \{w\}, V_2 \cup \{w\}, k)$ is at most $T(k, l - 1)$. This gives the following recurrence relation:

$$T(k, l) \leq T(k - 1, l) + T(k, l - 1).$$

Also note that none of the (non-branching) recursive calls in the algorithm (steps 3.1, 4.1, and 4.2) would increase the values k and l, and that $T(0, l) = 1$ for all l and $T(k, 0) = 1$ for all k (by steps 1-2). From all these facts, we can easily derive that $T(k, l) = \mathcal{O}(2^{k+l})$.

Finally, observe that along each root-leaf path in the search tree $\mathcal{T}$, the total number of executions of steps 1, 2, 3, 3.1, 4.1, and 4.2 of the algorithm is $\mathcal{O}(n)$ because each of these steps either stops immediately, or reduces the size of the set V_1 by at least 1 (and the size of V_1 is never increased during the execution of the algorithm). It remains to explain how each of the steps can be executed in $\mathcal{O}(n)$ time.

Before the first call to the **Feedback** algorithm, we use $\mathcal{O}(n^2)$ time, because this will happen only once. The three graphs, $G_1 = G[V_1], G_2 = G[V_2]$, and $G_{12} = (V, E \setminus (E(G_1) \cup E(G_2)))$ can be trivially constructed in $\mathcal{O}(n^2)$ time. G_1 and G_2 are forests, and G_{12} is a bipartite graph with the two vertex sets V_1 and V_2 as independent sets.

Steps 1, 2, 4.1, and 4.2 can be easily performed in $\mathcal{O}(n)$ time. For step 3, we simply search for a vertex of V_1 that has degree at least 2 in G_{12}, and for step 4 we search in G_1 for a leaf (vertex of degree at most 1). The condition for step 3.1 is that no two neighbors of w belong to the same tree in $G[V_2]$, which can be checked by simply marking each neighbor of w, and doing a search in the forest $G[V_2]$.

Each of the steps 3.1, 3.2, 4.1, and 4.2 changes one or more of the graphs G_1, G_2, G_{12}, and we have to argue that these manipulations can also be done in $\mathcal{O}(n)$ time. Looking closely at these steps, we can observe that only two operations are required. The first is to delete a vertex in V_1, which corresponds to deleting the vertex and all incident edges in G_1 and G_{12}. The second operation is to move a vertex w from V_1 to V_2, which corresponds to deleting w from G_1 and updating G_2 and G_{12}

as follows: add w to $V(G_2)$ and to V_2 of G_{12}, and read the set of edges incident to w in G, and add edges between w and vertices in V_2 to G_2 and between w and V_1 to G_{12}. Using double linked lists and pointers it is possible to delete a vertex and all incident edges in $\mathcal{O}(n)$ time, and to insert edges in $\mathcal{O}(1)$ time.

Therefore, the computation time along each root-leaf path in the search tree $\mathcal{T}$ is $\mathcal{O}(n^2)$. In conclusion, the running time of the algorithm **Feedback**(G, V_1, V_2, k) is $\mathcal{O}(2^{k+l}n^2)$. This completes the proof of the lemma. $\square$

Following the idea of *iterative compression* proposed by Reed et al. [90], we formulate the following problem:

> FVS REDUCTION: given a graph G and an FVS F of size $k+1$ for G, either construct an FVS of size at most k for G, or report that no such an FVS exists.

Lemma B.2 *The* FVS REDUCTION *problem on an n-vertex graph G can be solved in time $\mathcal{O}(5^k n^2)$.*

PROOF. We use the algorithm **Feedback** to solve the FVS REDUCTION problem. Let F be the FVS of size $k+1$ in the graph $G = (V, E)$. Every FVS F' of size at most k for G is a union of a subset F_1 of at most $k-j$ vertices in $V \setminus F$ and a subset F_2 of j vertices in F, for some integer j, $0 \le j \le k$. Note that since we assume that no vertex in $F \setminus F_2$ is in the FVS F', the induced subgraph $G[F \setminus F_2]$ must be a forest. For each j, $0 \le j \le k$, we enumerate all subsets of j vertices in F. For each such subset F_2 in F such that $G[F \setminus F_2]$ is a forest, we seek a subset F_1 of at most $k-j$ vertices in $V \setminus F$ such that $F_1 \cup F_2$ is an FVS in G.

Fix a subset F_2 in F, where $|F_2| = j$. Note that the graph G has an FVS $F_1 \cup F_2$ of size at most k, where $F_1 \subseteq V \setminus F$, if and only if the subset F_1 of $V \setminus F$ is an FVS for the graph $G - F_2$ and $|F_1| \le k - j$. Therefore, to solve the original problem, we construct an FVS F_1 for the graph $G - F_2$ such that $|F_1| \le k - j$ and $F_1 \subseteq V \setminus F$.

Since F is an FVS for G, we have that the induced subgraph $G[V \setminus F] = G - F$ is a forest. Moreover, by our assumption, the induced subgraph $G[F \setminus F_2]$ is also a forest. Note that $(V \setminus F) \cup (F \setminus F_2) = V \setminus F_2$, which is the vertex set for the graph $G' = G - F_2$. Therefore, $(V \setminus F, F \setminus F_2)$ is a forest bipartition of the graph G'. Thus, an FVS F_1 for the graph G' such that $|F_1| \le k - j$ and $F_1 \subseteq V \setminus F$ can be constructed by the algorithm **Feedback**$(G', V \setminus F, F \setminus F_2, k - j)$.

Since $|F| = k + 1$ and $|F_2| = j$, we have that $|F \setminus F_2| = k + 1 - j$. Therefore, the forest $G[F \setminus F_2]$ contains at most $k + 1 - j$ connected components. By Lemma B.1, the running time of the algorithm **Feedback**$(G', V \setminus F, F \setminus F_2, k - j)$ is $\mathcal{O}(2^{(k-j)+(k+1-j)}n^2) = \mathcal{O}(4^{k-j}n^2)$. Now for all integers j, $0 \le j \le k$, we enumerate all subsets F_2 of j vertices in F and apply the algorithm **Feedback**$(G', V \setminus F, F \setminus F_2, k - j)$ for those F_2 such that $G[F \setminus F_2]$ is a forest. As we discussed above, the graph G has an FVS of size at most k if and only if for some $F_2 \subseteq F$, the algorithm **Feedback**$(G', V \setminus F, F \setminus F_2, k - j)$ produces an FVS F_1 for the graph G'. The running time of this procedure is

$$\sum_{j=0}^{k} \binom{k+1}{j} \cdot \mathcal{O}(4^{k-j}n^2) = \sum_{j=0}^{k} \binom{k+1}{k-j+1} \mathcal{O}(4^{k-j+1}n^2) = \mathcal{O}(5^k n^2).$$

This completes the proof of the lemma. □

Finally, by combining Lemma E.1 with iterative compression, we obtain the main result of this section.

Theorem B.3 *The* FEEDBACK VERTEX SET *problem on an n-vertex graph is solvable in time $\mathcal{O}(5^k kn^2)$.*

PROOF. To solve the FEEDBACK VERTEX SET problem, for a given graph $G = (V, E)$, we start by applying Bafna et al.'s 2-approximation algorithm for the MINIMUM FEEDBACK VERTEX SET problem [5]. This algorithm runs in $\mathcal{O}(n^2)$ time, and either returns an FVS F' of size at most $2k$, or verifies that no FVS of size at most k exists. If no FVS is returned, the algorithm is terminated with the conclusion that no FVS of size at most k exists. In the case of the opposite result, we use any subset V' of k vertices in F', and put $V_0 = V' \cup (V \setminus F')$. Of course, the induced subgraph $G[V_0]$ has an FVS of size k, namely the set V' ($G[V_0 \setminus V']$ is a forest). Let $F' \setminus V' = \{v_1, v_2, \ldots, v_{|F'|-k}\}$, and let $V_i = V_0 \cup \{v_1, \ldots, v_i\}$ for $i \in \{0, 1, \ldots, |F'| - k\}$. Inductively, suppose that we have constructed an FVS F_i for the graph $G[V_i]$, where $|F_i| = k$. Then the set $F'_{i+1} = F_i \cup \{v_{i+1}\}$ is obviously an FVS for the graph $G[V_{i+1}]$ and $|F'_{i+1}| = k + 1$.

Now the pair $(G[V_{i+1}], F'_{i+1})$ is an instance for the FVS REDUCTION problem. Therefore, in time $\mathcal{O}(5^k n^2)$, we can either construct an FVS F_{i+1} of size k for the graph $G[V_{i+1}]$, or report that no such an FVS exists. Note that if the graph $G[V_{i+1}]$ does not have an FVS of size k, then the original graph G cannot have an FVS of size k. In this case, we simply stop and claim the non-existence of an FVS of size k for the original graph G. On the other hand, with an FVS F_{i+1} of size k for the graph $G[V_{i+1}]$, our induction proceeds to the next graph $G[V_{i+1}]$, until we reach the graph $G = G[V_{|F'|-k}]$. This process runs in time $\mathcal{O}(5^k k n^2)$ since $|F'| - k \leq k$, and solves the FEEDBACK VERTEX SET problem. $\square$

C. Feedback Vertex Set in Weighted Graphs

In this section, we discuss the FEEDBACK VERTEX SET problem on weighted graphs. A weighted graph $G = (V, E)$ is an undirected graph, where each vertex $u \in V$ is assigned a *weight* that is a positive real number. The weight of a vertex set $A \subseteq V$ is the sum of the vertex weights of all vertices in A. We denote by $|A|$ the cardinality of A. The (parameterized) FEEDBACK VERTEX SET problem on weighted graphs is formally defined as follows:

> WEIGHTED-FVS: given a weighted graph G and an integer k, either find an FVS F of minimum weight for G such that $|F| \leq k$, or report that no FVS of size at most k exists in G.

The algorithm for the weighted case has several similarities with the unweighted case, but has also a significant difference. The difference is that step 4.2 of **Algorithm-1** becomes invalid for weighted graphs. A degree-2 vertex w in the set V_1 cannot simply be excluded from the objective FVS. If the weight of w is smaller than that of its parent v in $G[V_1]$, it may become necessary to include w instead of v in the objective FVS.

To overcome this difficulty, we introduce a new partition structure of the vertices in a weighted graph.

Definition A triple (V_0, V_1, V_2) is an *independent-forest partition* (*IF-partition*) of a graph $G = (V, E)$ if (V_0, V_1, V_2) is a partitioning of V (i.e., $V_0 \cup V_1 \cup V_2 = V$, and V_0, V_1, and V_2 are pairwise disjoint), such that

(1) $G[V_1]$ and $G[V_2]$ are forests;

(2) $G[V_0]$ is an independent set;

(3) Every vertex u in V_0 is of degree 2 in G, with both neighbors in V_2.

The following problem is the analogue of the F-BIPARTITION FVS problem on weighted graphs.

> WEIGHTED IF-PARTITION FVS: given a weighted graph G, an IF-partition (V_0, V_1, V_2) of G, and an integer k, either find an FVS F of minimum weight for G that satisfies the conditions $|F| \leq k$ and $F \subseteq V_0 \cup V_1$, or report that no such an FVS exists.

To develop and analyze our algorithm for the WEIGHTED IF-PARTITION FVS problem, we need the following concept of *measure* for the problem instances. For a vertex subset V' in the graph G, we will denote by $\#c(V')$ the number of connected components in the induced subgraph $G[V']$.

Definition Let (G, V_0, V_1, V_2, k) be an instance of the WEIGHTED IF-PARTITION FVS problem with an IF-partition (V_0, V_1, V_2). The *deficiency* of the instance (G, V_0, V_1, V_2, k) is defined as

$$\tau(k, V_0, V_1, V_2) = k - (|V_0| - \#c(V_2) + 1),$$

Intuitively, $\tau(k, V_0, V_1, V_2)$ of the instance (G, V_0, V_1, V_2, k) is an upper bound on the number of vertices in the objective FVS that are in the set V_1 (this will become clearer during our discussion below). Our algorithm for the WEIGHTED IF-PARTITION FVS problem is based on the following observation: once we have correctly determined all vertices in the objective FVS that are in the set V_1, the problem will become solvable in polynomial time, as shown in the following lemma.

Lemma C.1 *Let* (G, V_0, V_1, V_2, k) *be an instance of the* WEIGHTED IF-PARTITION FVS *problem with an IF-partition* (V_0, V_1, V_2) *of an* n*-vertex graph* G*. If* $V_1 = \emptyset$*, or* $V_2 = \emptyset$*, or* $\tau(k, V_0, V_1, V_2) \leq 0$*, then a solution to the instance* (G, V_0, V_1, V_2, k) *can be constructed in time* $\mathcal{O}(n^2)$*.*

PROOF. First of all, note that if $k < 0$, then the solution to the instance is "No": we cannot remove a negative number of vertices from G. Thus, in the following discussion, we assume that $k \geq 0$.

If $V_2 = \emptyset$, then by the definition, V_0 should also be an empty set. Thus, the graph $G = G[V_1]$ is a forest, and the solution to the instance (G, V_0, V_1, V_2, k) is the empty set $\emptyset$.

Now consider the case $V_1 = \emptyset$. Then we need to find a minimum-weight subset of at most k vertices in the set V_0 whose removal from the graph $G = G[V_0 \cup V_2]$ results in a forest.

Construct a new graph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, where each vertex μ in $\mathcal{V}$ corresponds to a connected component in the induced subgraph $G[V_2]$, and each edge $[\mu, \nu]$ in $\mathcal{E}$ corresponds to a vertex v in the set V_0 such that the two neighbors of v are in the connected components in $G[V_2]$ that correspond to the two vertices μ and ν, respectively, in $\mathcal{H}$. Intuitively, the graph $\mathcal{H}$ can be obtained from the graph $G = G[V_0 \cup V_2]$ by "shrinking" each connected component in $G[V_2]$ into a single vertex and "smoothing" each degree-2 vertex in V_0 (note that the graph $\mathcal{H}$ may contain multiple edges and self-loops). Moreover, we give each edge in $\mathcal{H}$ a weight that is equal to the weight of the corresponding vertex in V_0. Thus, the graph $\mathcal{H}$ is a graph with edge weights. Observe that there is a one-to-one correspondence between the connected components in the graph $\mathcal{H}$ and the connected components in the graph

G. Moreover, since each connected component in the induced subgraph $G[V_2]$ is a tree, a connected component in the graph $\mathcal{H}$ is a tree if and only if the corresponding connected component in the graph G is a tree. Most importantly, removing a vertex in V_0 in the graph G corresponds to removing the corresponding edge in the graph $\mathcal{H}$. Therefore, the problem of constructing a minimum-weight vertex set in V_0 whose removal from G results in a forest is equivalent to the problem of constructing a minimum-weight edge set in the graph $\mathcal{H}$ whose removal from $\mathcal{H}$ results in a forest.

Let $\mathcal{H}_1$, …, $\mathcal{H}_s$ be the connected components of the graph $\mathcal{H}$, where for each i, the component $\mathcal{H}_i$ has n_i vertices and m_i edges. An edge set $\mathcal{E}_i$ in $\mathcal{H}_i$ whose removal from $\mathcal{H}_i$ results in a forest is of the minimum weight if and only if the complement graph $\mathcal{H}_i - \mathcal{E}_i$ is a spanning tree of the maximum weight in $\mathcal{H}_i$. Thus, the union $\mathcal{E}' = \bigcup_{i=1}^{s}(\mathcal{H}_i - \mathcal{T}_i)$ is a minimum-weight edge set whose removal from $\mathcal{H}$ results in a forest, where for each i, $\mathcal{T}_i$ is a maximum-weight spanning tree in the graph $\mathcal{H}_i$. Since the maximum-weight spanning tree $\mathcal{T}_i$ in $\mathcal{H}_i$ can be constructed in time $\mathcal{O}(n_i^2)$ by modifying the well-known minimum spanning tree algorithms (the algorithms work even for graphs with self-loops and multiple edges) [28], we conclude that the minimum-weight edge set $\mathcal{E}'$ in $\mathcal{H}$ can be constructed in time $\sum_{i=1}^{s}\mathcal{O}(n_i^2) = \mathcal{O}(n^2)$. Also note that the number of edges in the set $\mathcal{E}'$ is equal to $\sum_{i=1}^{s}(m_i - n_i + 1) = |\mathcal{E}| - |\mathcal{V}| + s$.

Correspondingly, in case $V_1 = \emptyset$, each minimum-weight FVS in V_0 for the graph G contains exactly $|\mathcal{E}| - |\mathcal{V}| + s$ vertices, and such an FVS can be constructed in time $\mathcal{O}(n^2)$. Note that $|\mathcal{E}| = |V_0|$, $|\mathcal{V}|$ is equal to the number $\#c(V_2)$ of connected components in the induced subgraph $G[V_2]$, and s (which is the number of connected components in $\mathcal{H}$) is equal to the number $\#c(G)$ of connected components in the graph $G = G[V_0 \cup V_2]$. Thus, each minimum-weight FVS in V_0 for the graph G contains exactly $|V_0| - \#c(V_2) + \#c(G)$ vertices, and such a minimum-weight FVS can be constructed in time $\mathcal{O}(n^2)$. Therefore, for the given instance (G, V_0, V_1, V_2, k) of the WEIGHTED IF-PARTITION FVS problem with $V_1 = \emptyset$, the solution is "No" if $k < |V_0| - \#c(V_2) + \#c(G)$; and the solution is an $\mathcal{O}(n^2)$ time constructible FVS of $|V_0| - \#c(V_2) + \#c(G)$ vertices in V_0 if $k \geq |V_0| - \#c(V_2) + \#c(G)$.

This completes the proof that when $V_1 = \emptyset$, a solution to the instance (G, V_0, V_1, V_2, k) can be constructed in time $\mathcal{O}(n^2)$.

Now consider the case $\tau(k, V_0, V_1, V_2) \leq 0$. If $V_2 = \emptyset$, then by the first part of the proof, the lemma holds. Thus, we assume that $V_2 \neq \emptyset$. As analyzed above, to break every cycle in the induced subgraph $G[V_0 \cup V_2]$ we have to remove at least $|V_0| - \#c(V_2) + \#c(V_0 \cup V_2)$ vertices in the set V_0. Therefore, if $\tau(k, V_0, V_1, V_2) \leq 0$, then $k \leq |V_0| - \#c(V_2) + 1 \leq |V_0| - \#c(V_2) + \#c(V_0 \cup V_2)$ (note that $V_2 \neq \emptyset$ so $\#c(V_0 \cup V_2) \geq 1$). Thus, in this case, all k vertices in the objective FVS must be in the set V_0 in order to break all cycles in the induced subgraph $G[V_0 \cup V_2]$, and no vertex in the objective FVS can be in the set V_1. Hence, if the induced subgraph $G[V_1 \cup V_2]$ contains a cycle, then the solution to the instance is "No". On the other hand, suppose that $G[V_1 \cup V_2]$ is a forest, then the graph G has another IF-partition (V_0', V_1', V_2'), where $V_0' = V_0$, $V_1' = \emptyset$, and $V_2' = V_1 \cup V_2$. It is easy to verify that in this case the instance (G, V_0', V_1', V_2', k) with the IF-partition (V_0', V_1', V_2') has the same solution set as the instance (G, V_0, V_1, V_2, k) with the IF-partition (V_0, V_1, V_2). Since $V_1' = \emptyset$, by the second part of the proof, a solution to the instance (G, V_0', V_1', V_2', k) with the IF-partition (V_0', V_1', V_2') can be constructed in time $\mathcal{O}(n^2)$. This completes the proof of the lemma. □

We are now in a position to introduce our main algorithm, which is given in Figure 6 and solves the WEIGHTED IF-PARTITION FVS problem. The subroutine **min-w**(S_1, S_2) on two vertex subsets S_1 and S_2 in the algorithm returns among S_1 and S_2 the one with a smaller weight (or any one of them if the weights are tied). To simplify our descriptions, we take the conventions that “No” is a special vertex set of an infinitely large weight and that any set plus “No” gives a “No”. Therefore, the value of **min-w**(S_1, S_2) will be (1) “No” if both S_1 and S_2 are “No”; (2) S_1 if S_2 is “No”; (3) S_2 if S_1 is “No”; and (4) the one of smaller weight among S_1 and S_2 if both S_1 and S_2 are not “No”.

For each tree in the forest $G[V_1]$, we fix a root so that we can talk about the “lowest leaf” in a tree in $G[V_1]$.

Lemma C.2 *The algorithm* **W-Feedback**(G, V_0, V_1, V_2, k) *correctly solves the* WEIGHTED IF-BIPARTITION FVS *problem, and its running time is* $\mathcal{O}(2^{\tau(k,V_0,V_1,V_2)}n^2)$, *where* n *is the number of vertices in the graph* G.

PROOF. We first verify the correctness of the algorithm. Step 1 of the algorithm is justified by Lemma C.1. Justifications for steps 2, 3, 4, 4.1, and 4.2 are exactly the same as that for steps 1, 4.1, 3, 3.1, and 3.2 in **Algorithm-1** for unweighted graphs. Now consider step 5. When the algorithm reaches step 5, the following conditions hold:

(1) the sets V_1 and V_2 are not empty;

(2) every vertex in the set V_1 has degree at least 2 in the graph G; and

(3) every vertex in the set V_1 has at most one neighbor in the set V_2.

Condition (1) holds because of step 1; condition (2) holds because of step 3; and condition (3) holds because of step 4.

By condition (1) and because the induced subgraph $G[V_1]$ is a forest, step 5 can always pick the vertex w_1. By conditions (2) and (3), the vertex w_1 has a unique neighbor in V_2. Also by conditions (2) and (3), the vertex w_1 must have a parent w in the tree T in $G[V_1]$. In consequence, the vertex w_1 has degree exactly 2 in G. Finally, since w_1 is the lowest leaf in the tree T, all children w_1, …, w_t of w in the tree T are also leaves in T. By conditions (2) and (3) again, each child w_i of w has a unique neighbor in the set V_2, and every child w_i of w has degree exactly 2 in the graph G.

Step 5.1 simply branches on the vertex w. To include the vertex w in the objective FVS, we simply remove w from the graph G (and from the set V_1), and recursively look for an FVS in $V_0 \cup (V_1 \setminus \{w\})$ of size at most $k-1$. Note that in this case, the sets V_0 and V_2 are unchanged, and the triple $(V_0, V_1 \setminus \{w\}, V_2)$ obviously makes a valid IF-partition for the graph $G - w$. On the other hand, to exclude the vertex w from the objective FVS, we move w from V_1 to V_2. First note that since the vertex w has at most one neighbor in V_2, the induced subgraph $G[V_2 \cup \{w\}]$ is still a forest. Moreover, since all children w_1, …, w_t of w have degree 2 in the graph G and each w_i has a unique neighbor in the set V_2, after moving w from V_1 to V_2, all these degree-2 vertices w_1, …, w_t have their both neighbors in the set $V_2 \cup \{w\}$. Therefore, these vertices w_1, …, w_t now can be moved to the set V_0. In particular, the triple $(V_0 \cup \{w_1, \ldots, w_t\}, V_1 \setminus \{w, w_1, \ldots, w_t\}, V_2 \cup \{w\})$ is a valid IF-partition of the vertex set of the graph G. This recursive branching is implemented by the two recursive calls in step 5.1.

If we reach step 5.2, then the two conditions in step 5.1 do not hold. Therefore, in addition to conditions (1)-(3), the following two conditions also hold:

(4) the vertex w has no neighbor in V_2; and

(5) the vertex w has a unique child w_1 in the tree T.

By conditions (2), (4), and (5), the vertex w has degree exactly 2 in the graph G (and w is not the root of the tree T). Therefore, the vertices w_1 and w are two adjacent degree-2 vertices in the graph G. Observe that in this case, a cycle in the graph G contains the vertex w_1 *if and only if* it also contains the vertex w. Therefore, we can safely assume that the one of larger weight among w_1 and w is not in the objective FVS. If the larger weight vertex is w_1, then the first recursive call in step 5.2 is executed, which moves w_1 from set V_1 to set V_2 (note that the triple $(V_0, V_1 \setminus \{w_1\}, V_2 \cup \{w_1\})$ is a valid IF-partition of G because w_1 has a unique neighbor in V_2). If the larger weight vertex is w, then the second recursive call in step 5.2 is executed, which moves w from V_1 to V_2. Note that since both neighbors of the degree-2 vertex w_1 are in the set $V_2 \cup \{w\}$, after adding w to the set V_2, we can also move the vertex w_1 from V_1 to V_0. Thus, the triple $(V_0 \cup \{w_1\}, V_1 \setminus \{w, w_1\}, V_2 \cup \{w\})$ is a valid IF-partition of the graph G.

We also remark that by our assumption, the input graph G contains neither multiple edges nor self-loops. Moreover, the graph in each of the recursive calls in the algorithm is either the original G, or G with a vertex deleted. Therefore, the graph in each of the recursive calls in the algorithm also contains neither multiple edges nor self-loops.

Since all possible cases are covered in the algorithm, we conclude that when the algorithm **W-Feedback** stops, it must output a correct solution to the given instance (G, V_0, V_1, V_2, k).

To analyze the running time, as in the unweighted case, we first count the number of leaves in the search tree corresponding to the execution of the algorithm. Let $T(k, V_0, V_1, V_2)$ be the number of leaves in the search tree for algorithm **W-Feedback**(G, V_0, V_1, V_2, k). We prove by induction on the value $\tau(k, V_0, V_1, V_2)$ that $T(k, V_0, V_1, V_2) \leq max(1, 2^{\tau(k,V_0,V_1,V_2)})$. First of all, if $\tau(k, V_0, V_1, V_2) \leq 0$, then by step

1 of the algorithm, we have $T(k, V_0, V_1, V_2) = 1$.

First consider the branching steps, i.e., step 4.2 and step 5.1. In case step 4.2 of the algorithm is executed, we have recursively

$$\begin{aligned}&T(k, V_0, V_1, V_2)\\ &\leq T(k-1, V_0, V_1 \setminus \{w\}, V_2) + T(k, V_0, V_1 \setminus \{w\}, V_2 \cup \{w\}).\end{aligned} \tag{4.1}$$

Since

$$\begin{aligned}\tau_1 &= \tau(k-1, V_0, V_1 \setminus \{w\}, V_2)\\ &= (k-1) - (|V_0| - \#c(V_2) + 1)\\ &= \tau(k, V_0, V_1, V_2) - 1\\ &< \tau(k, V_0, V_1, V_2),\end{aligned}$$

and

$$\begin{aligned}\tau_2 &= \tau(k, V_0, V_1 \setminus \{w\}, V_2 \cup \{w\})\\ &= k - (|V_0| - \#c(V_2 \cup \{w\}) + 1)\\ &\leq k - (|V_0| - (\#c(V_2) - 1) + 1)\\ &= \tau(k, V_0, V_1, V_2) - 1\\ &< \tau(k, V_0, V_1, V_2),\end{aligned}$$

where we have used the fact $\#c(V_2 \cup \{w\}) \leq \#c(V_2) - 1$ because in this case, we assume that the vertex w has two neighbors in two different trees in $G[V_2]$, therefore, adding w to V_2 merges at least two connected components in $G[V_2]$ and reduces the number of connected components by at least 1.

Therefore, by the inductive hypothesis, $T(k-1, V_0, V_1 \setminus \{w\}, V_2) \leq 2^{\tau_1}$, and $T(k, V_0, V_1 \setminus \{w\}, V_2 \cup \{w\}) \leq 2^{\tau_2}$. Combining these with Inequality (4.1), we get

$$\begin{aligned} T(k, V_0, V_1, V_2) &\leq T(k-1, V_0, V_1 \setminus \{w\}, V_2) + T(k, V_0, V_1 \setminus \{w\}, V_2 \cup \{w\}) \\ &\leq 2^{\tau_1} + 2^{\tau_2} \\ &\leq 2^{\tau(k, V_0, V_1, V_2)-1} + 2^{\tau(k, V_0, V_1, V_2)-1} \\ &= 2^{\tau(k, V_0, V_1, V_2)} \end{aligned}$$

In conclusion, the induction goes through for step 4.2 of the algorithm.

Now we consider step 5.1, which is the least trivial case, and makes the major difference from the unweighted cases. Let $V_0' = V_0 \cup \{w_1, \ldots, w_t\}$, $V_1' = V_1 \setminus \{w, w_1, \ldots, w_t\}$, and $V_2' = V_2 \cup \{w\}$. The execution of step 5.1 gives the following inequality:

$$T(k, V_0, V_1, V_2) \leq T(k-1, V_0, V_1 \setminus \{w\}, V_2) + T(k, V_0', V_1', V_2').$$

As we have shown above, by the inductive hypothesis, we have

$$T(k-1, V_0, V_1 \setminus \{w\}, V_2) \leq 2^{\tau(k, V_0, V_1, V_2)-1}. \tag{4.2}$$

To estimate the value $T(k, V_0', V_1', V_2')$, first note that $|V_0'| = |V_0| + t$. Moreover, at this point, we must have either that the vertex w has a neighbor in V_2 or that the vertex w has more than one child in the tree T in $G[V_1]$.

If w has a neighbor in V_2, then adding w to V_2 will “attach” the vertex w to a connected component in $G[V_2]$. In consequence, the number of connected components in $G[V_2]$ will be equal to that in $G[V_2 \cup \{w\}]$ (recall that w has only one neighbor in

V_2). In this case (note that $t \geq 1$), we have

$$\begin{aligned}\tau(k, V_0', V_1', V_2') &= k - (|V_0'| - \#c(V_2') + 1) \\ &= k - ((|V_0| + t) - \#c(V_2) + 1) \\ &\leq \tau(k, V_0, V_1, V_2) - 1\end{aligned}$$

Now let us assume that the vertex w has no neighbor in V_2 but has more than one child in the tree T in $G[V_1]$ (i.e., $t \geq 2$). Then $|V_0'| = |V_0| + t \geq |V_0| + 2$. In this case, adding the vertex w to the set V_2 increases the number of connected components in $G[V_2]$ by 1 (since w has no neighbor in V_2, the vertex w will become a single-vertex connected component in the induced subgraph $G[V_2 \cup \{w\}]$). That is, $\#c(V_2 \cup \{w\}) = \#c(V_2) + 1$. Therefore,

$$\begin{aligned}\tau(k, V_0', V_1', V_2') &= k - (|V_0'| - \#c(V_2') + 1) \\ &= k - ((|V_0| + t) - \#c(V_2 \cup \{w\}) + 1) \\ &\leq k - ((|V_0| + 2) - (\#c(V_2) + 1) + 1) \\ &\leq \tau(k, V_0, V_1, V_2) - 1\end{aligned}$$

In conclusion, in all cases in step 5.1, we will have $\tau(k, V_0', V_1', V_2') \leq \tau(k, V_0, V_1, V_2) - 1$. Therefore, now we can apply the induction and get

$$T(k, V_0', V_1', V_2') \leq 2^{\tau(k, V_0', V_1', V_2')} \leq 2^{\tau(k, V_0, V_1, V_2) - 1}$$

Combining this with the inequalities (4.1) and (4.2), we conclude that

$$T(k, V_0, V_1, V_2) \leq 2^{\tau(k, V_0, V_1, V_2)}$$

holds for the case of step 5.1.

We should also remark that it can be verified that for all non-branching recursive calls in the algorithm, i.e., steps 3, 4.1, and 5.2, the instance deficiency is never increased. In particular, if the first recursive call in step 5.2 is executed, then since the vertex w_1 has a unique neighbor in V_2, $\#c(V_2) = \#c(V_2 \cup \{w_1\})$. Thus,

$$\begin{aligned} \tau(k, V_0, V_1 \setminus \{w_1\}, V_2 \cup \{w_1\}) &= k - (|V_0| - \#c(V_2 \cup \{w_1\}) + 1) \\ &= k - (|V_0| - \#c(V_2) + 1) \\ &= \tau(k, V_0, V_1, V_2). \end{aligned}$$

If the second recursive call in step 5.2 is executed, then

$$\#c(V_2 \cup \{w\}) = \#c(V_2) + 1$$

because w has no neighbor in V_2 and w will become a single-vertex connected component in the induced subgraph $G[V_2 \cup \{w\}]$. Therefore,

$$\begin{aligned} & \tau(k, V_0 \cup \{w_1\}, V_1 \setminus \{w, w_1\}, V_2 \cup \{w\}) \\ = \; & k - (|V_0 \cup \{w_1\}| - \#c(V_2 \cup \{w\}) + 1) \\ = \; & k - ((|V_0| + 1) - (\#c(V_2) + 1) + 1) \\ = \; & \tau(k, V_0, V_1, V_2). \end{aligned}$$

Summarizing all the above discussions, we complete the inductive proof that the number of leaves in the search tree for the algorithm **W-Feedback**(G, V_0, V_1, V_2, k) is at most $2^{\tau(k, V_0, V_1, V_2)}$.

In the same way as in the proof for the unweighted case, we observe that along each root-leaf path in the search tree, the total number of executions of steps 1, 2, 3, 4, 4.1, 4.2, 5, 5.1, and 5.2 of the algorithm is $\mathcal{O}(n)$ because each of these steps either stops immediately, or reduces the size of the set V_1 by at least 1. Step 1 is only preformed in leaf nodes of the tree, and thus only adds $\mathcal{O}(n^2)$ time to the total. By similar arguments as the one used for the unweighted case, all steps except step 1 can be preformed in $\mathcal{O}(n)$ time. Therefore, the running time of the algorithm **W-Feedback**(G, V_0, V_1, V_2, k) is $\mathcal{O}(2^{\tau(k,V_0,V_1,V_2)}n^2)$. □

With Lemma C.2, we can now proceed in the same way as for the unweighted case to solve the original WEIGHTED-FVS problem. Consider the following weighted version of the FVS REDUCTION problem.

> WEIGHTED FVS REDUCTION: given a weighted graph G and an FVS F of size $k + 1$ for G, either construct an FVS F' of minimum weight that satisfies $|F'| \leq k$, or report that no such an FVS exists.

Note that in the definition of WEIGHTED FVS REDUCTION, we do not require that the given FVS F of size $k + 1$ have the minimum weight.

Lemma C.3 *The* WEIGHTED FVS REDUCTION *problem on an* n*-vertex graph is solvable in time* $\mathcal{O}(5^k n^2)$.

PROOF. The proof proceeds similarly to the proof of Lemma E.1. For the given FVS F of size $k + 1$ in the graph $G = (V, E)$, every FVS F' of size at most k for G (including the one with the minimum weight) is a union of a subset F_1 of at most $k - j$ vertices in $V \setminus F$ and a subset F_2 of j vertices in F, for some integer j, $0 \leq j \leq k$, where $(V \setminus F, F \setminus F_2)$ is a forest bipartition of the graph $G_0 = G - F_2$. Therefore, we can enumerate all subsets F_2 of j vertices in F, for each j, $0 \leq j \leq k$, such that $(V \setminus F, F \setminus F_2)$ is a forest bipartition of the graph $G_0 = G - F_2$, and construct the minimum-weight FVS F_0 of G_0 satisfying $|F_0| \leq k - j$. Note that the forest bipartition $(V \setminus F, F \setminus F_2)$ of G_0 is in fact a special IF-partition (V_0, V_1, V_2) of G_0, where $V_0 = \emptyset$,

$V_1 = V \setminus F$, and $V_2 = F \setminus F_2$. Therefore, by Lemma C.2, a minimum-weight FVS F_0 of G_0 satisfying $|F_0| \leq k - j$ can be constructed in time

$$\mathcal{O}(2^{\tau(k-j,V_0,V_1,V_2)}n^2) = \mathcal{O}(2^{(k-j)-(0-\#c(F\setminus F_2)+1)}n^2) = \mathcal{O}(4^{k-j}n^2),$$

where we have used the fact $\#c(F \setminus F_2) \leq |F \setminus F_2| = k+1-j$. Now the proof proceeds exactly the same way as that in Lemma E.1, and concludes that the WEIGHTED FVS REDUCTION problem can be solved in time $\mathcal{O}(5^k n^2)$. $\square$

Using Theorem B.3 and Lemma C.3, we obtain the main result of this chapter.

Theorem C.4 *The* WEIGHTED-FVS *problem on an n-vertex graph is solvable in time $\mathcal{O}(5^k kn^2)$.*

PROOF. Let (G, k) be a given instance of the WEIGHTED-FVS problem. As we explained in the proof of Theorem B.3, we can first construct, in time $\mathcal{O}(5^k kn^2)$, an FVS F of size $k+1$ for the graph G (the weight of F is not necessarily the minimum). Then we simply apply Lemma C.3. $\square$

```
**Algorithm-2 W-Feedback**$(G, V_0, V_1, V_2, k)$
**Input:** $G = (V, E)$ is a graph with an IF-partition $(V_0, V_1, V_2)$, $k$ is an
        integer.
**Output:** a minimum-weight FVS $F$ of $G$ such that $|F| \leq k$ and
        $F \subseteq V_0 \cup V_1$; or report "No" (i.e., no such an FVS exists).

1 if $(V_1 = \emptyset)$ or $(V_2 = \emptyset)$ or $(\tau(k, V_0, V_1, V_2) \leq 0)$ then
    solve the problem in time $\mathcal{O}(n^2)$;
2 if $(k < 0)$ or ($k = 0$ and $G$ is not a forest) then return "No";
3 if a vertex $w$ in $V_1$ has degree less than 2 in $G$ then
    return W-Feedback$(G - w, V_0, V_1 \setminus \{w\}, V_2, k)$;
4 else
    if a vertex $w$ in $V_1$ has at least two neighbors in $V_2$ then
4.1     if two neighbors of $w$ are in the same tree of $G[V_2]$ then
            return $(\{w\} \cup$ W-Feedback$(G - w, V_0, V_1 \setminus \{w\}, V_2, k - 1))$;
4.2     else
            $F_1 =$ W-Feedback$(G - w, V_0, V_1 \setminus \{w\}, V_2, k - 1)$;
            $F_2 =$ W-Feedback$(G, V_0, V_1 \setminus \{w\}, V_2 \cup \{w\}, k)$;
            return min-w$(F_1 \cup \{w\}, F_2)$;
5 else pick a lowest leaf $w_1$ in any tree $T$ in $G[V_1]$; let $w$ be the parent of
      $w_1$ in $T$, and let $w_1$, ..., $w_t$ be the children of $w$ in $T$;
5.1 if ($w$ has a neighbor in $V_2$) or ($w$ has more than one child in $T$) then
        $F_1 =$ W-Feedback$(G - w, V_0, V_1 \setminus \{w\}, V_2, k - 1)$;
        $F_2 =$ W-Feedback$(G, V_0 \cup \{w_1, \ldots, w_t\}, V_1 \setminus \{w, w_1, \ldots, w_t\},$
          $V_2 \cup \{w\}, k)$;
          return min-w$(F_1 \cup \{w\}, F_2)$;
5.2 else
        if the weight of $w_1$ is larger than the weight of $w$ then
          return W-Feedback$(G, V_0, V_1 \setminus \{w_1\}, V_2 \cup \{w_1\}, k)$;
        else return W-Feedback$(G, V_0 \cup \{w_1\}, V_1 \setminus \{w, w_1\}, V_2 \cup \{w\}, k)$.
```

Fig. 6. Algorithm for the WEIGHTED FEEDBACK VERTEX SET problem

CHAPTER V

DIRECTED FEEDBACK VERTEX SET*

In this chapter, we present the first fpt-algorithm of running time $O^*(4^k k!)$ for the FEEDBACK VERTEX SET problem on directed graphs. It had been an well-known open problem whether the FEEDBACK VERTEX SET problem on directed graphs is fixed-parameter tractable or not for 16 years.

Our algorithm transforms the FEEDBACK VERTEX SET problem into $O(k!)$ SKEW SEPARATOR problems, then solves each SKEW SEPARATOR problem in time of $O^*(4^k)$. Our algorithm for the SKEW SEPARATOR problem takes both the size of the skew separator to search and the minimum cut from the last source to all sinks as measures. The last measure is critical, because it avoids the need of bounding the length of cycles in graphs. In the rest of this chapter, we give detailed analysis of the algorithm for the FEEDBACK VERTEX SET problem on directed graphs.

A. Introduction

Let G be a directed graph. A *feedback vertex set* F (briefly, FVS) for G is a set of vertices in G such that every directed cycle in G contains at least one vertex in F, or equivalently, that the removal of F from the graph G leaves a directed acyclic graph (i.e., a DAG). The (parameterized) FEEDBACK VERTEX SET problem on directed graphs (briefly, the DFVS problem) is defined as follows: given a directed graph G and a parameter k, either construct an FVS of at most k vertices for G or report that no such set exists.

---

*Reprinted with permission from "A Fixed-Parameter Algorithm for the Directed Feedback Vertex Set Problem", by J. Chen, Y. Liu, S. Liu, B. O'Sullivan and I. Razgon, 2008, Journal of the ACM, Volume 55, No. 5, Article 21, Copyright [2008] by ACM.

The DFVS problem is a classic NP-complete problem that appeared in the first list of NP-complete problems in Karp's seminal paper [68], and has a variety of applications in areas such as operating systems [94], database systems [52], and circuit testing [76]. In particular, the DFVS problem has played an essential role in the study of *deadlock recovery* in database systems and in operating systems [94, 52]. In such a system, the status of system resource allocations can be represented as a directed graph G (i.e., the *system resource-allocation graph*), and a directed cycle in G represents a deadlock in the system. Therefore, in order to recover from deadlocks, we need to abort a set of processes in the system, i.e., to remove a set of vertices in the graph G, so that all directed cycles in G are broken. Equivalently, we need to find an FVS in the graph G. In practice, one may expect and desire that the number of vertices removed from the graph G, which is the number of processes to be aborted in the system, be small. This motivates the study of *parameterized algorithms* for the DFVS problem that find an FVS of k vertices in a directed graph of n vertices and run in time $f(k)n^{O(1)}$ for a fixed function f; thus, the algorithms become practically efficient when the value k is small.

This work has been part of a systematic study of the *theory of fixed-parameter tractability* [37], which has received considerable attention in recent years. A problem Q is a *parameterized problem* if each instance of Q contains a specific integral parameter k. A parameterized problem is *fixed-parameter tractable* if it can be solved in time $f(k)n^c$ for a function $f(k)$ and a constant c, where the function $f(k)$ is independent of the instance size n. A large number of NP-hard parameterized problems, such as the VERTEX COVER problem [20] and the ML TYPE-CHECKING problem [77], have been shown to be fixed-parameter tractable. On the other hand, strong evidence has been given that another group of well-known parameterized problems, including the INDEPENDENT SET problem and the DOMINATING SET problem, are not fixed-parameter tractable [37]. The study of fixed-parameter tractability of parameterized problems has become increasingly interesting, for both theoretical research and practical computation.

The fixed-parameter tractability of the DFVS problem was posted as an open problem in the very first papers on the study of fixed-parameter tractability [35, 34]. After numerous significant efforts, however, the problem still remained open. In the past fifteen years, the problem has been constantly and explicitly posted as an open problem in a large number of publications in the literature (see [58] for a recent survey on this study). The problem has become a well-known and outstanding open problem in parameterized computation and complexity.

In this chapter, we develop new algorithmic techniques that lead to the conclusion that the DFVS problem is fixed-parameter tractable, and thus resolve the above open problem in parameterized computation and complexity. We first show that the DFVS problem can be reduced in time $f(k)n^{O(1)}$ for some function f to a special version of the multi-cut problem, which will be called the SKEW SEPARATOR problem. We then develop an algorithm that shows the fixed-parameter tractability of the SKEW SEPARATOR problem. The combination of these two results gives an algorithm with running time $4^k k! n^{O(1)}$ for the DFVS problem, which proves its fixed-parameter tractability.

The relationship between the DFVS problem and multi-cut problems has been studied in the research of approximation algorithms for the FEEDBACK VERTEX SET problem [41, 75]. However, our problem formulations and the corresponding techniques are significantly different from those studied in the approximation algorithms. In particular, our formulations and techniques seem especially suitable for developing faster and more effective exact algorithms (of exponential-time) for NP-hard multi-cut problems. First of all, instead of seeking a multi-cut that separates a given set of *terminal vertices*, as formulated in most multi-cut problems, our problem is more general: we wish to construct a multi-cut that separates a *collection of terminal vertex-subsets.* This more general version of the multi-cut problem enables us to effectively reduce the search space size when we are searching for an *optimal* solution of a given problem instance. Secondly, unlike most multi-cut problems whose solutions are multi-cuts that are in general symmetric to the given terminal vertices, the multi-cuts for the SKEW SEPARATOR problem are asymmetric to the terminal vertex-subsets. Thirdly, we develop an (exponential-time) reduction that effectively reduces the problem of multi-cuts for multiple terminal vertex-subsets to the problem of minimum cuts from a single source vertex to a single sink vertex. Note that the latter is solvable in polynomial time via algorithms for the MAXIMUM FLOW problem. Such an exponential-time reduction is obviously very different from the polynomial-time processes used in the development of polynomial-time approximation algorithms. Finally, unlike most parameterized algorithms that are focused on effectively decreasing the sole parameter value k, our algorithm for the SKEW SEPARATOR problem adds another dimension of bounds in terms of the size of a minimum cut between two properly chosen terminal vertex-subsets. This dimension of bounds has become crucial in our development of the algorithm for the SKEW SEPARATOR problem because it effectively bounds the number of branches in which the parameter value k is not decreased.

Before we move to the technical discussion of our algorithms, we remark that the FEEDBACK VERTEX SET problem on undirected graphs (briefly, the UFVS problem) has also been an interesting and active research topic in parameterized computation and complexity. Since the first fixed-parameter tractable algorithm for the UFVS problem was published fifteen years ago [9], there has been an impressive list of improved algorithms for the problem. Currently the best algorithm for the UFVS problem runs in time $O(5^k kn^2)$ [23]. The FEEDBACK VERTEX SET problem on directed graphs (i.e., the DFVS problem) seems very different from the problem on undirected graphs (i.e., the UFVS problem). This fact has also been reflected in the study of approximation algorithms for the problems. The FEEDBACK VERTEX SET problem on undirected graphs is polynomial-time approximable with a ratio 2. This holds true even for weighted graphs [5]. On the other hand, it still remains open whether the FEEDBACK VERTEX SET problem on directed graphs has a constant-ratio polynomial-time approximation for the problem on directed graphs has a ratio $O(\log \tau \log \log \tau)$, where τ is the size of a minimum FVS for the input graph [41].

B. Preliminaries

Let $G = (V, E)$ be a directed graph and let $e = [u, v]$ be a (directed) edge in G. We say that the edge e *goes out* from the vertex u and *comes into* the vertex v. The edge e is called an *outgoing edge* of the vertex u, and an *incoming edge* of the vertex v. These concepts can be extended from single vertices to general vertex sets. Thus, for two vertex sets S_1 and S_2, we can say that an edge goes out from S_1 and comes into S_2 if the edge goes out from a vertex in S_1 and comes into a vertex in S_2. Moreover, we say that an edge *goes out from* S_1 if the edge goes out from a vertex in S_1 and comes into a vertex not in S_1, and that an edge *comes into* S_2 if the edge goes out from a vertex not in S_2 and comes into a vertex in S_2.

A *path* P from a vertex v_1 to a vertex v_h in the graph G is a sequence $\{v_1, v_2, \ldots, v_h\}$ of vertices in G such that $[v_i, v_{i+1}]$ is an edge in G for all $1 \leq i \leq h-1$. The path P is *simple* if no vertex is repeated in P. The path P is a *cycle* if $v_1 = v_h$, and the cycle is *simple* if no other vertices are repeated. We say that a path is from a vertex set S_1 to a vertex set S_2 if the path is from a vertex in S_1 to a vertex in S_2. The graph G

is a *DAG* (i.e., directed acyclic graph) if it contains no cycles.

For a vertex subset $V' \subseteq V$ in the directed graph $G = (V, E)$, we denote by $G[V']$ the subgraph of G that is induced by the vertex subset V'. Without any ambiguity, we will denote by $G - V'$ the induced subgraph $G[V - V']$, and by $G - w$, where w is a vertex in G, the induced subgraph $G[V - \{w\}]$.

A vertex subset F in the directed graph G is a *feedback vertex set* (*FVS*) if the graph $G - F$ is a DAG. Since a vertex v with a self-loop (i.e., an edge that both goes out from and comes into v) must be included in every FVS for the graph G, we will assume, without loss of generality, that the graphs in our discussion have no self-loops.

Definition Let $[S_1, \ldots, S_l]$ and $[T_1, \ldots, T_l]$ be two collections of l vertex subsets in a directed graph $G = (V, E)$. A *skew separator* X for $([S_1, \ldots, S_l], [T_1, \ldots, T_l])$ is a vertex subset in $V - \bigcup_{i=1}^{l}(S_i \cup T_i)$ such that for any pair of indices i and j satisfying $l \geq i \geq j \geq 1$, there is no path from S_i to T_j in the graph $G - X$.

The subsets $S_1, \ldots, S_l$ will be called the *source sets* and the subsets $T_1, \ldots, T_l$ will be called the *sink sets*. A vertex is a *non-terminal vertex* if it is not in $\bigcup_{i=1}^{l}(S_i \cup T_i)$. Note that by definition, all vertices in a skew separator must be non-terminal vertices. Moreover, a skew separator X is asymmetric to the source sets and the sink sets: a path from S_i to T_j with $i < j$ may exist in the graph $G - X$.

When there is only one source set S_1 and one sink set T_1, a skew separator for the pair $([S_1], [T_1])$ becomes a regular cut for S_1 and T_1, i.e., a vertex set whose removal leaves a graph in which there is no path from the set S_1 to the set T_1. Therefore, a skew separator for the pair $([S_1], [T_1])$ is also called a *cut from S_1 to T_1*. A cut from S_1 to T_1 is a *min-cut* (i.e., a minimum cut) if it has the smallest cardinality over all cuts from S_1 to T_1.

The following lemma can be easily derived based on standard maximum flow techniques [93]. Thus, we omit its proof.

Lemma B.1 *There is an $O(kn^2)$ time algorithm that for two given vertex subsets S and T in a directed graph G of n vertices, and a parameter k, either constructs a min-cut from S to T whose size is bounded by k, or reports that the min-cut from S to T has a size larger than k.*

The algorithm for the DFVS problem is obtained through careful development of algorithms for a series of problems. In the following, we give the formal definitions of these problems.

> SKEW SEPARATOR: given $(G, [S_1, \ldots, S_l], [T_1, \ldots, T_l], k)$, where G is a directed graph, $[S_1, \ldots, S_l]$ is a collection of l source sets and $[T_1, \ldots, T_l]$ is a collection of l sink sets in G, and a parameter k, such that
>
> 1. all sets S_1, …, S_l, T_1, …, T_l are pairwise disjoint;
> 2. for each i, $1 \leq i \leq l-1$, there is no edge coming into the set S_i; and
> 3. for each j, $1 \leq j \leq l$, there is no edge going out from the set T_j,
>
> either construct a skew separator of at most k vertices for the pair $([S_1, \ldots, S_l], [T_1, \ldots, T_l])$, or report that no such separator exists.

Note that in an instance of the SKEW SEPARATOR problem, condition (2) on source sets and condition (3) on sink sets are not completely symmetric. Although the first $l-1$ source sets are not allowed to have incoming edges, the last source set S_l is allowed to have incoming edges. On the other hand, *all* sink sets are not allowed to have outgoing edges.

We remark that conditions (1)-(3) in the definition of the SKEW SEPARATOR problem (plus the restriction that the skew separator can consist of only non-terminal vertices) may be relaxed, and our techniques for the problem may still be applicable. However, the above formulation of the problem will make our discussion simpler, and will also be sufficient for our solution to the DFVS problem, which is the focus of the current chapter. We leave the investigation of the separator problems of more general forms to later research.

Let $G = (V, E)$ be a directed graph, and let (D_1, D_2) be a bi-partition of the vertex set V of G, i.e., $D_1 \cup D_2 = V$ and $D_1 \cap D_2 = \emptyset$. The bi-partition (D_1, D_2) is a *DAG-bipartition* for the graph G if both induced subgraphs $G[D_1]$ and $G[D_2]$ are DAGs. A vertex subset F in the graph G is a D_1-FVS if F is an FVS for G and $F \subseteq D_1$.

> DAG-BIPARTITION FVS: given (G, D_1, D_2, k), where G is a directed graph, (D_1, D_2) is a DAG-bipartition for G, and k is the parameter, either construct a D_1-FVS of size bounded by k for the graph G, or report that no such D_1-FVS exists.

We will be also interested in a special version of the FEEDBACK VERTEX SET problem.

> DFVS REDUCTION: given a triple (G, F, k), where G is a directed graph and F is an FVS of size $k + 1$ for G, either construct an FVS of size bounded by k for G, or report that no such FVS exists.

Finally, our central problem in this chapter is as follows.

> DFVS: given a pair (G, k), where G is a directed graph and k is the parameter, either construct an FVS of size bounded by k for G, or report that no such FVS exists.

C. Solving the SKEW SEPARATOR Problem

In this section, we study the complexity of the SKEW SEPARATOR problem.

Let $(G, [S_1, \ldots, S_l], [T_1, \ldots, T_l], k)$ be an instance of the SKEW SEPARATOR problem. Define $T_{all} = \bigcup_{1 \leq i \leq l} T_i$. There are a few cases in which we can directly reduce the instance size:

Rule R1 There is no path from S_l to T_{all}, i.e., the size of a min-cut from S_l to T_{all} is 0: then we only need to find a skew separator of size k that separates S_i from T_j for all indices i and j satisfying $l-1 \geq i \geq j \geq 1$, i.e., we can work instead on the instance $(G, [S_1, \ldots, S_{l-1}], [T_1, \ldots, T_{l-1}], k)$. Note that in this case, by definition, if $l = 1$, then the solution to the instance $(G, [S_1, \ldots, S_{l-1}], [T_1, \ldots, T_{l-1}], k)$ is simply the empty set $\emptyset$;

Rule R2 There is an edge from S_l to T_{all}: then there is no way to even separate S_l from T_{all} – we can simply stop and claim that the given instance is a "No" instance;

Rule R3 There exists a non-terminal vertex w, an edge from S_l to w, and an edge from w to T_{all}: then the vertex w must be included in the skew separator in order to separate S_l and T_{all} – we can simply work on the instance $(G - w, [S_1, \ldots, S_l], [T_1, \ldots, T_l], k - 1)$ and recursively find a skew separator of size $k - 1$.

Note in Rules R1 and R3, the reduced instances $(G, [S_1, \ldots, S_{l-1}], [T_1, \ldots, T_{l-1}], k)$ and $(G - w, [S_1, \ldots, S_l], [T_1, \ldots, T_l], k - 1)$ are still valid instances of the SKEW SEPARATOR problem.

In the following, we assume that for the instance $(G, [S_1, \ldots, S_l], [T_1, \ldots, T_l], k)$, none of the rules above is applicable. In particular, since Rule R1 is not applicable, a min-cut from S_l to T_{all} has size larger than 0. Because Rules R1-R3 are not applicable, there must be a non-terminal vertex u_0 such that (1) there is an edge from S_l to u_0; and (2) there is no edge from u_0 to T_{all}. Such a vertex u_0 will be called an S_l-*extended vertex*. Fix an S_l-extended vertex u_0, let $S_l' = S_l \cup \{u_0\}$.

We start with the following simple but important lemma. The proof of this lemma is straightforward. Thus, we leave it to the reader.

Lemma C.1 *Let X be a subset of vertices in the graph G that does not contain the S_l-extended vertex u_0. Then X is a skew separator for $([S_1, \ldots, S_l], [T_1, \ldots, T_l])$ if and only if X is a skew separator for $([S_1, \ldots, S_{l-1}, S_l'], [T_1, \ldots, T_{l-1}, T_l])$.*

Lemma C.1 also directly implies the following two useful corollaries.

Corollary C.2 *A skew separator for $([S_1, \ldots, S_{l-1}, S_l'], [T_1, \ldots, T_{l-1}, T_l])$ is also a skew separator for $([S_1, \ldots, S_l], [T_1, \ldots, T_l])$.*

Corollary C.3 *The size of a min-cut from S_l' to T_{all} in the graph G is at least as large as the size of a min-cut from S_l to T_{all} in G.*

Now we are ready for our main theorem in this section.

Theorem C.4 *If the size of a min-cut from S_l to T_{all} is equal to the size of a min-cut from S_l' to T_{all}, then the pair $([S_1, \ldots, S_l], [T_1, \ldots, T_l])$ has a skew separator of size bounded by k if and only if the pair $([S_1, \ldots, S_{l-1}, S_l'], [T_1, \ldots, T_{l-1}, T_l])$ has a skew separator of size bounded by k.*

Proof. $\Leftarrow$: Suppose that the pair $([S_1, \ldots, S_{l-1}, S_l'], [T_1, \ldots, T_{l-1}, T_l])$ has a skew separator X' of size bounded by k. By Corollary C.2, X' is also a skew separator for $([S_1, \ldots, S_l], [T_1, \ldots, T_l])$. In consequence, $([S_1, \ldots, S_l], [T_1, \ldots, T_l])$ has a skew separator of size bounded by k.

$\Rightarrow$: Suppose that the pair $([S_1, \ldots, S_l], [T_1, \ldots, T_l])$ has a skew separator X of size bounded by k. If the skew separator X does not contain the S_l-extended vertex u_0, then by Lemma C.1, X is also a skew separator of size bounded by k for the pair $([S_1, \ldots, S_{l-1}, S_l'], [T_1, \ldots, T_{l-1}, T_l])$, and the theorem is proved. Therefore, we can assume that the set X contains the S_l-extended vertex u_0. We will define another set X' that does not contain u_0. We will show that $|X'| \leq |X|$ and that X' is a skew separator for $([S_1, \ldots, S_l], [T_1, \ldots, T_l])$. Then the theorem will immediately follow.

Let Y be a min-cut from S_l' to T_{all}. Then Y does not contain the S_l-extended vertex u_0. Moreover, since there is no edge coming into S_i from outside of S_i for all $i \leq l - 1$, the set Y does not contain any vertex in $\bigcup_{i=1}^{l-1} S_i$. In consequence, the set Y consists of only non-terminal vertices. By Corollary C.2, Y is also a cut from S_l to T_{all}. Moreover, by the assumption of the theorem that the size of a min-cut from S_l to T_{all} is equal to the size of a min-cut from S_l' to T_{all}, Y is actually also a min-cut from S_l to T_{all}. Let $R_Y(S_l)$ be the set of vertices v such that either $v \in S_l$ or there is a path from S_l to v in the subgraph $G - Y$. In particular, $u_0 \in R_Y(S_l)$ because Y does not contain u_0 and there is an edge from S_l to u_0.

We introduce a number of sets as follows.

$$\begin{aligned} Z &= X \cap Y; \\ X_{in} &= X \cap R_Y(S_l); \\ X_{out} &= X - (X_{in} \cup Z). \end{aligned}$$

That is, the skew separator X for $([S_1, \ldots, S_l], [T_1, \ldots, T_l])$ is decomposed into three disjoint subsets Z, X_{in}, and X_{out} (note that by definitions, $R_Y(S_l)$ and Y do not

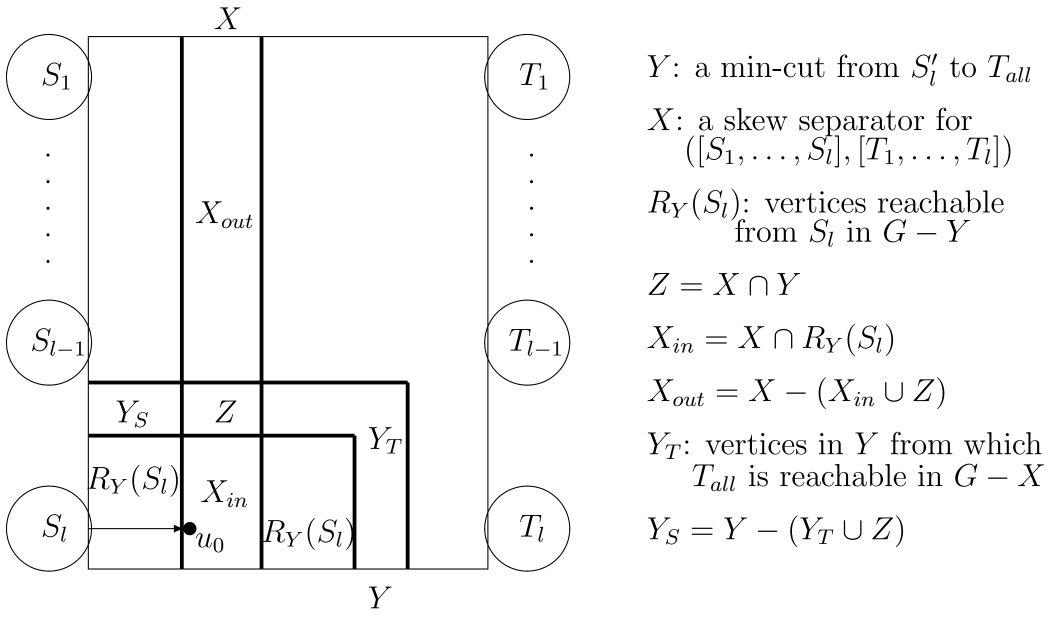

Fig. 7. Sets in the proof of Theorem C.4.

intersect).

Let Y_T be the set of vertices v in the min-cut Y such that there is a path from v to T_{all} in the subgraph $G - X$. By definition, we have $Y_T \cap Z = \emptyset$. Let

$$Y_S = Y - (Y_T \cup Z).$$

Thus, the min-cut Y from S_l to T_{all} is decomposed into three disjoint subsets Z, Y_T, and Y_S. Figure 7 gives an intuitive illustration of the sets Z, X_{in}, X_{out}, Y_T, Y_S, and $R_Y(S_l)$.

We first show that the set $Y' = Y_S \cup Z \cup X_{in}$ is also a cut from S_l to T_{all}. If by contradiction Y' is not a cut from S_l to T_{all}, then there is a path P_1 from S_l to T_{all} in the subgraph $G - Y'$. The path P_1 must contain vertices in the set Y since Y is a cut from S_l to T_{all}. Let w be the first vertex on the path P_1 that is in Y

when we traverse from S_l to T_{all} along the path P_1. Then w must be in Y_T since Y' contains both Y_S and Z. Now the partial path P_1' of P_1 from S_l to w (not including w) must be entirely contained in $R_Y(S_l)$ (note that the path P_1 does not intersect $Y_S \cup Z$). Moreover, the path P_1' contains neither vertices in $X_{in} \cup Z$ (by the definition of the set Y') nor vertices in X_{out} (since the sets X_{out} and $R_Y(S_l)$ are disjoint). In summary, the subpath P_1' from S_l to w contains no vertex in the set X. Moreover, by the definition of the set Y_T, and $w \in Y_T$, there is a path P_1'' from w to T_{all} in the subgraph $G - X$. Now the concatenation of the paths P_1' and P_1'' would result in a path from S_l to T_{all} in the graph $G - X$, contradicting the fact that X is a skew separator for the pair $([S_1, \ldots, S_l], [T_1, \ldots, T_l])$. This contradiction shows that the set Y' must be a cut from S_l to T_{all}.

Since Y is a min-cut from S_l to T_{all}, we have $|Y| \leq |Y'|$. By definition, $Y = Y_S \cup Z \cup Y_T$ and $Y' = Y_S \cup Z \cup X_{in}$. Also note that Y_S, Z, and Y_T are pairwise disjoint, and that Y_S, Z, and X_{in} are also pairwise disjoint. Therefore, we must have $|Y_T| \leq |X_{in}|$.

Consider the set $X' = X_{out} \cup Z \cup Y_T$. The set X' has the following properties: (1) X' consists of only non-terminal vertices (because both X and Y consist of only non-terminal vertices); (2) $|X'| \leq |X|$ (because $|Y_T| \leq |X_{in}|$), so the size of X' is bounded by k; and (3) the set X' does not contain the S_l-extended vertex u_0 (this is because u_0 is in X_{in} and Y does not contain u_0). Therefore, if we can prove that X' is a skew separator for the pair $([S_1, \ldots, S_l], [T_1, \ldots, T_l])$, then by Lemma C.1, X' is also a skew separator of size bounded by k for the pair $([S_1, \ldots, S_{l-1}, S_l'], [T_1, \ldots, T_{l-1}, T_l])$. This will complete the proof of the theorem.

Therefore, what remains is to prove that the set $X' = X_{out} \cup Z \cup Y_T$ is a skew separator for the pair $([S_1, \ldots, S_l], [T_1, \ldots, T_l])$. Let $R_Y(T_{all})$ be the set of vertices v such that either $v \in T_{all}$, or there is a path from v to T_{all} in the subgraph $G - Y$.

Suppose by contradiction that the set X' is not a skew separator for the pair $([S_1, \ldots, S_l], [T_1, \ldots, T_l])$. Then there is a path P_2 in the subgraph $G - X'$ from S_i to T_j for some $i \geq j$. The path P_2 has the following properties:

1. The path P_2 must contain a vertex in $R_Y(S_l)$: since X is a skew separator for the pair $([S_1, \ldots, S_l], [T_1, \ldots, T_l])$, the path P_2 from S_i to T_j with $i \geq j$ must contain at least one vertex w_1 in $X = X_{in} \cup Z \cup X_{out}$. Now since the path P_2 is in the subgraph $G - X'$, where $X' = X_{out} \cup Z \cup Y_T$, the vertex w_1 must be in X_{in}, which is a subset of $R_Y(S_l)$;

2. The path P_2 must contain a vertex in Y_S: by Property (1), P_2 contains a vertex w_1 in $R_Y(S_l)$. From the vertex w_1 to T_{all} along the path P_2, there must be a vertex w_2 in $Y = Y_S \cup Z \cup Y_T$ since Y is a cut from S_l to T_{all} while w_1 is reachable from S_l in the subgraph $G - Y$. Now since $X' = X_{out} \cup Z \cup Y_T$, and the path P_2 is in the subgraph $G - X'$, the vertex w_2 on the path P_2 must be in the set Y_S;

3. The path P_2 must end at a vertex in $R_Y(T_{all})$; this is simply because P_2 is ended in T_{all}. Note that by definition, no vertex in Y_S can be in $R_Y(T_{all})$.

By Properties (2)-(3), the path P_2 contains a vertex not in $R_Y(T_{all})$ and ends at a vertex in $R_Y(T_{all})$. Thus, there must be an internal vertex w in the path such that w is not in $R_Y(T_{all})$ but all vertices after w along the path P_2 (from S_i to T_j) are in $R_Y(T_{all})$. Note that no vertex w' after the vertex w along the path P_2 can be in the set X: w' in X would imply w' in X_{in} (since P_2 is a path in the subgraph $G - X'$), which would imply that there is another vertex after w' that is in Y thus is not in $R_Y(T_{all})$. Moreover, the vertex w must be in the set Y (otherwise, w would be in $R_Y(T_{all})$). Since P_2 is a path in $G - X'$ and $X' = X_{out} \cup Z \cup Y_T$, the vertex

w must be in the set Y_S. However, this derives a contradiction: the subpath of P_2 from w to T_{all} shows that the vertex w should belong to the set Y_T (note that all vertices after w on the path are not in X), and the sets Y_S and Y_T are disjoint. This contradiction proves that the set X' must be a skew separator for the pair $([S_1, \ldots, S_l], [T_1, \ldots, T_l])$. Since the size of the set X' is bounded by k and X' does not contain the S_l-extended vertex u_0, by Lemma C.1, the set X' is also a skew separator for the pair $([S_1, \ldots, S_{l-1}, S_l'], [T_1, \ldots, T_{l-1}, T_l])$, and the size of X' is bounded by k.

This completes the proof of the theorem. □

Theorem C.4 enables us to develop a parameterized algorithm for the SKEW SEPARATOR problem. The algorithm is presented in Figure 8.

Theorem C.5 *The algorithm* $\mathbf{SMC}(G, [S_1, \ldots, S_l], [T_1, \ldots, T_l), k]$ *solves the problem* SKEW SEPARATOR *in time* $O(4^k kn^3)$*, where* n *is the number of vertices in the input graph* G*.*

PROOF. For the correctness of the algorithm, let $(G, [S_1, \ldots, S_l], [T_1, \ldots, T_l], k)$ be an input to the algorithm, which is an instance of the SKEW SEPARATOR problem, where $G = (V, E)$ is a directed graph, $[S_1, \ldots, S_l]$ and $[T_1, \ldots, T_l]$ are the source sets and the sink sets, respectively, and k is the upper bound of the size of the skew separator we are looking for.

If $l = 1$, then the problem becomes the construction of a min-cut of size bounded by k from S_1 to T_1, which can be solved in $O(kn^2)$ time by Lemma B.1. Steps 2-4 were justified in the discussions of Rules 2, 1, 3, respectively, at the beginning of this section (note that we have also consistently defined that an instance is a "No" instance if the parameter k has a negative value). Therefore, if the algorithm reaches step 5, then none of the Rules 1-3 are applicable. In particular, since Rule 1 is not applicable and the sets S_l and T_{all} are disjoint, there must be an edge $[v, w]$, where $v \in S_l$ and $w \notin S_l$. Since Rule 2 is not applicable, the vertex w is not in the set T_{all}. The vertex w also cannot be in any source set S_i for $i < l$ because there is no edge coming into S_i from outside of S_i. Therefore, the vertex w is a non-terminal vertex. Finally, since Rule 3 is not applicable, there is no edge from w to T_{all}. Thus, w must be an S_l-extended vertex. This proves that at step 5, the algorithm can always find an S_l-extended vertex u_0.

In the case $m > k$ in step 7, i.e., the size m of a min-cut from S_l to T_{all} is larger than the parameter k, then even separating a single source set S_l from the sink sets $T_{all} = \bigcup_{j=1}^{l} T_j$ requires more than k vertices. Thus, no skew separator of size bounded by k can exist to separate S_i from T_j for all $l \geq i \geq j \geq 1$. Step 7 correctly handles this case by returning "No".

In the case $m = m'$ in step 9, i.e., the size m of a min-cut from S_l to T_{all} is equal to the size m' of a min-cut from S_l' to T_{all}, by Theorem C.4, the pair $([S_1, \ldots, S_l], [T_1, \ldots, T_l])$ has a skew separator of size bounded by k if and only if the pair $([S_1, \ldots, S_{l-1}, S_l'], [T_1, \ldots, T_{l-1}, T_l])$ has a skew separator of size bounded by k. Moreover, by Corollary C.2, a skew separator of size bounded by k for the pair $([S_1, \ldots, S_{l-1}, S_l'], [T_1, \ldots, T_{l-1}, T_l])$ is also a skew separator for $([S_1, \ldots, S_l], [T_1, \ldots, T_l])$. Therefore, in this case we can recursively call the algorithm **SMC**$(G, [S_1, \ldots, S_{l-1}, S_l'], [T_1, \ldots, T_{l-1}, T_l], k)$, and look instead for a skew separator of size bounded by k for the pair $([S_1, \ldots, S_{l-1}, S_l'], [T_1, \ldots, T_{l-1}, T_l])$, as handled by step 9.1.

In the case $m \neq m'$, then the algorithm branches into two subcases: step 9.2 includes the S_l-extended vertex u_0 in the skew separator and recursively looks for a skew separator of size bounded by $k - 1$ in the remaining graph $G - u_0$ for the pair $([S_1, \ldots, S_l], [T_1, \ldots, T_l])$; and step 9.3 excludes the S_l-extended vertex u_0 from the skew separator and recursively looks for a skew separator that does not contain u_0 and is of size bounded by k in the graph G for the pair $([S_1, \ldots, S_l], [T_1, \ldots, T_l])$, which is a skew separator of size bounded by k for the pair $([S_1, \ldots, S_{l-1}, S_l'], [T_1, \ldots, T_{l-1}, T_l])$ by Lemma C.1. This completes the verification of the correctness of the algorithm. Now we analyze its complexity.

The recursive execution of the algorithm can be described as a search tree $\mathcal{T}$. We first count the number of leaves in the search tree $\mathcal{T}$. Note that only steps 9.2-9.3 of the algorithm correspond to branches in the search tree $\mathcal{T}$. Let $D(k, m)$ be the total number of leaves in the search tree $\mathcal{T}$ for the algorithm $\mathbf{SMC}(G, [S_1, \ldots, S_l], [T_1, \ldots, T_l], k)$, where m is the size of a min-cut from S_l to T_{all}. Then steps 9.2-9.3 induce the following recurrence relation:

$$D(k, m) \leq D(k-1, m_1) + D(k, m_2) \tag{5.1}$$

where m_1 is the size of a min-cut from S_l to T_{all} in the graph $G - u_0$ as given in step 9.2, and m_2 is the size of a min-cut from S_l' to T_{all} in the graph G as given in step 9.3. Note that $m - 1 \leq m_1 \leq m$ because removing the vertex u_0 from the graph G cannot increase the size of a min-cut from S_l to T_{all}, and can decrease the size of a min-cut for the two sets by at most 1. Moreover, by Corollary C.3, in step 9.3 we must have $m_2 \geq m + 1$. Summarizing these, we have

$$m - 1 \leq m_1 \leq m \qquad \text{and} \qquad m_2 \geq m + 1. \tag{5.2}$$

We prove, by induction on $t = 2k - m$, that $D(k, m) \leq 2^{2k-m}$. First note that we always have $t = 2k - m \geq 0$ because by the definitions of k and m we always have $k \geq m \geq 0$. In particular, in the initial case when $t = 2k - m = 0$, we must have $k = m = 0$; in this case the algorithm can solve the instance without further branching. Therefore, we have $D(k, m) = 1$ when $t = 2k - m = 0$. For the inductive step, note that by Inequalities (5.2), we have

$$t_1 = 2(k-1) - m_1 \leq 2(k-1) - (m-1) = 2k - m - 1,$$

and

$$t_2 = 2k - m_2 \leq 2k - (m+1) = 2k - m - 1.$$

Therefore, we can apply the inductive hypothesis on Inequality (5.1), which gives

$$\begin{aligned} D(k,m) &\leq D(k-1, m_1) + D(k, m_2) \\ &\leq 2^{2(k-1)-m_1} + 2^{2k-m_2} \\ &\leq 2^{2k-m-1} + 2^{2k-m-1} \\ &= 2^{2k-m}. \end{aligned} \tag{5.3}$$

This completes the inductive proof. Moreover, we also note that certain non-branching steps (i.e., steps 3, 4, and 9.1) may also change the values of k and m, thus changing the value $t = 2k - m$. However, none of these steps increases the value $t = 2k - m$: (i) step 3 keeps the value k unchanged and does not decrease the value m (because in this case the size of a min-cut from S_l to T_{all} is 0 that cannot be larger than the size of a min-cut from S_{l-1} to $\bigcup_{j=1}^{l-1} T_j$); (ii) step 4 decreases the value k by 1 and the value m by at most 1 (because removing a vertex from G can reduce the size of a min-cut from S_l to T_{all} by at most 1), which as a total will decrease the value $t = 2k - m$ by at least 1; (iii) by the condition assumed, step 9.1 keeps both the values k and m unchanged, thus unchanging the value $t = 2k - m$. As a result, the value $t = 2k - m$ after a branching step to the next branching step can never be increased.

Summarizing the above discussion, we conclude that the total number of leaves, $D(k,m)$, in the search tree $\mathcal{T}$ for the algorithm $\mathbf{SMC}(G, [S_1, \ldots, S_l], [T_1, \ldots, T_l], k)$, where m is the size of a min-cut from S_l to T_{all}, satisfies the following inequality

$$D(k,m) \leq 2^{2k-m} \leq 4^k.$$

The running time of each execution of the algorithm **SMC**, not counting the time for the recursive calls in the execution, is bounded by $O(kn^2)$, where n is the number of vertices in the input graph. In particular, by Lemma B.1, step 1 that looks for a min-cut of size bounded by k from S_1 to T_1, steps 6-7 that determine if the size m of a min-cut from S_l to T_{all} is bounded by k, and steps 8-9 that determine if the size of a min-cut from S_l' to T_{all} is equal to m ($m \leq k$ at this point), all have their running time bounded by $O(kn^2)$.

Observe that for each recursive call in an execution of the algorithm **SMC**, either the number of source-sink pairs in the instance is decreased by 1 (step 3), or the number of non-terminal vertices in the instance is decreased by 1 (steps 4, 9.1, 9.2, and 9.3). When the number of source-sink pairs is equal to 1, the problem is solved in time $O(kn^2)$ by step 1, and when the number of non-terminal vertices is equal to 0, either step 2 or step 3 can be applied directly. In conclusion, along each root-leaf path in the search tree $\mathcal{T}$, there are at most $O(n)$ recursive calls to the algorithm **SMC**. Therefore, the running time along each root-leaf path in the search tree $\mathcal{T}$ is bounded by $O(kn^3)$.

Summarizing the above discussions, we conclude that the running time of the algorithm **SMC** is bounded by $O(4^k kn^3)$. This completes the proof of the theorem.

$\square$

D. Solving the DAG-BIPARTITION FVS Problem

In this section, we describe how to use the results in the previous section to solve the DAG-BIPARTITION FVS problem.

Recall that an instance of DAG-BIPARTITION FVS is given as a tuple (G, D_1, D_2, k), where G is a directed graph, (D_1, D_2) is a DAG-bipartition of G, and k is the parameter, with the objective of finding an FVS X for the graph G such that $X \subseteq D_1$ (recall that such an FVS is called a D_1-FVS) and that the size of X is bounded by k.

Let $\pi = \{v_1, v_2, \ldots, v_h\}$ be a topologically sorted order of the vertices in the induced DAG $G[D_2]$. We construct an instance of the SKEW SEPARATOR problem as follows:

1. Let G' be the graph obtained from G by removing all edges in $G[D_2]$.
2. In the graph G', replace each vertex v_i in D_2 by a pair (t_i, s_i) of vertices such that all incoming edges into v_i are now coming into the vertex t_i, and that all outgoing edges from v_i are now going out from the vertex s_i. Let the resulting graph be G_π.

Note that in the resulting graph G_π, the vertices s_i, $1 \le i \le h$, have no incoming edges, and the vertices t_j, $1 \le j \le h$, have no outgoing edges. Moreover, since we have removed all edges between the vertices in $G[D_2]$, every edge going out from a vertex s_i must come into a vertex in the set D_1, and every edge coming into a vertex t_j must go out from a vertex in the set D_1. In particular, $(G_\pi, [\{s_1\}, \ldots, \{s_h\}], [\{t_1\}, \ldots, \{t_h\}], k)$ is a valid instance for the SKEW SEPARATOR problem, which will be called an instance of the SKEW SEPARATOR *induced by the instance* (G, D_1, D_2, k) *of* DAG-BIPARTITION FVS *and the topologically sorted order* π *of the vertices in* $G[D_2]$.

Thus, each vertex v_i in the set D_2 in the graph G is now "split" into the two vertices s_i and t_i in the graph G_π. Moreover, there is a one-to-one mapping between the vertices in the set D_1 in the graph G and the non-terminal vertices in the graph G_π. Thus, in case of no ambiguity, we will use the same vertex name to refer to both a non-terminal vertex in the graph G_π and a vertex in the set D_1 in the graph G. In particular, a skew separator for the pair $([\{s_1\}, \ldots, \{s_h\}], [\{t_1\}, \ldots, \{t_h\}])$ in the graph G_π corresponds to a subset of D_1 in the graph G. We have the following important theorem.

Theorem D.1 *Let (G, D_1, D_2, k) be an instance of the* DAG-BIPARTITION FVS *problem, and let X be a D_1-FVS for the graph G. Then there is a topologically sorted order $\pi = \{v_1, \ldots, v_h\}$ of the vertices in $G[D_2]$ such that in the instance $(G_\pi, [\{s_1\}, \ldots, \{s_h\}], [\{t_1\}, \ldots, \{t_h\}], k)$ induced by (G, D_1, D_2, k) and π: (1) X is a skew separator for the pair $([\{s_1\}, \ldots, \{s_h\}], [\{t_1\}, \ldots, \{t_h\}])$ in the graph G_π; and (2) every skew separator for the pair $([\{s_1\}, \ldots, \{s_h\}], [\{t_1\}, \ldots, \{t_h\}])$ in G_π is a D_1-FVS for the graph G.*

PROOF. As assumed in the theorem, let (G, D_1, D_2, k) be an instance of the DAG-BIPARTITION FVS problem, and let X be a D_1-FVS for the graph G. Consider the subgraph $G - X$. Since X is an FVS for G, the graph $G - X$ is a DAG. Therefore, the vertices in $G - X$ can be topologically sorted into an ordered list π' such that there is no edge in $G - X$ that goes out from a later vertex in π' and comes into an earlier vertex in π'. Let $\pi = \{v_1, \ldots, v_h\}$ be the order of the vertices in D_2 that is induced from the order π' (i.e., π is obtained from π' by removing the vertices not in D_2. Note that all vertices in X are in D_1). The order π is obviously a topologically sorted order for the DAG $G[D_2]$. We show that this order π of the vertices in D_2 and the corresponding instance $(G_\pi, [\{s_1\}, \ldots, \{s_h\}], [\{t_1\}, \ldots, \{t_h\}], k)$ induced by (G, D_1, D_2, k) and π satisfy the conclusions of the theorem.

We first show that the set X is a skew separator for $([\{s_1\}, \ldots, \{s_h\}], [\{t_1\}, \ldots, \{t_h\}])$ in the graph G_π. If this were not the case, then there would be a path P in the graph $G_\pi - X$ that starts from a vertex s_i and ends at a vertex t_j with $i \geq j$. Since no vertex in $\{s_1, \ldots, s_h\}$ has incoming edges and no vertex in $\{t_1, \ldots, t_h\}$ has outgoing edges, all internal vertices on the path P are non-terminal vertices in G_π. In consequence, all internal vertices on P are vertices in the set D_1 in the graph G. Therefore, the path P in $G_\pi - X$ corresponds to a path P' in the graph $G - X$ that starts from the vertex v_i and ends at the vertex v_j, where $i \geq j$, with all internal vertices of P' in the set D_1. But it is impossible: (1) if $i = j$ then the path P' would be a cycle in the graph $G - X$, contradicting the assumption that X is an FVS for the graph G; and (2) if $i > j$, then P' would become a path from v_i to v_j with $i > j$ in the graph $G - X$, contradicting the assumption that $\pi = \{v_1, \ldots, v_h\}$ is an order of the vertices in D_2 that is induced from the topologically sorted order π' of the vertices in the DAG $G - X$. In conclusion, the path P does not exist, and the set X is a skew separator for the pair $([\{s_1\}, \ldots, \{s_h\}], [\{t_1\}, \ldots, \{t_h\}])$ in the graph G_π.

Now we prove that every skew separator X' for $([\{s_1\}, \ldots, \{s_h\}], [\{t_1\}, \ldots, \{t_h\}])$ in the graph G_π is a D_1-FVS for the graph G. First of all, by definition, a skew separator consists of only non-terminals, thus, all vertices in X' are in the set D_1. Suppose for a contradiction that X' is not a D_1-FVS for the graph G. Then there is a cycle C in the graph $G - X'$. Without loss of generality, we can assume that C is a simple cycle. Since both the induced subgraphs $G[D_1]$ and $G[D_2]$ are DAGs, the cycle C must contain both vertices in D_1 and vertices in D_2. We consider two different cases.

Case 1. The cycle C contains a single vertex v_i in the set D_2. Then all other vertices in the cycle C are in the set D_1. But then the cycle C would correspond to a path P_1 in the graph $G_\pi - X'$ that starts with the vertex s_i and ends at the vertex

t_i (with all internal vertices being non-terminal vertices). But this contradicts the assumption that X' is a skew separator for the pair $([\{s_1\}, \ldots, \{s_h\}], [\{t_1\}, \ldots, \{t_h\}])$ that should have cut all paths from s_i to t_i.

Case 2. The cycle C contains more than one vertex in D_2. Let $\{v_{i_1}, v_{i_2}, \ldots, v_{i_d}, v_{i_1}\}$ be the order of the vertices in D_2 that we encounter when traversing along the cycle C (starting from an arbitrary vertex v_{i_1} in D_2), where $d > 1$. Then there must be an index j such that $i_j > i_{j+1}$ (where we take $i_{j+1} = i_1$ if $j = d$). Now consider the subpath P_2 of C that starts from the vertex v_{i_j} and ends at the vertex $v_{i_{j+1}}$. The path P_2 cannot be a single edge from v_{i_j} to $v_{i_{j+1}}$ since $\pi = \{v_1, v_2, \ldots, v_h\}$ is a topologically sorted order for the vertices in the DAG $G[D_2]$ and $i_j > i_{j+1}$. Thus, the path P_2 contains at least one internal vertex. Since all internal vertices on the path P_2 are not in D_2 thus correspond to non-terminal vertices in the graph $G_\pi - X'$, the path P_2 would correspond to a path P_2' in the graph $G_\pi - X'$ that starts from the vertex s_{i_j} and ends at the vertex $t_{i_{j+1}}$, with $i_j > i_{j+1}$. Again this contradicts the assumption that X' is a skew separator for the pair $([\{s_1\}, \ldots, \{s_h\}], [\{t_1\}, \ldots, \{t_h\}])$, which should have cut all paths from s_{i_j} to $t_{i_{j+1}}$ when $i_j > i_{j+1}$.

This proves that the skew separator X' for $([\{s_1\}, \ldots, \{s_h\}], [\{t_1\}, \ldots, \{t_h\}])$ in the graph G_π must be a D_1-FVS for the graph G. This completes the proof of the theorem. □

Theorem D.1 enables us to reduce the DAG-BIPARTITION FVS problem to the SKEW SEPARATOR problem. An algorithm for the DAG-BIPARTITION FVS problem is given in Figure 9.

Theorem D.2 *The algorithm* **DBF**(G, D_1, D_2, k) *solves the* DAG-BIPARTITION FVS *problem in time* $O(4^k k n^3 h!)$, *where* h *is the number of vertices in the set* D_2, *and* n *is the number of vertices in the input graph* G.

Proof. The running time of the algorithm is obvious: the **for**-loop in step 1 is executed at most $h!$ times, and the time for each execution is dominated by the subroutine call to the algorithm **SMC** in step 1.2. By Theorem C.5, the running time of each execution of step 1.2 is bounded by $O(4^k kn^3)$.

For the correctness of the algorithm, first note that the algorithm always returns "No" unless it actually constructs a D_1-FVS of size bounded by k for G in step 1.3. In particular, if the input instance (G, D_1, D_2, k) contains no D_1-FVS of size bounded by k for the graph G, then the algorithm always correctly reports "No".

On the other hand, suppose that there is a D_1-FVS X_0 of size bounded by k for the graph G. Then by Theorem D.1, there is a topologically sorted order $\pi = \{v_1, \ldots, v_h\}$ of the vertices in $G[D_2]$ such that in the instance $(G_\pi, [\{s_1\}, \ldots, \{s_h\}], [\{t_1\}, \ldots, \{t_h\}], k)$ of the skew separator problem induced by (G, D_1, D_2, k) and π, X_0 is a skew separator for $([\{s_1\}, \ldots, \{s_h\}][\{t_1\}, \ldots, \{t_h\}])$ in the graph G_π, and every skew separator for $([\{s_1\}, \ldots, \{s_h\}], [\{t_1\}, \ldots, \{t_h\}])$ in G_π is a D_1-FVS for the graph G. In particular, $([\{s_1\}, \ldots, \{s_h\}], [\{t_1\}, \ldots, \{t_h\}])$ has at least one skew separator of size bounded by k (e.g., X_0) in the graph G_π. Therefore, step 1.2 of the algorithm **DBF** must return a skew separator X of size bounded by k for the pair $([\{s_1\}, \ldots, \{s_h\}], [\{t_1\}, \ldots, \{t_h\}])$ in the graph G_π (the set X may be different from the set X_0), and this set X is a D_1-FVS for the graph G. In conclusion, if there is a D_1-FVS of size bounded by k for the graph G, then the algorithm **DBF**(G, D_1, D_2, k) will correctly return a D_1-FVS of size bounded by k in step 1.3. □

E. Solving the DFVS Problem

We present our algorithm for the DFVS problem. We start with a more restricted version of the problem, the DFVS REDUCTION problem, defined as follows.

> DFVS REDUCTION: given a triple (G, F, k), where G is a directed graph and F is an FVS of size $k + 1$ for G, either construct an FVS of size bounded by k for G, or report that no such FVS exists.

Lemma E.1 *The* DFVS REDUCTION *problem on a triple* (G, F, k) *is solvable in time* $O(n^3 4^k k^3 k!)$, *where* n *is the number of vertices in the input graph* G.

PROOF. Let $G = (V, E)$ be the input directed graph with $n = |V|$ vertices, and let F be the input FVS of size $k+1$ for the graph G. Every FVS F' of size bounded by k for G can be split into two disjoint subsets F_1 and F_2, where F_2 consists of j vertices in F for some integer j, $0 \le j \le k$, and F_1 consists of at most $k-j$ vertices in $V-F$. Note that since we assume that no vertex in $F - F_2$ is in the FVS F', the induced subgraph $G[F - F_2]$ must be a DAG. Therefore, for each j, $0 \le j \le k$, we enumerate all subsets of j vertices in F. For each such subset F_2 of F such that $G[F - F_2]$ is a DAG, we seek a subset F_1 of at most $k-j$ vertices in $V - F$ such that $F_1 \cup F_2$ makes an FVS for the graph G.

Fix a subset F_2 of F, such that $|F_2| = j$ and that the induced subgraph $G[F-F_2]$ is a DAG. Note that the graph G has an FVS $F_1 \cup F_2$ of size bounded by k, where $F_1 \subseteq V - F$, if and only if the subset F_1 of $V - F$ is an FVS for the graph $G - F_2$ and the size of F_1 is bounded by $k - j$. Therefore, to solve the original problem, we can instead consider how to construct an FVS F_1 for the graph $G - F_2$ such that $|F_1| \le k - j$ and $F_1 \subseteq V - F$.

Since F is an FVS for G, we have that the induced subgraph $G[V - F] = G - F$ is a DAG. Moreover, by our assumption, the induced subgraph $G[F - F_2]$ is also a DAG. Note that $(V - F) \cup (F - F_2) = V - F_2$, which is the vertex set for the graph $G' = G - F_2$. Therefore, $(V - F, F - F_2)$ is a DAG-bipartition of the graph

G'. Thus, an FVS F_1 for the graph G' such that $|F_1| \leq k - j$ and $F_1 \subseteq V - F$, is actually a $(V - F)$-FVS of size bounded by $k - j$ for the graph G' with the DAG-bipartition $(V - F, F - F_2)$. Therefore, the set F_1 can be constructed by the algorithm $\mathbf{DBF}(G', V - F, F - F_2, k - j)$.

Since $|F| = k + 1$ and $|F_2| = j$, we have $|F - F_2| = k + 1 - j$. Therefore, the DAG $G[F - F_2]$ contains exactly $k + 1 - j$ vertices. By Theorem D.2, the running time of the algorithm $\mathbf{DBF}(G', V - F, F - F_2, k - j)$ is bounded by $O(4^{k-j}(k - j)n^3(k + 1 - j)!)$. Now for all integers j, $0 \leq j \leq k$, we enumerate all subsets F_2 of j vertices in F and apply the algorithm $\mathbf{DBF}(G', V - F, F - F_2, k - j)$ for those F_2 such that $G[F - F_2]$ is a DAG. As we discussed above, the graph G has an FVS of size bounded by k if and only if for some F_2 of j vertices in F, where $0 \leq j \leq k$, the algorithm $\mathbf{DBF}(G', V - F, F - F_2, k - j)$ produces an FVS F_1 of size bounded by $k - j$ for the graph G'. The running time of this process is bounded by the order of

$$\sum_{j=0}^{k} \binom{k+1}{j} \left(4^{k-j}(k - j)n^3(k + 1 - j)!\right) = O(n^3 4^k k^3 k!).$$

This completes the proof of the lemma. □

The rest of our process for solving the original DFVS problem is to apply the *iterative compression* method. The method was proposed by [90] and has been used for solving the FEEDBACK VERTEX SET problem on undirected graphs [33, 57]. Here we extend the method and apply it to solve the DFVS problem.

Theorem E.2 *The* DFVS *problem is solvable in time* $O(n^4 4^k k^3 k!)$.

PROOF. Let (G, k) be an instance of the DFVS problem, where $G = (V, E)$ is a directed graph with $n = |V|$ vertices, and k is the parameter. Pick any subset V_0 of $k + 1$ vertices in G, and let F_0 be any subset of k vertices in V_0. Note that the set F_0

is an FVS of k vertices for the induced subgraph $G_0 = G[V_0]$ since the graph $G_0 - F_0$ consists of a single vertex (note that by our assumption, the graph G contains no self-loops).

Let $V - V_0 = \{v_1, v_2, \ldots, v_{n-k-1}\}$. Let $V_i = V_0 \cup \{v_1, \ldots, v_i\}$, and let $G_i = G[V_i]$ be the subgraph induced by V_i, for $i = 0, 1, \ldots, n-k-1$. Inductively, suppose that for an integer i, $0 \leq i < n-k-1$, we have constructed an FVS F_i of size bounded by k for the induced subgraph G_i (this has been the case for $i = 0$). Without loss of generality, we can assume that the set F_i consists of exactly k vertices – otherwise we simply pick $k - |F_i|$ vertices (arbitrarily) from $G_i - F_i$ and add them to the set F_i. Now consider the set $F'_{i+1} = F_i + v_{i+1}$. Since $G_{i+1} - F'_{i+1} = G_i - F_i$ and F_i is an FVS for G_i, the set F'_{i+1} is an FVS of size $k+1$ for the induced subgraph G_{i+1}. In particular, the triple (G_{i+1}, F'_{i+1}, k) is a valid instance for the DFVS REDUCTION problem.

Apply Theorem E.1 to the instance (G_{i+1}, F'_{i+1}, k), which either returns an FVS F_{i+1} of size bounded by k for the graph G_{i+1}, or claims that no such FVS exists. It is easy to see that if the induced subgraph $G_{i+1} = G[V_{i+1}]$ does not have an FVS of size bounded by k, then the original graph G cannot have an FVS of size bounded by k. Therefore, in this case, we can simply stop and conclude that there is no FVS of size bounded by k for the original input graph G. On the other hand, suppose that an FVS F_{i+1} of size bounded by k is constructed for the graph G_{i+1} in the above process, then the induction successfully proceeds from i to $i+1$ with a new pair (G_{i+1}, F_{i+1}).

In conclusion, the above process either stops at some point and correctly reports that the input graph G has no FVS of size bounded by k, or eventually ends with an FVS F_{n-k-1} of size bounded by k for the graph $G_{n-k-1} = G[V_{n-k-1}] = G$.

This process is involved in solving at most $n-k-1$ instances (G_i, F_i, k) of the DFVS REDUCTION problem, for $0 \leq i \leq n-k-2$. By Theorem E.1, the running time of the process is bounded by $O(n^3 4^k k^3 k!(n-k-1)) = O(n^4 4^k k^3 k!)$, and the process correctly solves the DFVS problem. $\square$

F. Final Remarks

The running time of the algorithm in Theorem E.2 can be further improved by taking advantage of existing approximation algorithms for the FEEDBACK VERTEX SET problem on directed graphs. [41] have developed a polynomial time approximation algorithm for the FEEDBACK VERTEX SET problem that for a given directed graph G, produces an FVS F of size bounded by $c \cdot \tau \log \tau \log\log \tau$ in time $O(n^2 M(n) \log^2 n)$, where c is a constant, τ is the size of a minimum FVS for the graph G, and $M(n) = O(n^{2.376})$ is the complexity of the multiplication of two $n \times n$ matrices. Therefore, for a given instance (G, k) of the DFVS problem, we can first apply the approximation algorithm in [41] to construct an FVS F for the graph G. If $|F| > c \cdot k \log k \log\log k$, then we know that the graph G has no FVS of size bounded by k. On the other hand, suppose that $|F| \leq c \cdot k \log k \log\log k$. Then we pick a subset F_0 of arbitrary k vertices in F, and let $G_0 = G - (F - F_0)$. The set F_0 is an FVS of size k for the graph G_0. Now we can proceed exactly the same way as we did in Theorem E.2: let $F - F_0 = \{v_1, v_2, \ldots, v_h\}$, where $h \leq c \cdot k \log k \log\log k - k$, and let $V_i = V_0 \cup \{v_1, \ldots, v_i\}$, and $G_i = G[V_i]$, for $i = 0, 1, \ldots, h$. By repeatedly applying the algorithm in Lemma E.1, we can either stop with a certain index i where the induced subgraph G_{i+1} has no FVS of size bounded by k (thus the original input graph G has no FVS of size bounded by k), or eventually construct an FVS F_h of size bounded by k for the graph $G_h = G[V_h] = G$. This process calls for the execution of the algorithm in Lemma E.1 at most $h = O(k \log k \log\log k)$ times, and each execution takes time $O(n^3 4^k k^3 k!)$. In conclusion, the DFVS problem can be solved in time

$O(n^3 4^k k^4 k! \log k \log\log k + n^{4.376} \log^2 n)$, where the second term in the complexity is due to the approximation algorithm given in [41].

We presented a parameterized algorithm of running time $O(n^4 4^k k^3 k!)$ for the DFVS problem, which shows that the problem is fixed-parameter tractable, and resolves an outstanding open problem in parameterized computation and complexity. Before we close the chapter, we give a few remarks on our results and on directions for future research.

There is an edge version of the FEEDBACK SET problem, which is called the FEEDBACK ARC SET problem (briefly, the DFAS problem): given a directed graph G and a parameter k, either construct a set of at most k edges in G whose removal leaves a DAG, or report that no such edge set exists. The DFAS problem is also a well-known NP-complete problem [53]. As shown by [41], the DFAS problem and the DFVS problem can be reduced in linear time from one to the other with the same parameter. Therefore, our results also imply an $O(n^4 4^k k^3 k!)$ time algorithm for the DFAS problem.

The techniques developed in this chapter for solving the SKEW SEPARATOR problem seem to be powerful and generally useful in the study of a variety of separator problems. For example, it has been used recently in developing improved algorithms for a multi-cut problem on undirected graphs in which a separator is sought to (uniformly) separate a set of given terminals [23]. It will be interesting to identify the conditions for the multi-cut problems under which these techniques (and their variations and generalizations) are applicable. In particular, it will be interesting to see if the techniques are applicable to derive the fixed-parameter tractability of the FEEDBACK VERTEX SET problem on *weighted and directed graphs.* Note that the fixed-parameter tractability of the problem on weighted and *undirected* graphs has been derived recently [18].

It will be interesting to develop new techniques that lead to faster parameterized algorithms for the DFVS problem and other related problems. For example, is it possible that the DFVS problem can be solved in time $O(c^k n^{O(1)})$ for a constant c? Another direction is to look at the *kernelization* of the DFVS problem, by which we refer to a polynomial-time algorithm that on an instance (G, k) of the DFVS problem, produces a (smaller) instance (G', k') of the problem, such that the size of the graph G' (the kernel) is bounded by a function $g(k)$ of k (but independent of the size of the original graph G), that $k' \leq k$, and that the graph G has an FVS of size bounded by k if and only if the graph G' has an FVS of size bounded by k'. Since now it is known that the DFVS problem is fixed-parameter tractable, by a general theorem in parameterized complexity theory [37], such a kernelization algorithm exists for the DFVS problem. However, how small can the size of the kernel G' be? In particular, can the kernel G' have its size bounded by a polynomial of the parameter k? We note that recently there has been progress in the study of kernelization for the FEEDBACK VERTEX SET problem on undirected graphs. [7] was able to give a kernel of size $O(k^3)$ for the FEEDBACK VERTEX SET problem on undirected graphs, and [8] have shown that the FEEDBACK VERTEX SET problem on undirected planar graphs has a kernel of size $O(k)$.

```
Algorithm SMC($G, [S_1, \ldots, S_l], [T_1, \ldots, T_l], k$)
input: an instance $(G, [S_1, \ldots, S_l], [T_1, \ldots, T_l], k)$ of the SKEW SEPARATOR
        problem.
output: a skew separator of size bounded by $k$ for the pair
        $([S_1, \ldots, S_l], [T_1, \ldots, T_l])$, or report "No" (i.e., no such separator
        exists).

1.    if $l = 1$ then solve the problem in time $O(kn^2)$;
2.    if Rule R2 applies or $k < 0$ then return "No";
3.    if Rule R1 applies then
          return SMC($G, [S_1, \ldots, S_{l-1}], [T_1, \ldots, T_{l-1}], k$);
4.    if Rule R3 applies on a vertex $w$
      then return $\{w\} \cup$ SMC($G - w, [S_1, \ldots, S_l], [T_1, \ldots, T_l], k - 1$); §
5.    pick an $S_l$-extended vertex $u_0$;   let $S'_l = S_l \cup \{u_0\}$;
6.    let $m$ be the size of a min-cut from $S_l$ to $T_{all} = \bigcup_{i=1}^{l} T_i$;
7.    if $m > k$ then return "No";
8.    let $m'$ be the size of a min-cut from $S'_l$ to $T_{all}$;
9.    if ($m = m'$)
9.1.  then return SMC($G, [S_1, \ldots, S_{l-1}, S'_l], [T_1, \ldots, T_{l-1}, T_l], k$);
9.2.  else $X = \{u_0\} \cup$ SMC($G - u_0, [S_1, \ldots, S_l], [T_1, \ldots, T_l], k - 1$);
          if $X \neq$ "No" then return $X$;
9.3.      else return SMC($G, [S_1, \ldots, S_{l-1}, S'_l], [T_1, \ldots, T_{l-1}, T_l], k$).

§ For simplicity, we assume that a "No" plus anything gives a "No".
```

Fig. 8. An algorithm for the SKEW SEPARATOR problem.

```
Algorithm DBF($(G, D_1, D_2, k)$)
input: an instance $(G, D_1, D_2, k)$ of the DAG-BIPARTITION FVS problem.
output: a $D_1$-FVS of size bounded by $k$ for $G$, or report "No"
      (i.e., no such $D_1$-FVS exists).

1.    for each topologically sorted order $\pi = \{v_1, \ldots, v_h\}$ of the vertices in
      $G[D_2]$ do
1.1.    construct the instance $(G_\pi, [\{s_1\}, \ldots, \{s_h\}], [\{t_1\}, \ldots, \{t_h\}], k)$ of
        the SKEW SEPARATOR problem induced by $(G, D_1, D_2, k)$ and $\pi$;
1.2.    let $X = \mathbf{SMC}(G_\pi, [\{s_1\}, \ldots, \{s_h\}], [\{t_1\}, \ldots, \{t_h\}], k)$;
1.3.    if $X$ is a $D_1$-FVS of size bounded by $k$ for $G$
          then return($X$); stop;
2.    return("No").
```

Fig. 9. An algorithm for the DAG-BIPARTITION FVS problem.

CHAPTER VI

MAX-LEAF

In this chapter, we give an fpt-algorithm of running time $O^*(4^k)$ for the MAX-LEAF OUT-BRANCHING problem on directed graphs, thus improving the previous best algorithm of running time $O^*(2^{O(k^3 \log k)})$ [12]. The algorithm also works for the MAX-LEAF OUT-BRANCHING problem on undirected graphs, thus improving the previous best algorithm of running time $O^*(6.75^k)$ for the MAX-LEAF SPANNING-TREE problem on undirected graphs [14].

As we do in previous chapters, we consider two measures for the MAX-LEAF OUT-BRANCHING problem on directed graphs. Our algorithm extends an out-tree gradually into an out-branching T_s. During the process of extending an out-tree, we divide the leaves of the out-tree into two subsets L_1 and L_2 such that leaves in L_1 will reach exactly one leaf in T_s and the leaves may reach more than one leaves in T_s. The sizes of L_1 and L_2 are the measures for our algorithm. These two measures are *indirect* measures, since both are not the parameter–the number of leaves in an out-branching–for the MAX-LEAF OUT-BRANCHING problem.

A. Introduction

The MAX-LEAF SPANNING-TREE problem is to find a spanning tree with the maximum number of leaves in an undirected graph. This problem has a version for directed graphs (i.e., digraphs), which is the MAX-LEAF OUT-BRANCHING problem. These problems are of importance in many theoretical and practical applications [56, 97, 102]. The problems are NP-hard [53]. In terms of polynomial time approximability, the MAX-LEAF SPANNING-TREE problem is APX-hard, and can be approximated with a ratio 2 in polynomial time [95], while very recent research shows that the

MAX-LEAF OUT-BRANCHING problem can be approximated with a ratio $O(\sqrt{n})$ in polynomial time [40].

These problems have been intensively studied [1, 2, 9, 11, 12, 14, 43, 79, 95]. For the parameterized MAX LEAF problem on undirected graphs, there is a chain of improved algorithms: $O((17k^4)!)$ [9], then $O((2k)^{4k})$ [36], then $O(14.23^k)$ [43], then $O(9.49^k)$ [11] and finally $O(6.75^k)$ [14]. For the parametrized MAX LEAF problem on digraph, algorithm of time $O(2^{O(k^2 logk)})$ was given for strongly connected digraph and acyclic digraph [2], which was improved to $O(2^{klog^2 k})$ for strongly connected digraph and $O(2^{klogk})$ for acyclic digraph. The parameterized MAX LEAF problem on general digraph was proposed as an open problem in [1], [2], and [58], and was shown to be fixed-parameter tractable with an algorithm of time $O(2^{O(k^3 logk)})$ [12].

The more general MAX-LEAF OUT-BRANCHING problem seems more difficult. The problem has become significantly interesting very recently because of its rich combinatorial structures and challenging algorithmic techniques. In this chapter, we present a simple branch-and-search algorithm of running time $O^*(4^k)$ that solves the MAX-LEAF OUT-BRANCHING problem. This result significantly improves the previous best algorithm for the problem that runs in time $O^*(2^{O(k \log k)})$ [13]. The MAX-LEAF SPANNING-TREE problem can be trivially reduced to the MAX-LEAF OUT-BRANCHING problem if we convert the input undirected graph G into a digraph by replacing each undirected edge $[uv]$ by two directed edges uv and vu. Therefore, our algorithm of time $O^*(4^k)$ can be used to solve the MAX-LEAF SPANNING-TREE problem, improving the previous best algorithm of running time $O^*(6.75^k)$ [14].

B. Preliminaries

All graphs in our discussion are simple graphs, i.e., there are no multiple edges from a vertex to another vertex, and no self-loops on any vertex. For an edge xy in a digraph G, the vertex x is called an *in-neighbor* of the vertex y, and the vertex y is called an *out-neighbor* of the vertex x. The *in-degree* of a vertex x in G is the number of in-neighbors of x, and the *out-degree* of a vertex x is the number of out-neighbors of x.

A *path* P in a digraph G from a vertex x_1 to another vertex x_q is a sequence of vertices $P = x_1 \cdots x_q$ such that $x_i x_{i+1}$ is an edge in G for all $1 \leq i \leq q-1$. The path P is *simple* if no vertex is repeated in P. A vertex x can *reach* another vertex y (or equivalently, the vertex y is reachable from the vertex x) in a digraph G if there is a path from x to y in G.

An acyclic digraph T is an *out-tree* (rooted at a vertex r) if the vertex r has in-degree 0 and for every vertex x in T there is a unique path from r to x. The vertex r is the *root* of the out-tree T, and each vertex of out-degree 0 is a *leaf* of T. A vertex x in T is an *internal vertex* if it is not a leaf. A *k-out-tree* is an out-tree that has at least k leaves. An out-tree *in a digraph* G is a subgraph of G that is an out-tree. An *out-branching* of a digraph G is an out-tree of G that contains all vertices of G. A *k-out-branching* is an out-branching that has at least k leaves.

Let T be an out-tree in a digraph G, and let x_0 be a leaf of T. An *out-chain* $\pi(l)$ of T is a simple path $x_0 x_1 \ldots x_q$ in G such that for all $0 \leq i \leq q-1$, x_{i+1} is the only out-neighbor of x_i not in $T \cup \{x_1, \ldots, x_i\}$. A *$y$-truncated* out-chain $\pi_y(l)$ of an out-chain $\pi(l)$, where $y \neq l$, is defined as follows: (1) if $y \in \pi(l)$, then $\pi_y(l)$ is the path from l to z in $\pi(l)$, where z is the in-neighbor of y in $\pi(l)$; and (2) if $y \notin \pi(l)$, then $\pi_y(l) = \pi(l)$. Inductively, $\pi_{y_1,\ldots,y_q}(l)$ is the y_q-truncated out-chain of $\pi_{y_1,\ldots,y_{q-1}}(l)$.

For a given set Q of out-chains of the out-tree T, a leaf l of T is *determined* if an out-chain $\pi(l)$ is included in Q. An ordered set Q of out-chains $\pi(l_1), \ldots, \pi(l_q)$ is *consistent* if $\pi(l_i)$ and $\pi(l_j)$ contain a common vertex z, then the path from z to the last vertex of w $\pi(l_i)$ is also in $\pi(l_j)$ for all $1 \leq i < j \leq q$.

Let T be an out-tree in a digraph G with a given consistent set $\{\pi(l_1), \ldots, \pi(l_p)\}$ of out-chains, and let k be a positive integer. A k-out-branching T_s is an *extended out-branching for* $(T; \pi(l_1), \ldots, \pi(l_p); k)$ such that (1) T_s and T have the same root, (2) T is a subgraph of T_s, and (3) all vertices reachable in T_s from l_i are in $\pi(l_i)$.

We formulate the following problem.

> EXTENDING MAX-LEAF: given a digraph T, an out-tree T in G with a consistent set $\{\pi(l_1), \ldots, \pi(l_q)\}$ of out-chains, and a parameter k, either construct an extended out-branching for $(T; \pi(l_1), \ldots, \pi(l_p); k)$, or report 'No' if no such an out-branching exists.

In the rest of this chapter, $(T; \pi(l_1), \ldots, \pi(l_p); k)$ also refers to the instance of the EXTENDING MAX-LEAF problem with inputs of an out-tree T, a consistent set of out-chains $\{\pi(l_1), \ldots, \pi(l_p)\}$ for T, and an integer k.

The following reduction will play an important role in our discussion.

Definition An instance $(T; \pi(l_1), \ldots, \pi(l_p); k)$ is *reducible to* another instance $(T'; \pi'(l_1), \ldots, \pi'(l_q); k')$ if

(1) any extended out-branching for $(T'; \pi'(l_1), \ldots, \pi'(l_q); k')$ is an extended out-branching for $(T; \pi(l_1), \ldots, \pi(l_p); k)$;

(2)if there is an extended out-branching for $(T; \pi(l_1), \ldots, \pi(l_p); k)$, there is an extended out-branching for $(T'; \pi'(l_1), \ldots, \pi'(l_q); k')$.

It is easy to verify that the above reduction is a transitive relation.

The following notations will be used to simplify our discussion.

- $T + xy_1 + \ldots + xy_q$ is the the resulting graph after adding edge $xy_1, \ldots, xy_q$ to T and including vertex y_i in $T + xy$ if y_i is not in T for all $1 \leq i \leq q$.
- $T - zy$ is the resulting graph after removing edge zy from T and keeping all vertices of T in $T - zy$.
- $G - T$ is the resulting graph after removing all edges between vertices of T from G.

In the rest of this section, we give two lemmas on out-trees and out-branchings, which will be used in our later discussion.

Lemma B.1 *Let T and T' be two out-trees in a digraph with the same root such that T' is a subgraph of T, and let l be a leaf of T' and $y \neq l$ be a vertex in T'. Then l cannot reach y in T.*

Proof. Since y is a vertex of T' and T' is an out-tree, there is a unique path P_1 in T' from the root r of T' to y, which obviously does not pass through the leaf l because l has out-degree 0 in T'. Since T' is a subgraph of T, P_1 is also a path from r to y in T. Now if l can reach y in T, then the unique path in T from r to l followed by the path from l to y in T forms a second path P_2 from r to y in T that passes through the vertex l. This contradicts the assumption that T is an out-tree since r is also the root of T. □

Lemma B.2 *Let T_s be an out-branching and T be a k-out-tree in a digraph such that T_s and T have the same root r and T is a subgraph of T_s. Then T_s is a k-out-branching.*

PROOF. Let $\{l_1, \ldots, l_p\}$ be the set of all leaves of T. For each $1 \leq i \leq p$, let L_i be the set of leaves of T_s that can be reached from l_i in T_s. Note that each set L_i contains at least one vertex in T_s.

We prove that no two sets L_i and L_j share a common vertex. Suppose that $w \in L_i \cap L_j$. Then the unique path P_i' in T (which is also the unique path in T_s) from the root r to l_i followed by a path P_i'' from l_i to w in T_s forms a path P_i in T_s from the root r to w. Similarly, there is a path P_j in T_s from the root r to w that passes through the vertex l_j. Since T_s is an out-tree, we must have $P_i = P_j$. In consequence, the path P_i in T_s contains the vertex l_j. Since both l_i and l_j are leaves in T, the unique path P_i' in T from r to l_i does not contain the vertex l_j. Therefore, the vertex l_j must be contained in the path P_i'' from l_i to w, so l_i can reach l_j in T_s. However, this is impossible according to Lemma B.1.

Therefore, T_s has at least $|L_1| + \cdots + |L_p| \geq p$ leaves. Since T is a k-out-tree, $p \geq k$. Thus, T_s is a k-out-branching. □

C. Extending an Out-tree

Throughout the discussion of this section, we let T be an out-tree with the root r in a digraph G, a given consistent set $\{\pi(l_1), \ldots, \pi(l_p)\}$ of out-chains for T. Let x be a vertex in T that is not a determined leaf of T and let y be an out-neighbor of x in $G - T$. In subsection 1, we discuss some properties that are useful for reducing the instance $(T; \pi(l_1), \ldots, \pi(l_p); k)$ to the instance $(T + xy; \pi_y(l_1), \ldots, \pi_y(l_p); k)$. In subsection 2, we show that the results in subsection 1 can be generalized to the case where many out-neighbors of x in $G - T$ are considered. Furthermore, if T is already a k-out-tree, then we can solve the instance $(T; \pi(l_1), \ldots, \pi(l_p); k)$ in polynomial time, which gives a boundary condition for our algorithm.

1. Properties for Extending an Out-tree

We first verify that $T + xy$ is an out-tree in G and that the y-truncated out-chains $\pi_y(l_1)$, …, $\pi_y(l_p)$ are a consistent set of out-chains for $T+xy$. This is necessary before we can discuss the extendability of $(T + xy; \pi_y(l_1), \ldots, \pi_y(l_p); k)$.

Lemma C.1 *$T + xy$ is an out-tree with root r in the digraph G.*

Proof. Since the vertex y has out-degree 0 in $T + xy$, it cannot help creating any new path in $T + xy$ from the root r to any vertex $w \neq y$ in $T + xy$. Thus, there is a unique path from the root r to w in $T+xy$. Moreover, since x is the only in-neighbor of y in $T+xy$, the only path from the root r to y in $T+xy$ must be the unique path from r to x followed by the edge xy. Finally, y is of in-degree one, r is of in-degree zero, and any other vertex $w \neq r$ of T is of in-degree one in $T+xy$. Therefore, $T+xy$ is an out-tree. □

Lemma C.2 *The truncated out-chains $\pi_y(l_1)$, …, $\pi_y(l_p)$ is a consistent set of out-chains for $T + xy$.*

Proof. Pick any out-chain of T: $\pi(l_i) = \{u_0, u_1, \ldots, u_q\}$, where $u_0 = l_i$. Then u_j has a unique out-neighbor u_{j+1} in $G - (T \cup \{u_0, \ldots, u_j\})$ for all $0 \leq j \leq q-1$. Thus, and by that T is a subgraph of $T + xy$ and the definition of out-chain, u_{j+1} is the unique out-neighbor in $G - (T + xy + u_0 \cdots u_j)$ for any $u_j \in \pi_y(l_i)$ except the last vertex in $\pi_y(l_i)$. So $\pi_y(l_i)$ is an out-chain of $T + xy$ for all $1 \leq i \leq p$.

It is given that $\pi(l_1), \ldots, \pi(l_p)$ is a consistent set of out-chains for T, i.e., if z is in both $\pi(l_i)$ and $\pi(l_j)$ where $i < j$, then the path from z to the last vertex w of $\pi(l_i)$ is in $\pi(l_j)$. Let z be a common vertex of $\pi_y(l_i)$ and $\pi_y(l_j)$ where $1 \leq i < j \leq q$. If $y \neq w$ is in the path from w to the last vertex of $\pi(l_j)$, then $\pi_y(l_i) = \pi(l_i)$ and $\pi_y(l_j)$ is the path from l_j to the in-neighbour of y in $\pi_y(l_j)$, which also includes the path from z to w in $\pi(l_i)$. That is, the path from z to w in $\pi_y(l_i)$ is also in $\pi_y(l_j)$. If y is in the path from z to w in $\pi(l_i)$, let p be the in-neighbour of y in $\pi(l_i)$. Then $\pi_y(l_i)$ is the path from l_i to p in $\pi(l_i)$, and $\pi_y(l_j)$ is the path from l_j to p in $\pi(l_i)$. Since the path from z to w is in both $\pi(l_i)$ and $\pi(l_j)$, and y is in the path from z to w, the path from z to p is in both $\pi_y(l_i)$ and $\pi_y(l_j)$. If $y \neq z$ is either in the path from l_i to z in $\pi(l_i)$ or in the path from l_j to z in $\pi(l_j)$, $\pi_y(l_i)$ and $\pi_y(l_j)$ have no common vertex. Finally, if y is not in $\pi(l_i)$ nor $\pi(l_j)$, $\pi_y(l_i) = \pi(l_i)$ and $\pi_y(l_j) = \pi(l_j)$. Thus, the path from z to the last vertex of $\pi_y(l_i)$ in $\pi_y(l_i)$ is in $\pi_y(l_j)$. In summary, if $\pi_y(l_i)$ and $\pi_y(l_j)$ have a common vertex z, the path from z to the last vertex of $\pi_y(l_i)$ is in $\pi_y(l_j)$ for all $1 \leq i \leq j$. Therefore, the set $\{\pi_y(l_1), \ldots, \pi_y(l_p)\}$ of truncated out-chains is consistent. □

Lemma C.3 *Any extended out-branching for* $(T + xy; \pi_y(l_1), \ldots, \pi_y(l_p); k)$ *is an extended out-branching for* $(T; \pi(l_1), \ldots, \pi(l_p); k)$.

PROOF. Let T_s be an extended out-branching for $(T + xy; \pi_y(l_1), \ldots, \pi_y(l_p); k)$. By the definition we have: (1) T_s is a k-out-branching; (2) $T + xy$ and T_s has the same root r; (3) $T + xy$ is a subgraph of T_s; and (4) all vertices reachable in T_s from l_i are in $\pi_y(l_i)$. We show T_s is also an extended out-branching for $(T; \pi(l_1), \ldots, \pi(l_p); k)$.

By Lemma C.1, T and $T + xy$ have the same root. Thus, by (2), T and T_s have the same root. Since T is a subgraph of $T + xy$, by (3) T is also a subgraph of T_s. Finally, since $\pi_y(l_i) \subseteq \pi(l_i)$, by (4), all vertices reachable in T_s from l_i are in $\pi(l_i)$. In conclusion, T_s is an extended out-branching for $(T; \pi(l_1), \ldots, \pi(l_p); k)$. □

In the rest of this section, we assume that T_s is an extended out-branching for $(T; \pi(l_1), \ldots, \pi(l_p); k)$. Let z be the (unique) in-neighbor of y in T_s. We will prove that $T_s - zy + xy$ is an extended out-branching for $(T + xy; \pi_y(l_1), \ldots, \pi_y(l_p); k)$. We first prove the following properties.

Lemma C.4 *$T + xy$ is a subgraph of $T_s - zy + xy$.*

Proof. Since T is a subgraph of T_s, and $y \notin T$ so the edge zy is not in T, all edges in T are in $T_s - zy + xy$. Moreover, the edge xy is contained in $T_s - zy + xy$. Therefore, all edges in $T + xy$ are in $T_s - zy + xy$ i.e., the out-tree $T + xy$ is a subgraph of $T_s - zy + xy$. □

Lemma C.5 *$T_s - zy + xy$ is an out-branching of G with root r. Moreover, if x is not a leaf of T, then $T_s - zy + xy$ is an k-out-branching of G with root r.*

Proof. Since T_s is an out-branching of the digraph G, by the definition, $T_s - zy + xy$ is also rooted at r and contains all vertices in G. For any vertex w in G, if the unique path P_w in T_s from the root r to w does not pass through the vertex y, then P_w is also a path in $T_s - zy + xy$. On the other hand, if the path P_w from r to w is a concatenation of a path P'_w from r to y and a path P''_w from y to w, then the path in T from r to x (note that $zy \notin T$) followed by the edge xy then by the path P''_w ($zy \notin P_w$" since y is of in-degree one) makes a path in $T_s - zy + xy$ from r to w. The uniqueness of the path in $T_s - zy + xy$ from r to w can be easily verified because each vertex in $T_s - zy + xy$ has in-degree bounded by 1.

If x is not a leaf of T_s, then deleting the edge zy then adding the edge xy can not decrease the number of vertices of degree 0 in the out-branching. Thus, $T_s - zy + xy$ has at least as many leaves as that of T_s. Since T_s is an extended out-branching for $(T; \pi(l_1), \ldots, \pi(l_p); k)$, T_s has at least k leaves. In consequence, $T_s - zy + xy$ is a k-out-branching. □

Lemma C.6 *The vertices reachable in $T_s - zy + xy$ from a determined leaf l_i are all in $\pi_y(l_i)$.*

Proof. We first prove that the vertices reachable in $T_s - zy + xy$ from the determined leaf l_i are all in $\pi(l_i)$. Suppose that l_i can reach a vertex $w \notin \pi(l_i)$ in

$T_s - zy + xy$ via a path P. Then the edge xy must be in P. Otherwise, the path P is also in T_s and l_i can reach $w \notin \pi(l_i)$ in T_s, contradicting the assumption that T_s is an extended out-branching for $(T; \pi(l_1), \ldots, \pi(l_p); k)$. Thus, l_i can reach x in T_s, which is impossible by Lemma B.1 because both l_i and x are in T, and l_i is a determined leaf of T but x is not a determined leaf. This contradiction proves that all vertices reachable in $T_s - zy + xy$ from l_l must be in $\pi(l_i)$.

Now we prove that the vertices reachable in $T_s - zy + xy$ from l_i are all in $\pi_y(l_i)$. If $y \notin \pi(l_i)$, then $\pi_y(l_i) = \pi(l_i)$, and the lemma is proved by the above analysis. Suppose $y \in \pi(l_i)$ and l_i can reach a vertex $w \in \pi(l_i) - \pi_y(l_i)$ via a path P'. Since l_i is a leaf of T, by Lemma B.1 any vertex $u \neq l_i$ of T cannot be in P'. Thus, $\pi_y(l_i) \cup \{y\}$ must be in P'. However, since (1) both l_i and y are in $T + xy$, (2) l_i is a leaf of $T + xy$, (3) $T_s - zy + xy$ have the same root as T by Lemmas C.1 and C.5, and (4) $T + xy$ is a subgraph of $T_s - zy + xy$ by Lemma C.4, therefore, l_i can not reach y in $T_s - zy + xy$ by Lemma B.1. Thus, all vertices reachable from l_i in $T_s - zy + xy$ are in $\pi_y(l_i)$ when $y \in \pi(l_i)$. □

2. Extending an Out-tree

Now, we are ready to show that we can extend $T + xy$ instead of T. Note that $T + xy$ can be found in polynomial time, and this can occur at most n times because $T + xy$ has one more vertex than T. Thus, this is an efficient operation without branching.

Lemma C.7 *If x is an internal vertex of T, the instance $(T; \pi(l_1), \ldots, \pi(l_p); k)$ is reducible to the instance $(T + xy; \pi_y(l_1), \ldots, \pi_y(l_p); k)$*[1].

PROOF. Let T_s be an extended out-branching for $(T; \pi(l_1), \ldots, \pi(l_p); k)$. By

---

[1] *by lemma C.2, $\{\pi_y(l_1), \ldots, \pi_y(l_p)\}$ is a consistent set of out-chains. The requirement on a consistent set of out-chains can be verified to be true for instances discussed in this chapter. We ignore this in later discussions.*

Lemma C.5, $T_s - zy + xy$ is a k-out-branching with root r where z is the unique in-neighbour of y in T_s. Thus $T_s - zy + xy$ and $T + xy$ have the same root according to Lemma C.1, and $T + xy$ is a subgraph of $T_s - zy + xy$ by Lemma C.4. Finally, all vertices reachable in $T_s - zy + xy$ from a determined leaf l_i are in $\pi_y(l_i)$ by Lemma C.6. In consequence, $T_s - zy + xy$ is an extended out-branching for $(T + xy; \pi_y(l_1), \ldots, \pi_y(l_p); k)$.

By Lemma C.3, any extended out-branching for $(T + xy; \pi_y(l_1), \ldots, \pi_y(l_p); k)$ is an extended out-branching for $(T; \pi(l_1), \ldots, \pi(l_p); k)$. This completes the proof of the lemma. $\square$

Lemma C.7 can be generalized to the case where we add many out-neighbors of the vertex x in $G - T$ to the out-tree T. For this, let y_1, …, y_q be out-neighbors of x in $G - T$.

Lemma C.8 *If x is an internal vertex of T, the instance $(T; \pi(l_1), \ldots, \pi(l_p); k)$ is reducible to the instance $(T + xy_1 + \ldots + xy_q; \pi_{y_1,\ldots,y_q}(l_1), \ldots, \pi_{y_1,\ldots,y_q}(l_p); k)$.*

PROOF. We prove the lemma by induction on q. The initial case $q = 1$ is proved by Lemma C.7.

Now suppose that when $q = i$, the instance $(T; \pi(l_1), \ldots, \pi(l_p); k)$ is reducible to the instance $(T + xy_1 + \ldots + xy_i; \pi_{y_1,\ldots,y_i}(l_1), \ldots, \pi_{y_1,\ldots,y_i}(l_p); k)$. For $q = i + 1$, note that $T + xy_1 + \ldots + xy_{i+1}$ is the out-tree $T + xy_1 + \ldots + xy_i$ plus the edge xy_{i+1} where $y_{i+1} \notin T + xy_1 + \ldots + xy_i$, and the truncated out-chain $\pi_{y_1,\ldots,y_i,y_{i+1}}(l_j)$ is a y_{i+1}-truncated out-chain of the out-chain $\pi_{y_1,\ldots,y_i}(l_j)$ for the tree $T + xy_1 + \ldots + xy_i$ for all j. Moreover, the vertex x is obviously an internal vertex of the tree $T + xy_1 + \ldots + xy_i$. Therefore, by Lemma C.7 again, the instance $(T + xy_1 + \ldots + xy_i; \pi_{y_1,\ldots,y_i}(l_1), \ldots, \pi_{y_1,\ldots,y_i}(l_p); k)$ is reducible to the instance $(T + xy_1 + \ldots + xy_{i+1}; \pi_{y_1,\ldots,y_{i+1}}(l_1), \ldots, \pi_{y_1,\ldots,y_{i+1}}(l_p); k)$. Now the transitivity of the reduction proves that the instance $(T; \pi(l_1), \ldots, \pi(l_p); k)$

is reducible to the instance $(T + xy_1 + \ldots + xy_{i+1}; \pi_{y_1,\ldots,y_{i+1}}(l_1), \ldots, \pi_{y_1,\ldots,y_{i+1}}(l_p); k)$, and the induction goes through. $\square$

The conditions in the previous lemmas can be further relaxed without requiring that the vertex x be an internal vertex of T, if the out-tree T is a k-out-tree, as shown in the following lemmas.

Lemma C.9 *Suppose that x is not a determined leaf of T and T is a k-out-tree. Then $T + xy$ is a k-out-tree and the instance $(T; \pi(l_1), \ldots, \pi(l_p); k)$ is reducible to the instance $(T + xy; \pi_y(l_1), \ldots, \pi_y(l_p); k)$.*

PROOF. By Lemma C.1, $T + xy$ is an out-tree with root r. The number of leaves of $T + xy$ cannot be less than that of T because adding the edge xy to T adds a vertex y of out-degree 0 and may change the out-degree of at most one vertex in T (i.e., x). Since T is a k-out-tree, $T + xy$ is also a k-out-tree. Next we show that the instance $(T; \pi(l_1), \ldots, \pi(l_p); k)$ is reducible to the instance $(T + xy; \pi_y(l_1), \ldots, \pi_y(l_p); k)$.

By Lemma C.3, any extended out-branching for $(T + xy; \pi_y(l_1), \ldots, \pi_y(l_p); k)$ is an extended out-branching for $(T; \pi(l_1), \ldots, \pi(l_p); k)$. Now suppose that there is an extended out-branching T_s for $(T; \pi(l_1), \ldots, \pi(l_p); k)$. By Lemma C.5, the graph $T_s - zy + xy$ is an out-branching. Since T is a subgraph of $T_s - zy + xy$ by Lemma C.4 and T is a k-out-tree, $T_s - zy + xy$ is a k-out-branching by Lemma B.2. Also $T_s - zy + xy$ and $T + xy$ have the same root r according to Lemma C.1 and Lemma C.5. Moreover, $T + xy$ is a subgraph of $T_s - zy + xy$ by Lemma C.4. Finally, all vertices reachable in $T_s - zy + xy$ by a determined leaf l_i are in $\pi_y(l_i)$ by Lemma C.6. So $T_s - zy + xy$ is a $(T + xy; \pi_y(l_1), \ldots, \pi_y(l_p); k)$-extended out-branching.

Therefore, $T + xy$ is a k-out-tree with root r and the instance $(T; \pi(l_1), \ldots, \pi(l_p); k)$ is reducible to the instance $(T + xy; \pi_y(l_1), \ldots, \pi_y(l_p); k)$. $\square$

Similarly, we can generalize Lemma C.9 to many out-neighbors of the vertex x. For this, again let y_1, ..., y_q be out-neighbors of x in $G - T$.

Lemma C.10 *When T is a k-out-tree, the instance $(T; \pi(l_1), \ldots, \pi(l_p); k)$ is reducible to the instance $(T + xy_1 + \ldots + xy_q); \pi_{y_1,\ldots,y_q}(l_1), \ldots, \pi_{y_1,\ldots,y_q}(l_p); k)$, if x is not a determined leaf of T.*

Proof. The lemma is proved by induction on q. The initial case $q = 1$ is proved by Lemma C.9, and the inductive step is proved by Lemma C.9 and the transitivity of the reduction. □

Lemma C.10 is very useful since it implies that by continuously adding out-neighbors of a undetermined leaf, if T is already a k-out-tree, we can find an extended out-branching for $(T, ; \pi(l_1), \ldots, \pi(l_p); k)$. This will be discussed in the following algorithm and its proof.

Theorem C.11 *For a given k-out-tree T in a digraph G of n vertices, with a set $\{\pi(l_1), \ldots, \pi(l_p)\}$ of out-chains for T, the algorithm SpanTree$(G, T; \pi(l_1), \ldots, \pi(l_p); k)$ runs in polynomial time and (1) either constructs an extended out-branching for $(T; \pi(l_1), \ldots, \pi(l_p); k)$ if such one exists, or (2) reports 'No' otherwise.*

Proof. We first prove the correctness of the algorithm by showing that (1) the instance $(T_1; \pi_1(l_1), \ldots, \pi_1(l_q); k)$ before step 2 is reducible to $(T_2; \pi_2(l_1), \ldots, \pi_2(l_q); k)$, the instance before step 3, (2) step 3 returns 'No' correctly, (3) step 4 construct an extended out-branching for the instance $(T_4; \pi_4(l_1), \ldots, \pi_4(l_q); k)$ before step 4, and (4) step 5 returns an extended out-branching for the instance $(T_4; \pi_4(l_1), \ldots, \pi_4(l_q); k)$.

Claim S does not contain any determined leaf of T. Moreover, any vertex of T which have out-neighbours in $G - T$ is either in S or a determined leaf of T.

We prove this Claim by induction on the number of iterations of the while loop in step 2.

```
SpanTree($G, T; \pi(l_1), \dots, \pi(l_p); k$)
INPUT: a $k$-out-tree $T$ in a digraph $G$ with a $\{\pi(l_1), \dots, \pi(l_p)\}$ of
        out-chains for $T$, and a parameter $k$
OUTPUT: an extended out-branching for $(T; \pi(l_1), \dots, \pi(l_p); k)$ if
        there is such one, or "No" otherwise
1.    $S = T - \{l_1, \dots, l_q\}$;
2.    while $S \neq \emptyset$ do
        pick any vertex $x \in S$;
2.1     if $x$ has out-neighbors $y_1, \dots, y_q$ in $G - T$ then
          for $i = 1$ to $p$  do $\pi(l_i) = \pi_{y_1,\dots,y_q}(l_i)$;
          $T = T + xy_1 + \dots + xy_q$;
          $S = (S \cup \{y_1, \dots, y_q\}) - \{x\}$;
2.2     else $S = S - \{x\}$;
3.    if there is a vertex $x \notin T \cup \{\pi(l_1), \dots, \pi(l_p)\}$ then return 'No';
4.    for $i = p$ to 1  do
        $x = l_i$;
4.1     while $x$ has an out-neighbor in $\pi(l_i)$ do
          $T = T + xy$;       $x = y$;
5.   return $T$.
```

Fig. 10. Algorithm for the SPAN k-OUT-TREE problem

Before the first iteration of the while loop in step 2, S contains all vertices of T which are not determined leaves of T, according to step 1. Thus S contains all possible vertices of T which are not determined leaves of T and have out-neighbours in $G - T$.

Now assume that S does not contain any determined leaf of T, and any vertex of T which has out-neighbours in $G - T$ is either in S or a determined leaf of T before a iteration of the while loop at step 2. We need to show that this is true before the next iteration of the while loop in step 2.

In step 2.1, only $y_1, \dots, y_q$, which are not in T, are added to T, and no out-chains of $y_1, \dots, y_q$ for $T + xy_1 + \dots + xy_q$ are created. So S does not contain any determined leaf of $T+xy_1+\ldots+xy_q$. Moreover, the vertices of $T+xy_1+\ldots+xy_q$ are the vertices of T plus $y_1,\ldots,y_q$. By our inductive hypothesis, any vertex in $T+xy_1+\ldots+xy_q$ which has out-neighbours not in $T+xy+\ldots+xy_q$ is either in $S\cup\{y_1,\ldots,y_q\}$ or a determined leaf of T which is also a determined leaf of $T+xy+\ldots+xy_q$. Since $T+xy+\ldots+xy_q$ is the T and $S\cup\{y_1,\ldots,y_q\}$ is the S before the next iteration, the Claim is true before the next iteration of the while loop in step 2.

In step 2.2, x has no out-neighbours in $G-T$, according to the condition of the if statement in step 2.1. By out inductive hypothesis, $S-x$ does not contain any determined leaf of T, and any vertex of T which have out-neighbours in $G-T$ is either in $S-x$ or a determined leaf of T. Since the T before the next iteration is the same T before this iteration, and the S before the next iteration is $S-x$, the Claim is true before the next iteration of the while loop in step 2. This completes the proof of the Claim.

By the Claim, S in step 2.1 does not contain any determined leaf of T. Then $x\in S$ in step 2.1 is not a determined leaf of T. By Lemma C.10 and that k is a k-out-tree, the instance $(T;\pi(l_1),\cdots,\pi(l_q);k)$ before in step 2.1 is reducible to the instance $(T+xy_1+\cdots+xy_q;\pi_{y_1,\cdots,y_q}(l_1),\cdots,\pi_{y_1,\cdots,y_q}(l_q);k)$. Therefore, by the transitivity of the reduction, the instance $(T_1;\pi_1(l_1),\ldots,\pi_1(l_q);k)$ before step 2 is reducible to $(T_2;\pi_2(l_1),\ldots,\pi_2(l_q);k)$, the instance before step 3.

By the Claim, only determined leaves of T_3 can have out-neighbours in $G-T_3$. Since any determined leaf l_i of T_i can only reach vertices in $\pi(l_i)$, step 3 returns ‘NO’ correctly, if there is a vertex x not in T nor in any out-chain.

After step 3, every vertex not in T is in some out-chain. We prove that step 4 finds an extended out-branching T_s for the instance $(T_4;\pi_4(l_1),\ldots,\pi_4(l_q);k)$ where T_s is the T after step 4 by the following loop invariant: before the tth execution of step 4.1, T is an out-tree with the same root as T_4, the vertices of any out-chain in the ordered set $\{\pi_4(l_{p-t+2}), \ldots, \pi_4(l_{p-1}), \pi_4(l_p)\}$ are in T, and if a vertex w of $\pi_4(l_j)$ where $j < p-t+2$ is in T, then the vertices in the path from w to the last vertex of $\pi_4(l_t)$ are in T. Note that $i = p-t+1$ for the tth execution of step 4.1.

The initialization case is when $i = 1$, since T_4 is an out-tree and the ordered set $\{\pi_4(l_{p-t+2}), \ldots, \pi_4(l_p)\}$ is empty $(p-t+2 = p+1 > p)$. That is, the loop invariant is true for $t = 1$. Suppose the loop invariant is true before the tth execution of step 4.1, i.e., before the tth execution of step 4.1, T is an out-tree with the same root as T_4, the vertices of any out-chain in $\pi_4(l_{p-t+2}), \ldots, \pi_4(l_{p-1}), \pi_4(l_p)$ are in T, and if a vertex w of $\pi_4(l_j)$ where $j < p-t+2$ is in T, then the vertices in the path from w to the last vertex of $\pi_4(l_t)$ are in T.

We show it is still true before the $(t+1)$th iteration. When the step 4.1 is skipped because all vertices of $\pi(l_{p-t+1})$ are already in T, then the loop invariant still holds, since nothing changes.

During the tth execution of step 4.1, all remaining vertices of $\pi_4(l_{p-t+1})$ which are not in T yet are included in T during step 4.1. So all vertices of $\pi_4(l_{p-t+1})$ are in T after the tth execution of step 4.1. If a vertex $w \notin T$ of $\pi_4(l_j)$ where $j < p-t+1$ is visited during the tth execution of step 4.1, then all vertices from w to the last vertex of $\pi_4(l_j)$ are in T after the tth execution of step 4.1, since the set $\{\pi(l_1), \ldots, \pi(l_q)\}$ is consistent.

Moreover, the T before the $(t+1)$th execution of step 4.1, is still an out-tree with the same root as the T before the tth execution, since the ith execution of step 4.1 just adding a path to a leaf of T. By our inductive hypothesis, the T before the tth execution of step 4.1 has the same root as T_4. Thus, the T before the $(t+1)$th execution of step 4.1, has the same root as T_4. In consequence, the loop invariant is true before the $(t+1)$th execution of step 4.1.

When the for loop terminates, $T_s = T$ is an out-tree with the same root as T_4

(by the transitivity of the reduction), and all vertices in $\pi_4(l_1), \ldots, \pi_4(l_q)$ are in T. Thus, T_s is an out-branching with the same root as T_4. Since T_4 is a subgraph of T_s (no edges of T are removed during step 4), T_s is a k-out-branching by Lemma B.2. By step 4.1, l_i can only reach vertices in $\pi_4(l_i)$ in T_s for all $1 \leq i \leq q$. So T_s is an extended out-branching for $(T_4; \pi_4(l_1), \ldots, \pi_4(l_q); k)$.

Step 5 returns T_s constructed in step 4, which is an extended out-branching for $(T_4; \pi_4(l_1), \ldots, \pi_4(l_q); k)$.

Summarizing the above discussion, we conclude that the algorithm SpanTree either constructs an extended out-branching for $(T; \pi(l_1), \ldots, \pi(l_p); k)$ if such one exists, or correctly reports 'No' otherwise.

Now we analyze the complexity of the algorithm.

In both step 2.1 and 2.2, a vertex is removed from S. Only in step 2.1, vertices are added into S. However, only vertices not in T may be added into S in step 2.1, and these vertices are in T once they are added into S. So any vertex can be added into S at most one time. In consequence, step 2 can not be executed with more than n iterations. It is obvious that steps 2.1 and 2.2 take polynomial time. Thus, step 2 can be done in polynomial time. Moreover, we can see that steps 3, 4 and 5 can be done in polynomial time. Therefor, the algorithm SpanTree runs in polynomial time. $\square$

D. The Main Algorithm and Complexity Analysis

Before we solve the MAX-LEAF OUT-BRANCHING problem, we give an algorithm to solve the problem EXTENDING MAX-LEAF with inputs $(T; \pi(l_1), \ldots, \pi(l_p); k)$ on a digraph G, which is used as a subroutine for the algorithm solving the MAX-LEAF OUT-BRANCHING problem. Consider the algorithm ExtendTree given in Figure 11.

```
ExtendTree($G, T; \pi(l_1), \ldots, \pi(l_p); k$)
INPUT: an out-tree $T$ in a digraph $G$, a consistent set $\{\pi(l_1), \ldots, \pi(l_p)\}$
        of out-chains for $T$, and a parameter $k$
OUTPUT: an extended out-branching for $(T; \pi(l_1), \ldots, \pi(l_p); k)$ if such
        one exists, or "No" otherwise.

1.   if $T$ is an out-branching with fewer than $k$ leaves then return 'No';
2.   if $T$ has at least $k$ leaves then
         return SpanTree($G, T; \pi(l_1), \ldots, \pi(l_p); k$);
3.   if an internal vertex $x \in T$ has out-neighbors $y_1$, …, $y_q$ in $G - T$
      then return
         ExtendTree($G, T + xy_1 + \ldots + xy_q; \pi_{y_1,\ldots,y_q}(l_1), \ldots, \pi_{y_1,\ldots,y_q}(l_p); k$);
4.   if all leaves of $T$ are determined leaves then return 'No';
5.   $x$ = any undetermined leaf $l$ of $T$;
     $\pi(l) = x$;    $x_s = x$;
     while $x_s$ has a unique out-neighbor $y$ in $G - (T \cup \pi(l))$ do
         add $y$ to $\pi(l)$;    $x_s = y$;
     if $x_s$ has no out-neighbors in $G - (T \cup \pi(l))$ then
6.        return ExtendTree($G, T; \pi(l_1), \ldots, \pi(l_p), \pi(l); k$);
7.1  else $T'$ =ExtendTree($G, T; \pi(l_1), \ldots, \pi(l_p), \pi(l); k$);
         if $T'$ is not 'No'  then return $T'$;
7.2      let $y_1$, …,$y_q$ be the out-neighbors of $x_s$ in $G - (T \cup \pi(l))$, return
     ExtendTree($G, T + \pi(l) + x_s y_1 + \ldots + x_s y_q; \pi_{y_1,\ldots,y_q}(l_1), \ldots, \pi_{y_1,\ldots,y_q}(l_p); k$)
```

Fig. 11. Algorithm for the EXTENDING MAX-LEAF problem

Theorem D.1 *For a given out-tree T in a digraph G of n vertices, with a consistent set $\{\pi(l_1), \ldots, \pi(l_p)\}$ of out-chains for T, the algorithm ExtendTree runs in time $O^*(4^k)$ and either constructs an extended out-branching for $(T; \pi(l_1), \ldots, \pi(l_p); k)$ if such one exists, or reports 'No' otherwise.*

PROOF. We first verify the correctness of the algorithm.

If the input T is already an out-branching with less than k leaves, then there is no extended out-branching for $(T; \pi(l_1), \ldots, \pi(l_p); k)$. So step 1 reports 'No' correctly. If T is a k-out-tree, then by Theorem C.11, the algorithm SpanTree either constructs an extended out-branching for $(T; \pi(l_1), \ldots, \pi(l_p); k)$ if such one exists, or reports 'No' otherwise. Therefore, step 2 is correct. In step 3, x is an internal vertex of T. By Lemma C.8, the instance $(T; \pi(l_1), \ldots, \pi(l_p); k)$ is reducible to the instance $(T + xy_1 + \ldots + xy_q); \pi_{y_1,\ldots,y_q}(l_1), \ldots, \pi_{y_1,\ldots,y_q}(l_p); k)$. Thus, step 3 is correct.

After step 3, only leaves of T may have out-neighbors in $G - T$ and T has fewer than k leaves. Suppose that T_s is an extended out-branching for $(T; \pi(l_1), \ldots, \pi(l_p); k)$. Then any determined leaf l_i of T can only reach vertices of $\pi(l_i)$ in T_s. By the definition of out-chains, l_i can only reach vertices in $\pi(l_i)$ one by one in T_s. So l_i can only reach one leaf in T_s. If all leaves of T are determined leaves, then T_s has fewer than k leaves since T has fewer than k leaves. This contradicts that T_s is an extended out-branching for $(T; \pi(l_1), \ldots, \pi(l_p); k)$. Therefore, step 4 returns 'No' correctly.

After step 4, T has fewer than k leaves and some leaves of T are not determined leaves. Step 5 constructs an out-chain $\pi(l)$ for an undetermined leaf l such that the end vertex x_s of $\pi(l)$ either has at least two out-neighbors or has no out-neighbors in $G - (T \cup \pi(l))$. Note that $\pi(l)$ can be $\{l\}$. Moreover, for any $1 \leq i \leq p$, if $\pi(l_i)$ and $\pi(l)$ have a common vertex z, then the path from z to the last vertex w of $\pi(l_i)$ must be in $\pi(l)$.

Otherwise, let t be the first vertex in the path from z to w in $\pi(l_i)$ which is not in $\pi(l)$, and let p be the in-neighbour of t in $\pi(l)$. Since z is in $\pi(l)$, $t \neq z$ and since $t \neq l$, p is in $\pi(l)$. Since pt is in $\pi(l_i)$, p has only an out-neighbour t not in T, the out-tree for which we construct $\pi(l_i)$. Algorithm ExtendTree does remove edges from T, so T is a subgraph of the out-tree T', for which $\pi(l)$ is constructed. Thus, p still has at most one out-neighbour t not in T'. If p has t as the only out-neighbour not in T', then t should be added into $\pi(l)$ in step 5, contradicting the assumption that t is not in $\pi(l)$. If p has no out-neighbours not in T', then t is in T'. Only steps 3 and 7.2 add to T' vertices not in T. Both steps would change $\pi(l_i)$ to $\pi_t(l_i)$, which should not contain t. This contradicts that $\pi(l_i)$ contains t. In consequence, the path from z to the last vertex w of $\pi(l_i)$ must be in $\pi(l)$. Therefore, the set $\{\pi(l_1), \ldots, \pi(l_p), \pi(l)\}$ of out-chains is consistent.

If the last vertex x_s of $\pi(l)$ has no out-neighbor in $G - (T \cup \pi(l))$, then all vertices reachable from l are in $\pi(l)$ for all possible extended out-branching for $(T; \pi(l_1), \ldots, \pi(l_p); k)$. This means that $(T; \pi(l_1), \ldots, \pi(l_p), \pi(l); k)$ is the only instance to solve. So step 6 is correct.

Now suppose x_s has more than one out-neighbors in $G - (T \cup \pi(l))$. If there is an extended out-branching T_s for $(T; \pi(l_1), \ldots, \pi(l_p), \pi(l); k)$, then T_s is also a $(T; \pi(l_1), \ldots, \pi(l_p); k)$-extended out-branching. So step 7.1 is correct. If step 7.1 can not find an out-branching for $(T; \pi(l_1), \ldots, \pi(l_p), \pi(l); k)$, then for any possible extended out-branching T_s for $(T; \pi(l_1), \ldots, \pi(l_p); k)$, the vertices reachable in T_s from l must contain some vertex x_s not in $\pi(l)$. This can only happen when x_s is reachable from l and x_s is not a leaf in T_s, i.e, x_s is an internal vertex of T_s. Then all vertices in $\pi(l)$ should be reachable from l in T_s. By Lemma C.8, $(T; \pi(l_1), \ldots, \pi(l_p); k)$ is reducible to $(T + \pi(l) + x_x y_1 + \ldots + x_s y_q); \pi_{y_1,\ldots,y_q}(l_1), \ldots, \pi_{y_1,\ldots,y_q}(l_p); k)$. Thus step 7.2 is correct.

Summarizing the above discussion, we conclude that the algorithm ExtendTree correctly solves the instance $(T; \pi(l_1), \ldots, \pi(l_p); k)$ of the EXTENDING MAX-LEAF problem.

Now we consider the complexity of the algorithm.

Let u the number of leaves of T, p be the number of determined leaves of T, then $a = k - u$ and $b = k - p$. Let $f(a, b) = a + b = 2k - u - p$ be the complexity of the algorithm ExtendTree. We use the search-tree method to analyze the time complexity. Step 1 takes time $O(n)$. Step 2 takes time $O(n^2)$ by Theorem C.11. Steps 3, 4, and 5 can be done in time polynomial of n.

At step 6, the leaf l becomes a determined leaf from a undetermined leaf. So $u' = u$ and $p' = p + 1$. Then the complexity changes from $f(a, b) = 2k - u - p$ to $f'(a', b') = 2k - u - p - 1 = f(a, b) - 1$. At step 7.1, the complexity is the same as step 6: $f(a_1, b_1) = 2k - u - p - 1 = f(a, b) - 1$. Since x_s has at least two out-neighbors in step 7.2, $u' = u + 1$ and $p' = p$. So the complexity of step 7.2 is $f(a_2, b_2) = 2k - u - p - 1 = f(a, b) - 1$. Since steps 7.1 and 7.2 are two branches, we have the recurrence equation: $f(a, b) = 2f(a, b) - 1$.

Initially, $a + b = 2k - u - p \leq 2k$. When either $a = k - u = 0$ or $b = k - p = 0$, T is a k-out-tree. By Theorem C.11, EXTENDING$(T; \pi(l_1), \ldots, \pi(l_p); k)$ can be solved in time of $O(n^2)$, i.e., no branches are needed when $a = 0$ or $b = 0$. Then when $a + b = 0$, no branches are needed for ExtendTree, i.e., $f(0, 0) = 1$. By $f(a, b) = 2f(a, b) - 1$, we have $f(a, b) \leq 2^{a+b} = O(4^k)$, which is an upper bound of the total number of leaves in the search-tree for the algorithm ExtendTree. Since steps 1, 2, and 4 are executed only once, steps 3 and 5 are executed at most $O(n)$ times from the root to a leaf in the search-tree, the depth of the search-tree is $O(n)$, and the running time is polynomial of n for each step, the total running time of SpanTree is $O^*(4^k)$. $\square$

Now we are ready to present our algorithm for the MAX-LEAF OUT-BRANCHING problem.

Theorem D.2 *There is an algorithm to solve the* MAX-LEAF OUT-BRANCHING *problem in time $O(n^4 4^k)$.*

PROOF. The algorithm for solving the MAX-LEAF OUT-BRANCHING problem is very simple: for each vertex $x \in G$, we call the algorithm ExtendTree$(G, T + xy_1 + \ldots + xy_q); \emptyset; k)$ where (1) T is the single vertex x; (2) y_1, …, y_q are all out-neighbors of x in G; and (3) $\emptyset$ means that all leaves of $T + xy_1 + \ldots + xy_q$ are not determined. If an extended out-branching T_s for $(T + xy_1 + \ldots + xy_q; \emptyset; k)$ is found for some x, then T_s is an k-out-branching of G; otherwise reports 'No'.

To prove that the algorithm is correct, we prove the following claim.

Claim. If there is an k-out-branching T_s of G whose root is r with out-neighbour $y_1, \ldots, y_q$, then ExtendTree$(G, T+xy_1+\ldots+xy_q; \emptyset; k)$ find an extended out-branching for $(T + xy_1 + \ldots + xy; \emptyset; k)$.

First r must have some out-neighbor z in T_s, and z must be in $\{y_1, \ldots, y_q\}$. Suppose that $z = y_1$. Then T_s is an extended out-branching for $(T + ry_1; \emptyset; k)$ since (1) T_s is a k-out-branching; (2) T_s and $T + ry_1$ have the same root r; (3) $T + ry_1$ is a subgraph of T_s; and (4) all vertices reachable in T_s from a determined leaf l_i of $T + ry_1$ are in $\pi(l_i)$ since there is no determined leaf in $T + ry_1$. Therefore, T_s is an extended out-branching for $(T + ry_1; \emptyset; k)$, i.e., we only need to solve the instance $(T + ry_1; \emptyset; k)$.

Now r is an internal vertex of $T + ry_1$, so the instance $(T + ry_1; \emptyset; k)$ is reducible to the instance $(T + ry_1 + \ldots + ry_q, \emptyset; k)$ by Lemma C.8. Since there is an extended out-branching for $(T + ry_1; \emptyset; k)$, there is an extended out-branching for $(T + ry_1 + \ldots + ry_q; \emptyset; k)$. Thus ExtendTree$(G, T + ry_1 + \ldots + ry_q; \emptyset; k)$ will find an extended out-branching for $(T + ry_1 + \ldots + ry; \emptyset; k)$. This proves the Claim.

If there is a k-out-branching T_s rooted at r for G, the algorithm above for the MAX-LEAF OUT-BRANCHING problem must call ExtendTree$(G, T + ry_1 + \ldots + ry_q); \emptyset; k)$, where y_1, …, y_q are the out-neighbors of the vertex r, since ExtendTree is executed on every vertex in G. By our claim, ExtendTree$(G, T + ry_1 + \ldots + ry_q; \emptyset; k)$ will find an extended out-branching for $(T + ry_1 + \ldots + ry; \emptyset; k)$, which is a k-out-branching by definition.

If no extended out-branching for $(T + xy_1 + \ldots + xy_q); \emptyset; k)$ is found for any $x \in G$, then there is no k-out-branching of G by our Claim. In summary, the algorithm for the MAX-LEAF OUT-BRANCHING problem is correct: it either constructs a k-out-branching if such one exists, or reports ‘No’ otherwise.

The running time of this algorithm is $O(n)$ times the time of ExtendTree, thus is $O^*(4^k)$. $\square$

E. Final Remarks

In this chapter, we presented an $O^*(4^k)$ time algorithm for the MAX-LEAF OUT-BRANCHING problem on digraphs, which significantly improves the previous best algorithm of running time $O^*(2^{O(k \log k)})$ for the problem. The algorithm can be applied directly to solve the simpler MAX-LEAF SPANNING-TREE problem on undirected graphs, which also gives an improvement over the previous best algorithm of running time $O^*(6.75^k)$ for the problem. The improvements are based on a deeper study of the combinatorial structures of digraphs that reveals further relationship between out-trees and out-branchings in digraphs, and on applications of a new algorithmic technique for the design and analysis of parameterized algorithms. In particular, the new algorithmic technique identifies indirected branching measures that bound the number of computational branches in a branch-and-search process that do not reduce the parameter value effectively.

Recently, Kneis et al. independently developed an algorithm of the same running time $O^*(4^k)$ [69]. Their algorithm and our algorithm use the same idea of extending an out-tree to an out-branching.

CHAPTER VII

SATISFIABILITY

In this chapter, we study the complexity of the well-known SATISFIABILITY problem with respect to the total length L of formulas. We present an exact algorithm of running time $O^*(1.0652^L)$ for the SATISFIABILITY problem, which improves the previous best algorithm of running time $O^*(1.0663^L)$ [80]. Our algorithm considers both the number of variables and their associated weights as the measures for the SATISFIABILITY problem. By considering such measures, we design reduction rules to design a simple algorithm for the SATISFIABILITY problem. Our result demonstrates that our measure-driven approach is also powerful for designing exact algorithms (without parameters) for NP-hard problems.

A. Introduction

The SATISFIABILITY problem (briefly, SAT: given a CNF Boolean formula, decide if the formula has a satisfying assignment) is perhaps the most famous and most extensively studied NP-complete problem. The problem requires a precise answer Yes/No, and approximation algorithms do not seem to help much. Given the NP-completeness of the problem [53], it has become natural to develop exponential time algorithms that solve the problem as fast as possible.

There are three popular parameters that have been used in measuring exponential time algorithms for the SAT problem: the number n of variables in the input formula, the number m of clauses in the input formula, and the *total length* L of the input formula, which is the sum of the clause lengths in the input formula. Note that the parameter L is probably the most precise parameter in terms of standard complexity theory, and both parameters n and m could be sublinear in instance length.

Algorithms for SAT in terms of each of these parameters have been extensively studied. See [55] for a comprehensive review and see [99] for more recent progress on the research in these directions.

In the current chapter, we are focused on algorithms for SAT in terms of the parameter L. The research started 20 years ago since the first published algorithm of time $O(1.0927^L)$ [54]. The upper bound was subsequently improved by an impressive list of publications. We summarize the major progress in the following table.

Table III. History of exact algorithms for the SATISFIABILITY problem

Reference	**Bound**	**Year Published**
Van Gelder [54]	1.0927^L	1988
Kullmann *et al.* [73]	1.0801^L	1997
Hirsh [62]	1.0758^L	1998
Hirsh [61]	1.074^L	2000
Wahlstom [80]	1.0663^L	2005

The branch-and-search method has been widely used in the development of SAT algorithms. Given a Boolean formula $\mathcal{F}$, let $\mathcal{F}[x]$ and $\mathcal{F}[\overline{x}]$ be the resulting formula after assigning TRUE and FALSE, respectively, to the variable x in the formula $\mathcal{F}$. The branch-and-search method is based on the fact that $\mathcal{F}$ is satisfiable if and only if at least one of $\mathcal{F}[x]$ and $\mathcal{F}[\overline{x}]$ is satisfiable. Most SAT algorithms are based on this method.

Unfortunately, analysis directly based on the parameter L usually does not give a good upper bound in terms of L for a branch-and-search SAT algorithm. Combinations of the parameter L and other parameters, such as the number n of variables in the input formula, have been used as "measures" in the analysis of SAT algorithms. For example, the measure $L - 2n$ [81] and a more general measure that is a function $f(L, n)$ of the parameters L and n [80] have been used in the analysis of SAT algorithms whose complexity is measured in terms of the parameter L.

In the current chapter, we introduce a new measure, the *l-value* of a Boolean formula $\mathcal{F}$. Roughly speaking, the measure l-value $l(\mathcal{F})$ is defined based on weighted variable frequencies in the input formula $\mathcal{F}$. We develop a branch-and-search algorithm that tries to maximize the decreasing rates in terms of the l-value during its branch-and-search process. In particular, by properly choosing the variable frequency weights so that the formula l-value is upper bounded by $L/2$, adopting new reduction rules, and applying the analysis technique of Measure and Conquer recently developed by Fomin et al. [50], we develop a new branch-and-search algorithm for the SAT problem whose running time is bounded by $O(1.1346^{l(\mathcal{F})})$ on an input formula $\mathcal{F}$. Finally, by combining this algorithm with the algorithm in [81] to deal with formulas of lower variable frequencies and converting the measure $l(\mathcal{F})$ into the parameter L, we achieve a SAT algorithm of running time $O(1.0652^L)$, improving the previously best SAT algorithm of running time $O(1.0663^L)$ [80].

We remark that although the analysis of our algorithm is lengthy, our algorithm itself is very simple and can be easily implemented. Note that the lengthy analysis needs to be done only once to ensure the correctness of the algorithm, while the simplicity of the algorithm gives its great advantage when it is applied (many times) to determine the satisfiability of CNF Boolean formulas.

B. Preliminaries

We introduce the notations and terminology that will be used in our discussion.

A (Boolean) *variable* x can be assigned value either 1 (TRUE) or 0 (FALSE). The variable x has two corresponding *literals* x and $\overline{x}$. The literal x is *satisfied* if $x = 1$ and the literal $\overline{x}$ is *satisfied* if $x = 0$. Note that exactly one of the literals x and $\overline{x}$

is satisfied for any value assignment to the variable x. A *clause* C is a disjunction of a set of literals, which can be regarded as a set of literals. Therefore, we may write $C_1 = zC_2$ to indicate that the clause C_1 consists of the literal z plus all literals in the clause C_2, and use C_1C_2 to denote the clause that consists of all literals that are in either C_1 or C_2, or both. The *length* of a clause C, denoted by $|C|$, is the number of literals in C. A clause C is *satisfied* if any literal in C is satisfied. A (CNF Boolean) *formula* $\mathcal{F}$ is a conjunction of clauses C_1 , …, C_m, which can be regarded as a collection of the clauses. The formula $\mathcal{F}$ is *satisfied* if all clauses in $\mathcal{F}$ are satisfied. The *length* L of the formula $\mathcal{F}$ is defined as $L = |C_1| + \cdots + |C_m|$.

A literal z is an *i-literal* if z is contained in exactly i clauses, and is an *i^+-literal* if z is contained in at least i clauses. An *(i, j)-literal* z is a literal such that exactly i clauses contain z and exactly j clauses contain $\overline{z}$. Note that an (i, j)-literal z implies that the literal $\overline{z}$ is a (j, i)-literal. We say that a clause C *contains a variable* x if C contains either the literal x or the literal $\overline{x}$. A variable x is an *i-variable* if exact i of the clauses contain the variable x (in this case, we also say that the *degree* of x is i). A clause C is an *i-clause* if $|C| = i$, and is an *i^+-clause* if $|C| \geq i$.

Let xC_1, …, xC_s be all the clauses in the input formula $\mathcal{F}$ that contain the literal x, and let $\overline{x}D_1$, …, $\overline{x}D_t$ be all the clauses in $\mathcal{F}$ that contain the literal $\overline{x}$. A *resolvent* on a variable x in $\mathcal{F}$ is a clause of the form C_iD_j for some i and j, $1 \leq i \leq s$ and $1 \leq j \leq t$. The *resolution* on the variable x in the formula $\mathcal{F}$, written as $DP_x(\mathcal{F})$, is a formula that is obtained by first removing all clauses containing the variable x from $\mathcal{F}$ and then adding all possible resolvents on the variable x into $\mathcal{F}$.

A *branching vector* is a tuple of positive real numbers. A branching vector $t = (t_1, \ldots, t_r)$ corresponds to a polynomial $1 - \sum_{i=1}^{r} x^{-t_i}$, which has a unique positive root $\tau(t)$ [20]. We say that a branching vector t' is *inferior* to a branching vector t'' if $\tau(t') \geq \tau(t'')$. In particular, if either $t'_1 \leq t''_1$ and $t'_2 \leq t''_2$, or $t'_1 \leq t''_2$ and $t'_2 \leq t''_1$, then it can be proved [20] that the branching vector $t' = (t'_1, t'_2)$ is inferior to the branching vector $t'' = (t''_1, t''_2)$.

The execution of a SAT algorithm based on the branch-and-search method can be represented as a search tree $\mathcal{T}$ whose root is labeled by the input formula $\mathcal{F}$. Recursively, if at a node w_0 labeled by a formula $\mathcal{F}_0$ in the search tree $\mathcal{T}$, the algorithm breaks $\mathcal{F}_0$, in polynomial time, into r smaller formulas $\mathcal{F}_1$, …, $\mathcal{F}_r$, and recursively works on these smaller formulas, then the node w_0 in $\mathcal{T}$ has r children, labeled by $\mathcal{F}_1$, …, $\mathcal{F}_r$, respectively. Suppose that we use a measure $\mu(\mathcal{F})$ for a formula $\mathcal{F}$, then the branching vector for this branching, with respect to the measure μ, is $t = (t_1, \ldots, t_r)$, where $t_i = \mu(\mathcal{F}_0) - \mu(\mathcal{F}_i)$ for all i. Finally, suppose that t' is a branching vector that is inferior to all branching vectors for any branching in the search tree $\mathcal{T}$, then the complexity of the SAT algorithm is bounded by $O(\tau(t')^{\mu(\mathcal{F})})$ times a polynomial of L [20].

Formally, for a given input formula $\mathcal{F}$, we define the *l-value* for $\mathcal{F}$ to be $l(\mathcal{F}) = \sum_{i\geq 1} w_i n_i$, where for all $i \geq 1$, n_i is the number of i-variables in $\mathcal{F}$, and the *frequency weight* w_i for i-variables are set by the following values:

$$\begin{array}{lll} w_0 = 0; & w_1 = 0.32; & w_2 = 0.45 \\ w_3 = 0.997, & w_4 = 1.897, & w_i = i/2, \quad \text{for } i \geq 5. \end{array} \tag{7.1}$$

Define $\delta_i = w_i - w_{i-1}$, for $i \geq 1$. Then we can easily verify that

$$\begin{aligned} &\delta_i \geq 0.5, \quad \text{for all } i \geq 3, \\ &\delta_{\min} = \min\{\delta_i \mid i \geq 1\} = \delta_2 = 0.13, \\ &\delta_{\max} = \max\{\delta_i \mid i \geq 1\} = \delta_4 = 0.9. \end{aligned} \tag{7.2}$$

Note that the length L of the formula $\mathcal{F}$ is equal to $\sum_{i\geq 1} i \cdot n_i$, and that $i/5 \leq w_i \leq i/2$

for all i. Therefore, we have $L/5 \leq l(\mathcal{F}) \leq L/2$.

Given two formulas $\mathcal{F}_1$ and $\mathcal{F}_2$, by definition we have $l(\mathcal{F}_1) = \sum_{x \in \mathcal{F}_1} w(x)$ and $l(\mathcal{F}_2) = \sum_{x \in \mathcal{F}_2} w'(x)$, where $w(x)$ is the frequency weight of x in $\mathcal{F}_1$ and $w'(x)$ is the frequency weight of x in $\mathcal{F}_2$. The *l-value reduction* from $\mathcal{F}_1$ to $\mathcal{F}_2$ is $l(\mathcal{F}_1) - l(\mathcal{F}_2)$. The *contribution* of x to the l-value reduction from $\mathcal{F}_1$ to $\mathcal{F}_2$ is $w(x) - w'(x)$. The contribution of a variable set S to the l-value reduction from $\mathcal{F}_1$ to $\mathcal{F}_2$ is the summation of contributions of all variables in S.

C. Reduction Rules

We say that two formulas $\mathcal{F}_1$ and $\mathcal{F}_2$ are *equivalent* if $\mathcal{F}_1$ is satisfiable if and only if $\mathcal{F}_2$ is satisfiable. A literal z in a formula $\mathcal{F}$ is *monotone* if the literal $\overline{z}$ does not appear in $\mathcal{F}$.

We present in this section a set of reduction rules that reduce a given formula $\mathcal{F}$ to an equivalent formula $\mathcal{F}'$ without increasing the l-value. Consider the algorithm given in Figure 12.

Lemma C.1 *The algorithm* **Reduction**$(\mathcal{F}_1)$ *produces a formula equivalent to the formula* $\mathcal{F}_1$*.*

Proof. It suffices to prove that in each of the listed cases, the algorithm **Reduction** on the formula $\mathcal{F}_1$ produces an equivalent formula $\mathcal{F}_2$. This can be easily verified for Cases 1, 2, 3, and 5.

The claim holds true for Cases 4 and 9 from the resolution principle [31].

In Case 6, the clause $z_1\overline{z_2}C$ in $\mathcal{F}_1$ is replaced with the clause z_1C in $\mathcal{F}_2$. If an assignment A_2 satisfies $\mathcal{F}_2$, then obviously A_2 also satisfies $\mathcal{F}_1$. On the other hand, if an assignment A_1 satisfies $\mathcal{F}_1$ but does not satisfy the clause z_1C in $\mathcal{F}_2$, then because of the clause z_1z_2 in $\mathcal{F}_1$, we must have $z_1 = 0$ and $z_2 = 1$. Since A_1 satisfies the clause $z_1\overline{z_2}C$ in $\mathcal{F}_1$, this would derive a contradiction that A_1 must satisfy z_1C. Therefore, A_1 must also satisfy the formula $\mathcal{F}_2$.

In Case 7, the clause $z_1z_2C_1$ in $\mathcal{F}_1$ is replaced with the clause z_2C_1 in $\mathcal{F}_2$. Again, the satisfiability of $\mathcal{F}_2$ trivially implies the satisfiability of $\mathcal{F}_1$. For the other direction, let z_2C_3 be the third clause that contains z_2 (note that z_2 is a $(2,1)$-literal). If an assignment A_1 satisfies $\mathcal{F}_1$ (in particular satisfies the clause $z_1z_2C_1$) but not $\mathcal{F}_2$ (i.e., not the clause z_2C_1), then we must have $z_1 = 1$, $z_2 = 0$, and $C_3 = 1$ under A_1. By replacing the assignment $z_2 = 0$ with $z_2 = 1$ in A_1, we will obtain an assignment A_2' that satisfies all z_2C_1, $z_1\overline{z_2}C_2$, and z_2C_3, thus satisfies the formula $\mathcal{F}_2$.

In Case 8, the formula $\mathcal{F}_2$ is obtained from the formula $\mathcal{F}_1$ by removing the clause z_1z_2. Thus, the satisfiability of $\mathcal{F}_1$ trivially implies the satisfiability of $\mathcal{F}_2$. On the other hand, suppose that an assignment A_2 satisfying $\mathcal{F}_2$ does not satisfy $\mathcal{F}_1$ (i.e., does not satisfy the clause z_1z_2). Then A_2 must assign $z_1 = 0$ and $z_2 = 0$. We can simply replace $z_1 = 0$ with $z_1 = 1$ in A_2 and keep the assignment satisfying $\mathcal{F}_2$: this is because $\overline{z_1z_2}C$ is the only clause in $\mathcal{F}_2$ that contains $\overline{z_1}$. Moreover, the new assignment now also satisfies z_1z_2, thus satisfies the formula $\mathcal{F}_1$.

For Case 10, it is suffice to show that we can always set $z_1 = \overline{z_2}$ in a satisfying assignment for the formula $\mathcal{F}_1$. For the subcase where z_1 is a 1-literal and z_1z_2 is a 2-clause, if a satisfying assignment A_1 for $\mathcal{F}_1$ assigns $z_2 = 1$ then we can simply let $z_1 = \overline{z_2} = 0$ since z_1 is only contained in the clause z_1z_2. If A_1 assigns $z_2 = 0$ then because of the 2-clause z_1z_2, A_1 must assign $z_1 = \overline{z_2} = 1$. For the other subcase, note that the existence of the 2-clauses z_1z_2 and $\overline{z_1z_2}$ in the formula $\mathcal{F}_1$ trivially requires that every assignment satisfying $\mathcal{F}_1$ have $z_1 = \overline{z_2}$.

Finally, in Case 11, the clauses CD_1 and CD_2 in $\mathcal{F}_1$ are replaced with the clauses $\overline{x}C$, xD_1, and xD_2 in $\mathcal{F}_2$. If an assignment A_1 satisfies $\mathcal{F}_1$ (thus satisfies CD_1 and

CD_2), then in case $C = 0$ under A_1 we assign the new variable $x = 0$, and in case $C = 1$ under A_1 we assign the new variable $x = 1$. It is not hard to verify that this assignment to the new variable x plus A_1 will satisfy $\mathcal{F}_2$. For the other direction, suppose that an assignment A_2 satisfies $\mathcal{F}_2$. If A_2 assigns $x = 1$ then we have $C = 1$ under A_2 thus the assignment A_2 also satisfies CD_1 and CD_2 thus $\mathcal{F}_1$; and if A_2 assigns $x = 0$ then we have $D_1 = 1$ and $D_2 = 1$ under A_2 and again A_2 satisfies $\mathcal{F}_1$.

□

Next, we show that the algorithm **Reduction** always decreases the l-value.

Lemma C.2 *Let $\mathcal{F}_1$ and $\mathcal{F}_2$ be two formulas such that $\mathcal{F}_2 =$ **Reduction**$(\mathcal{F}_1)$, and $\mathcal{F}_1 \neq \mathcal{F}_2$. Then $l(\mathcal{F}_1) \geq l(\mathcal{F}_2) + 0.003$.*

PROOF. Since $\mathcal{F}_1 \neq \mathcal{F}_2$, at least one of the cases in the algorithm **Reduction** is applicable to the formula $\mathcal{F}_1$. Therefore, it suffices to verify that each case in the algorithm **Reduction** decreases the l-value by at least 0.003.

Cases 1-8 simply remove certain literals, which decrease the degree of certain variables in the formula. Therefore, if any of these cases is applied on the formula $\mathcal{F}_1$, then the l-value of the formula is decreased by at least $\delta_{\min} = \delta_2 = 0.13$.

Consider Cases 9-11. Note that if we reach these cases then Cases 4-5 are not applicable, which implies that the formula $\mathcal{F}_1$ contains only 3^+-variables.

Case 9. If both $\overline{z}_1$ and $\overline{z}_2$ are in the same clause, say C_1, then the resolution $DP_x(\mathcal{F})$ after the next application of Case 2 in the algorithm will replace the four clauses $\overline{x}z_1$, $\overline{x}z_2$, xC_1, and xC_2 with two 3-clauses z_1C_2 and z_2C_2, which decreases the l-value by w_4 (because of removing the 4-variable x) and on the other hand increases the l-value by at most $2\delta_{\max}$ (because of increasing the degree of the two variables in C_2). Therefore, in this case, the l-value is decreased by at least $w_4 - 2\delta_{\max} =$

$w_4 - 2\delta_4 = 0.097$. If $\overline{z}_1$ and $\overline{z}_2$ are not in the same clause of C_1 and C_2, say $\overline{z}_1$ is in C_1 and $\overline{z}_2$ is in C_2, then the resolution $DP_x(\mathcal{F})$ after the next application of Case 2 in the algorithm will replace the four clauses $\overline{x}z_1$, $\overline{x}z_2$, xC_1, and xC_2 with two 3-clauses z_1C_2 and z_2C_1. In this case, the l-value is decreased by exactly $w_4 = 1.897$ because of removing the 4-variable x.

Case 10. Suppose that z_1 is an i-variable and z_2 is a j-variable. Replacing z_1 by $\overline{z}_2$ removes the i-variable z_1 and makes the j-variable z_2 into an $(i+j)$-variable. However, after an application of Case 2 in the algorithm, the clause z_1z_2 in the original formula disappears, thus z_2 becomes an $(i+j-2)$-variable. Therefore, the total value decreased in the l-value is $(w_i + w_j) - w_{i+j-2}$. Because of the symmetry, we can assume without loss of generality that $i \leq j$. Note that we always have $i \geq 3$. If $i = 3$, then $w_3 + w_j = \delta_{\max} + 0.097 + w_j \geq w_{j+1} + 0.097 = w_{3+j-2} + 0.097$. If $i = 4$, then $w_4 + w_j = 2\delta_{\max} + w_j + 0.097 \geq w_{j+2} + 0.097 = w_{4+j-2} + 0.097$. If $i \geq 5$, then $w_i + w_j = i/2 + j/2 = (i + j - 2)/2 + 1 = w_{i+j-2} + 1$. Therefore, in this case, the l-value of the formula is decreased by $(w_i + w_j) - w_{i+j-2}$, which is at least 0.097.

Case 11. Since the clauses CD_1 and CD_2 in $\mathcal{F}_1$ are replaced with $\overline{x}C$, xD_1 and xD_2, each variable in C has its degree decreased by 1. Since all variables in $\mathcal{F}_1$ are 3^+-variables and $|C| \geq 2$, the degree decrease for the variables in C makes the l-value to decrease by at least $2 \cdot \min\{\delta_i \mid i \geq 3\} = 1$. On the other hand, the introduction of the new 3-variable x and the new clauses $\overline{x}C$, xD_1 and xD_2 increases the l-value by exactly $w_3 = 0.997$. In consequence, the total l-value in this case is decreased by at least $1 - 0.997 = 0.003$. □ □

By definition, the l-value $l(\mathcal{F}_1)$ of the formula $\mathcal{F}_1$ is bounded by $L/2$, where L is the length L of the formula $\mathcal{F}_1$. By the proof of Lemma C.2, each application of a case in the algorithm **Reduction** takes time polynomial in L and decreases the l-value by at least a constant. Therefore, the algorithm **Reduction** must stop in polynomial time and produce an equivalent formula $\mathcal{F}_2$ for which no cases in the algorithm **Reduction** are applicable. Such a formula $\mathcal{F}_2$ will be called a *reduced formula.* Reduced formulas have a number of interesting properties, which are given in the following lemmas.

Lemma C.3 *There are no* 1*-variables or* 2*-variables in a reduced formula.*

PROOF. Each 1-variable makes a monotone literal. Each 2-variable either makes a monotone literal or has at most one non-trivial resolvent. Therefore, the Cases 4-5 in the algorithm **Reduction** ensure that a reduced formula contains no 1-variables and 2-variables. □

Lemma C.4 *Let* $\mathcal{F}$ *be a reduced formula and* xy *be a clause in* $\mathcal{F}$*. Then*

(1) *No other clauses contain* xy*;*

(2) *No clause contains* $x\overline{y}$ *or* $\overline{x}y$*;*

(3) *At most one* 3^+*-clause contains* $\overline{xy}$*. Moreover, if* y *is a* 3*-variable or* x *is a 1-literal, then no clause contains* $\overline{xy}$*.*

PROOF. (1) Since Case 1 is not applicable, there is no other clauses containing xy.

(2) Since Case 6 is not applicable, no clause in $\mathcal{F}$ can contain either $x\overline{y}$ or $\overline{x}y$.

(3) Given a clause $\overline{xy}C$, the clause C cannot be empty since Case 10 is not applicable. If there are two 3^+-clauses containing $\overline{xy}$, Case 11 would be applicable. Therefore, there is at most one 3^+-clause that contains $\overline{xy}$.

If y is a 3-variable, then $\overline{xy}$ can not be in any clause. Otherwise, the resolution on y would have at most one non-trivial resolvent, and Case 4 would be applicable to $\mathcal{F}$. If $\overline{x}$ is a 1-literal, then $\overline{xy}$ can not be in any clause, since Case 8 is no applicable to $\mathcal{F}$. □

Lemma C.5 *Let $\mathcal{F}$ be a reduced formula. Both x and y are variables in $\mathcal{F}$. If only 3^+-clauses contain both variables x and y, then for any of xy, $\overline{y}$, $\overline{y}x$, and $\overline{xy}$, there is at most one clause containing it. Moreover, if y is a 3-variable, then only $\overline{x}y$ can be contained in a 3^+-clause.*

Proof. Since $\mathcal{F}$ is a reduced formula, Case 11 is not applicable to $\mathcal{F}$. Then for any of xy, $\overline{y}$, $\overline{y}x$, and $\overline{xy}$, there is at most one clause containing it.

If y is a 3-variable, then $\overline{xy}$ can not occur in any clause. Otherwise, the resolution on y would have at most one non-trivial resolvent, and Case 4 would be applicable to $\mathcal{F}$. If $x\overline{y}$ exists, Case 7 would be applicable to $\mathcal{F}$. □

D. Main Algorithm

Our main algorithm for the SAT problem is given in Figure 13. The *degree* $d(\mathcal{F})$ of a formula $\mathcal{F}$ is defined to be the largest degree of a variable in the formula $\mathcal{F}$. Let $\mathcal{F}[x]$ be the resulting formula after removing literal $\overline{x}$ and all clauses containing literal x from $\mathcal{F}$. Similarly, $\mathcal{F}[\overline{x}]$ is the resulting formula after removing literal x and all clauses containing literal $\overline{x}$.

Theorem D.1 *The algorithm* **SATSolver**$(\mathcal{F})$ *solves the SAT problem in time $O(1.0652^L)$, where L is total length of the input formula $\mathcal{F}$.*

Proof. It is clear that the algorithm **SATSolver**$(\mathcal{F})$ solves the SAT problem. When the degree of $\mathcal{F}$ is 3, we just apply the algorithm by Wahlström [81] at step 4 in the algorithm **SATSolver**$(\mathcal{F})$. The running time of Wahlström's algorithm is $O(1.1279^n)$, where n is the number of variables in $\mathcal{F}$ [81], which is also $O(1.1346^{l(\mathcal{F})})$ since $l(\mathcal{F}) = w_3 n$. The proof that the algorithm **SATSolver**$(\mathcal{F})$ runs in time $O(1.1346^{l(\mathcal{F})})$ when the degree of $\mathcal{F}$ is greater than 3 is given in the next section. The equality $O(1.1346^{l(\mathcal{F})}) = O(1.0652^{L})$ is because $l(\mathcal{F}) \leq L/2$. □

E. Analysis of the Main Algorithm

Given a formula $\mathcal{F}$, let *reduced*$(\mathcal{F})$ be the output formula of **Reduction**$(\mathcal{F})$, and *reduced*$_p(\mathcal{F})$ be the first formula during the execution of **Reduction**$(\mathcal{F})$ such that Cases 1-8 are not applicable to the formula. Next we discuss the relationship among the l-value of $\mathcal{F}$, reduced$_p(\mathcal{F})$, and reduced$(\mathcal{F})$.

Lemma E.1 $l(\mathcal{F}) \geq l(\mathit{reduced}_p(\mathcal{F})) \geq l(\mathit{reduced}(\mathcal{F}))$.

Proof. Note that reduced$(\mathcal{F})$=**Reduction**(reduced$_p(\mathcal{F})$). Thus, by lemma C.2, we have $l(\text{reduced}(\mathcal{F})) \leq l(\text{reduced}_p(\mathcal{F}))$. As shown in the proof of lemma C.2, every case in the algorithm **SATSolver**$(\mathcal{F})$ decrease the l-value of its input formula. So $l(\mathcal{F}) \geq l(\text{reduced}_p(\mathcal{F}))$. □

The reason we consider reduced$_p(\mathcal{F})$ is that it is easy to give a bound on the l-value reduction from $\mathcal{F}$ to reduced$_p(\mathcal{F})$.

Lemma E.2 *The contributions of any subset of variables is not more than the l-value reduction from $\mathcal{F}$ to reduced$_p(\mathcal{F})$.*

Proof. Let V be the set of all variables in $\mathcal{F}$. Note that $l(\mathcal{F}) = \sum_{x \in V} w(x)$ and $l(\text{reduced}_p(\mathcal{F})) = \sum_{x \in V} w'(x)$, where $w(x)$ is the frequency weight of x in $\mathcal{F}$ and $w'(x)$ is the frequency weight of x in reduced$_p(\mathcal{F})$. By the definition of reduced$_p(\mathcal{F})$, only Cases 1-8 have been applied before we get *reduced*$_p(\mathcal{F})$. Since Cases 1-8 do not introduce new variables, we have

$$l(\mathcal{F}) - l(\text{reduced}_p(\mathcal{F})) = \sum_{x \in V}(w(x) - w'(x)).$$

Therefore, the contribution of V equals to the l-value reduction from $\mathcal{F}$ to reduced$_p(\mathcal{F})$. Since Cases 1-8 do not increase the degree of any variable, the contribution of any variable is not negative. Thus the contribution of any subset of variables is not more than the contribution of V, i.e., the l-value reduction from $\mathcal{F}$ to reduced$_p(\mathcal{F})$. □

From now on in this section, let $\mathcal{F}$ be the formula after step 1 in the algorithm **SATSolver**($\mathcal{F}$). Then $\mathcal{F}$ is a reduced formula. In the algorithm **SATSolver**($\mathcal{F}$), we break $\mathcal{F}$ into two formulas $\mathcal{F}_1$ and $\mathcal{F}_2$ of smaller l-values at step 3 or 4. To give better bound, we are interested in the branching vector from $\mathcal{F}$ to reduced($\mathcal{F}_1$) and reduced($\mathcal{F}_2$), instead of the branching vector from $\mathcal{F}$ to $\mathcal{F}_1$ and $\mathcal{F}_2$. To give feasible analysis, we focus on the branching vector from $\mathcal{F}$ to reduced$_p(\mathcal{F}_1)$ and reduced$_p(\mathcal{F}_2)$. This is correct by the following lemma.

Lemma E.3 *The branching vector from $\mathcal{F}$ to reduced$_p(\mathcal{F}_1)$ and reduced$_p(\mathcal{F}_2)$ is inferior to the branching vector from $\mathcal{F}$ to reduced($\mathcal{F}_1$) and reduced($\mathcal{F}_2$).*

PROOF. By lemma E.1, $l(\text{reduced}_p(\mathcal{F}_1)) \le l(\text{reduced}_p(\mathcal{F}_1))$ and $l(\text{reduced}_p(\mathcal{F}_2)) \le l(\text{reduced}_p(\mathcal{F}_2))$. Thus,

$$l(\mathcal{F}) - l(\text{reduced}(\mathcal{F}_1)) \ge l(\mathcal{F}) - l(\text{reduced}_p(\mathcal{F}_1)), \quad \text{and}$$
$$l(\mathcal{F}) - l(\text{reduced}(\mathcal{F}_2)) \ge l(\mathcal{F}) - l(\text{reduced}_p(\mathcal{F}_2)).$$

Note that $(l(\mathcal{F}) - l(\text{reduced}(\mathcal{F}_1)), l(\mathcal{F}) - l(\text{reduced}(\mathcal{F}_2)))$ is the branching vector from $\mathcal{F}$ to reduced($\mathcal{F}_1$) and reduced($\mathcal{F}_2$), and $(l(\mathcal{F}) - l(\text{reduced}_p(\mathcal{F}_1)), l(\mathcal{F}) - l(\text{reduced}_p(\mathcal{F}_2)))$ is the branching vector from $\mathcal{F}$ to reduced$_p(\mathcal{F}_1)$ and reduced$_p(\mathcal{F}_2)$. By definition, the branching vector from $\mathcal{F}$ to reduced$_p(\mathcal{F}_1)$ and reduced$_p(\mathcal{F}_2)$ is inferior to the branching vector from $\mathcal{F}$ to reduced$(\mathcal{F}_1)$ and reduced$(\mathcal{F}_2)$. $\square$

Most of the time in the algorithm **SATSolver**$(\mathcal{F})$, we break $\mathcal{F}$ into $\mathcal{F}[x]$ and $\mathcal{F}[\overline{x}]$ where $\mathcal{F}$ is of degree i and x is an i-variable in $\mathcal{F}$. Let y be a variable contained with variable x in a clause. We can bound the contribution of y to the l-value reduction from $\mathcal{F}$ to reduced$_p(\mathcal{F}[x])$.

Lemma E.4 *Let y be an i-variable. The contribution of y to the l-value reduction from $\mathcal{F}$ to reduced$_p(\mathcal{F}[x])$ is at least*

(1) w_3, if $i = 3$ and y is in a clause with literal x;

(2) w_i, if y is contained with literal $\overline{y}$ in a 2-clause;

(3) δ_i, if $i > 3$ and y is only in one clause with literal x, or $2\delta_i$ if $i > 3$ and y is in more than one clause with literal x.

Proof. (1) The clause containing literal x and y is not in $\mathcal{F}[x]$. Thus y is of degree at most 2 in $\mathcal{F}[x]$, and is not in reduced$_p(\mathcal{F}[x])$ (same proof as in lemma C.3). So the contribution of y is w_3.

(2) y is a 1-clause in $\mathcal{F}[x]$. y is not in reduced$_p(\mathcal{F}[x])$ (same proof as in lemma C.3). Thus the contribution of y is w_i.

(3) If $i > 3$ and y is only in one clause with literal x, then y is at most a $(i-1)$-variable in reduced$_p(\mathcal{F}[x])$. Thus the contribution of y is at least δ_i.

If $i > 3$ and y is in more than one clause with literal x, then y and literal x are in exactly two clauses in $\mathcal{F}$: xyC_0 and $x\overline{y}C_1$. Otherwise, Case 9 would be applicable to $\mathcal{F}$, which contradicts that $\mathcal{F}$ is a reduced formula. So the contribution of y is at least $\delta_i + \delta_{i-1} \geq 2\delta_i$ when $i > 3$. $\square$

Let S be the set of variables which are contained with variable x in some clause. We do not include x in S. Let x be an i-variable in $\mathcal{F}$. Then we can bound the l-value reduction from $\mathcal{F}$ to reduced$_p(\mathcal{F}[x])$ with the following calculations:

Step 1: set $c_x = w_i$ and $c_y = 0$ for $y \in S$.

Step 2: for each 2-clause $\overline{x}y$ or $\overline{xy}$,

(1) when $i = 3$, add w_3 to c_y,

(2) when $i > 3$, add $w_i - \delta_i$ to c_y if there is a clause xC containing variable y, or add w_i to c_y otherwise.

Step 3: for each clause xyC where $y \in S$,

(1) when $i = 3$, add w_3 to c_y,

(2) when $i > 3$, add δ_i to c_y.

Step 4: $c = c_x + \sum_{y \in S} c_y$.

The value c calculated above is the *c-value* from $\mathcal{F}$ to reduced$_p(\mathcal{F}[x])$. The c-value from $\mathcal{F}$ to reduced$_p(\mathcal{F}[\overline{x}])$ can be calculated similarly. Next we show that the c-value is not larger than the l-value reduction from $\mathcal{F}$ to reduced$_p(\mathcal{F})$.

Lemma E.5 *The c-value is not larger than the contribution of x to the l-value reduction from $\mathcal{F}$ to reduced$_p(\mathcal{F}[x])$.*

Proof. By lemma E.2, the contribution of $S + x$ is not larger than the l-value reduction from $\mathcal{F}$ to reduced$_p(\mathcal{F}[x])$. We complete our proof by showing that c is not larger than the contribution of $S + x$. Since x is an i-variable in $\mathcal{F}$ and x is not in reduced$_p(\mathcal{F}[x])$, the contribution of x from $\mathcal{F}$ to $\mathcal{F}[x]$ is w_i. Note that $c = c_x + \sum_{y \in S} c_y$ at step 4 and $c_x = w_y$ at step 1. So we only need to show that c_y is not larger than its contribution from $\mathcal{F}$ to reduced$_p(\mathcal{F}[x])$ for all $y \in S$.

At step 1, c_y is initialized to be 0.

At step 2, we only change the c_y of variable y which is in 2-clause $\overline{x}y$ or $\overline{x}\,\overline{y}$. For such a variable y, there is exactly one clause containing both variable y and literal $\overline{x}$. Then c_y is either w_i or $w_i - \delta_i$ after step 2. Moreover, by the rule (2) of lemma E.4, the contribution of y from $\mathcal{F}$ to reduced$_{(}\mathcal{F}[x])$ is w_i. So after step2, c_y is not larger than the contribution of y from $\mathcal{F}$ to reduced$_p(\mathcal{F}[x])$.

At step 3, we only change the c_y of variable y which is in a clause xC where literal y or $\overline{y}$ is in C. We consider two cases at step 3.

Case 1: When y is a 3-variable, there is no clause $\overline{x}y$ or $\overline{x}\,\overline{y}$. So c_y is 0 after step 2, and is w_3 after step 3. The contribution of y is at least w_3 by lemma E.4. Thus c_y is not larger than its contribution from $\mathcal{F}$ to reduced$_p(\mathcal{F}[x])$.

Case 2: When y is an i-variable where $i > 3$.

If there is a 2-clause $\overline{x}y$ or $\overline{x}\,\overline{y}$, then c_y is $w_i - \delta_i$ after step 2 as shown for step 2, and there is exactly one clause xC containing variable y and literal x by lemma C.4. Thus c_y is w_i after step 3. By lemma E.4, the contribution of y from $\mathcal{F}$ to reduced$_p(\mathcal{F}[x])$ is at least w_i. So c_y is not larger than the contribution of y after step 3.

If there is no 2-clause $\overline{x}y$ or $\overline{x}\,\overline{y}$, then c_y is 0 after step 2. For such a variable y, there are at most two clauses xC_0 and xC_1 containing variable y, since $\mathcal{F}$ is a reduced formula, and since Cases 1 and 9 are not applicable to $\mathcal{F}$. Thus c_y is not larger than $2\delta_i$ after step 3, which is not larger than the contribution of y by lemma E.4.

For both cases, we can conclude that c_y is not larger than its contribution from $\mathcal{F}$ to reduced$_p(\mathcal{F}[x])$ for all $y \in S$. Then we complete our proof of this lemma. $\square$

To give better analysis, some notations are needed.

n_1: the number of 3^+-clauses containing literal x.

n_3: the number of 2-clauses containing 3-variables and literal x.

n_4: the number of 2-clauses containing 4-variables and literal x.

n_5: the number of 2-clauses containing 5^+-variables and literal x.

$\overline{n_1}$: the number of 3^+-clauses containing literal $\overline{x}$.

$\overline{n_3}$: the number of 2-clauses containing 3-variables and literal $\overline{x}$.

$\overline{n_4}$: the number of 2-clauses containing 4-variables and literal $\overline{x}$.

$\overline{n_5}$: the number of 2-clauses containing 5^+-variables and literal $\overline{x}$.

$m_1 = w_i + 2n_1\delta_i + (n_3 + \overline{n_3} + \overline{n_4})w_3 + n_4\delta_4 + n_5\delta_5 + \overline{n_5}w_4.$

$m_2 = w_i + 2\overline{n_1}\delta_i + (n_3 + \overline{n_3} + n_4)w_3 + \overline{n_4}\delta_4 + \overline{n_5}\delta_5 + n_5w_4.$

Recall that $\mathcal{F}$ is a reduced formula, $d(\mathcal{F}) = i$ and x is an i-variable in $\mathcal{F}$. We have the following lemma:

Lemma E.6 *The value m_1 is not larger than the l-value reduction from $\mathcal{F}$ to the formula reduced$_p(\mathcal{F}[x])$, and the value m_2 is not larger than the l-value reduction from $\mathcal{F}$ to the formula reduced$_p(\mathcal{F}[\overline{x}])$.*

PROOF. First we prove that m_1 is not larger than c. By the calculation of the c-value, each 2-clause containing a 3-variable and literal $\overline{x}$ adds w_3 to c, each 2-clause containing a 4-variable and literal $\overline{x}$ adds w_3 to c, and each 2-clause containing a i^+-variable and literal adds at least $w_i - \delta_i \geq w_4$ to c. where $i \geq 5$. So those 2-clauses containing literal $\overline{x}$ add at least $(\overline{n_3} + \overline{n_4})w_3 + \overline{n_5}w_4$. Also each 3^+-clause containing literal x adds at least $2\delta_i$ to c, each 2-clause containing a 3-variable and literal x adds w_3 to c, each 2-clause containing a 4-variable and literal x adds δ_4 to c, and each 2-clause containing a 5^+-variable and literal x adds $\delta_i \geq \delta_5$ to c since $i \geq 5$. So the clauses containing literal x add at least $2n_1\delta_i + n_3w_3 + n_4\delta_4 + n_5\delta_5$ to c. Thus c is at least $2n_1\delta_i + n_3w_3 + n_4\delta_4 + n_5\delta_5 + (\overline{n_3} + \overline{n_4})w_3 + \overline{n_5}w_4 = m_1$.

By lemma E.5, c is not larger than the l-value reduction from $\mathcal{F}$ to reduced$_p(\mathcal{F}[x])$.

So m_2 is not larger than the l-value reduction from $\mathcal{F}$ to reduced$_p(\mathcal{F}[x])$. By symmetry, we can prove that m_2 is not larger than the l-value reduction from $\mathcal{F}$ to reduced$_p(\mathcal{F}[\overline{x}])$. $\square$

Lemma E.6 is sufficient for most cases in the following analysis. Sometimes, we may need values better than m_1. Let

$n_{1,1}$: the number of 3-clauses containing literal x.

$n_{1,2}$: the number of 4^+-clauses containing literal x.

$\overline{n_{4,1}}$: the number of 2-clauses containing literal $\overline{x}$ and variable y such that some clause containing both literal x and variable y.

$\overline{n_{4,2}}$: the number of 2-clauses containing literal $\overline{x}$ and variable y such that no clauses containing both literal x and variable y.

$m_1' = w_i + (2n_{1,1} + 3n_{1,2})\delta_i + (n_3 + \overline{n_3} + \overline{n_{4,1}})w_3 + n_4\delta_4 + n_5\delta_5 + \overline{n_{4,2}}w_4 + \overline{n_5}w_4.$

By a proof similar to that for Lemma E.6, we can prove following lemma.

Lemma E.7 *The value m_1' is not larger than the l-value reduction from $\mathcal{F}$ to the formula reduced$_p(\mathcal{F}[x])$, and the value m_2 is not larger than the l-value reduction from $\mathcal{F}$ to the formula reduced$_p(\mathcal{F}[\overline{x}])$.*

Now we are ready to analyze the branching vector from $\mathcal{F}$ to reduced$_p(\mathcal{F}[x])$ and reduced$_p(\mathcal{F}(\overline{x})$.

Lemma E.8 *Let $\mathcal{F}$ be a reduced formula with $d(\mathcal{F}) = i$, and x be an i-variable in reduced($\mathcal{F}$). Then both (m_1, m_2) and (m_1', m_2) are inferior to the branching vector from $\mathcal{F}$ to reduced$_p(\mathcal{F}[x])$ and reduced$_p(\mathcal{F}[\overline{x}])$.*

Proof. By Lemma E.6, (m_1, m_2) is inferior to the branching vector from $\mathcal{F}$

to reduced$_p(\mathcal{F}[x])$ and reduced$_p(\mathcal{F}[\overline{x}])$. By Lemma E.7, (m_1', m_2) is inferior to the branching vector from $\mathcal{F}$ to reduced$_p(\mathcal{F}[x])$ and reduced$_p(\mathcal{F}[\overline{x}])$. □

If x is a $(i-1,1)$-literal and no 2-clause contains x, we can have a better branching vector.

Lemma E.9 *Given a reduced formula $\mathcal{F}$ of degree i, and an $(i-1,1)$-literal x in $\mathcal{F}$ such that no 2-clause contains x, let*

$$m_1' = w_i + 2n_1\delta_i + n_3w_3 + n_4\delta_4 + n_5\delta_5, \text{ and } m_2' = w_i + 3w_3.$$

Then (m_1', m_2') is inferior to the branching vector from $\mathcal{F}$ to reduced$_p(\mathcal{F}[x])$ and reduced$_p(\mathcal{F}[\overline{x}])$. Moreover, $\overline{n_1} = 1$ and $\overline{n_3} = \overline{n_4} = \overline{n_5} = 0$.

Proof. First we show that $\overline{n_1} = 1$ and $\overline{n_3} = \overline{n_4} = \overline{n_5} = 0$. Let $\overline{x}C$ be the clause containing literal $\overline{x}$. Then we must have $|\mathcal{C}| \geq 2$: if $|\mathcal{C}| = 0$, then Case 5 in the algorithm **Reduction** would be applicable to $\overline{x}$ in $\mathcal{F}$, and if $|\mathcal{C}| = 1$, then Case 10 in the algorithm would be applicable to $\overline{x}\mathcal{C}$ in $\mathcal{F}$. So we must have $|\mathcal{C}| \geq 2$. Since x is an $(i-1,1)$-literal, we have $\overline{n_1} = 1$, and $\overline{n_3} = \overline{n_4} = \overline{n_5} = 0$.

By the definition of inferior vectors, (m_1', m_2') is inferior to the branching vector from $\mathcal{F}$ to reduced$_p(\mathcal{F}[x])$ and reduced$_p(\mathcal{F}[\overline{x}])$, once we show that m_1' is not larger than the l-value reduction from $\mathcal{F}$ to reduced$_p(\mathcal{F}[x])$ and m_2' is not larger than the l-value reduction from $\mathcal{F}$ to reduced$_p(\mathcal{F}[\overline{x}])$.

We first show that m_1' is not larger than the l-value reduction from $\mathcal{F}$ to the formula reduced$_p(\mathcal{F}[x])$. Since $\overline{n_3} = \overline{n_4} = \overline{n_5} = 0$, $m_1 = w_i + 2n_1\delta_i + n_3w_3 + n_4\delta_4 + n_5\delta_5 = m_1'$. By lemma E.6, $m_1' = m_1$ is not larger than the l-value reduction from $\mathcal{F}$ to reduced$_p(\mathcal{F}[x])$.

Now we show that m_2' is at not larger than the l-value reduction from $\mathcal{F}$ to reduced$_p(\mathcal{F}[\overline{x}])$.

When $x = 0$, then all literals in $\mathcal{C}$ must be false in $\mathcal{F}[\overline{x}]$, since reduced$_p(\mathcal{F}[x])$ is not satisfiable. i.e., all variables in $x + C$ are not in reduced$_p(\mathcal{F}[\overline{x}])$. Since $\mathcal{F}$ is a reduced formula, all variables in $x + C$ are 3^+-variables. By lemma E.2, the l-value reduction from $\mathcal{F}$ to reduced$_p(\mathcal{F}[x])$ is at least $w_i + |\mathcal{C}|w_3$. If $|\mathcal{C}| \geq 3$, then the l-value reduction is at least $w_i + 3w_3$. Otherwise $|\mathcal{C}| = 2$. Let $\mathcal{C} = z_1z_2$. If both z_1 and z_2 are 4^+-variables, then the l-value reduction is at least $2w_4 \geq 3w_3$. Now we only need consider that at least one of z_1 and z_2, say z_1, is a 3-variable. Then literals x and $\overline{z_2}$ can not co-occur with literal $\overline{z_1}$ in any clause. Otherwise, Case 4 (on z_1) in the algorithm **reduction** would be applicable to $\mathcal{F}$. According to lemma C.5, only literal z_2 can co-occur with literal $\overline{z_1}$ in a 3^+-clause. Now we consider two cases.

Case 1: If $\overline{z_1}z_2$ is in a clause $\overline{z_1}z_2yC$, then z_2 must be a 4^+-variable. Otherwise, the clause $\overline{z_1}z_2yC$ can not exist by lemma C.5. Since y can not be the $(i-1,1)$-literal x, the l-value reduction is at least $w_i + w_3 + w_4 + \delta_i > w_i + 3w_3$ when $x = 0$ ($z_1 = 0$, $z_2 = 0$ and $y = 1$).

Case 2: If literal z_2 does not co-occur with literal $\overline{z_1}$ in any clause, then there is a clause $\overline{z_1}y_1\mathcal{C}_1$ such that y_1 is not in $\{x, \overline{x}, z_2, \overline{z_2}\}$. Also there is a clause $\overline{z_2}y_2\mathcal{C}_2$ such that y_2 is not in $\{x, \overline{x}, z_2, \overline{z_2}\}$. Otherwise, there is only a clause $\overline{z_2}z_1$. The Case 6 is applicable, thus contradicting that $\mathcal{F}$ is a reduced formula. Therefore, the l-value reduction is at least $w_i + w_3 + w_3 + 2\delta_j > w_i + 2w_3 + 1 > w_i + 3w_3$. □

1. Analysis for Degree-4 Formulas

Suppose that $d(\mathcal{F}) = 4$ and x is a 4-variable, let d_1 be the degree of literal x and d_0 be the degree of $\overline{x}$. We then consider all combinations of d_1 and d_0 such that $d_1 + d_0 = 4$. Without loss of generality, we assume that $d_1 \geq d_0$. Since $d(\mathcal{F}) = 4$, we have $n_5 = 0$ and $\overline{n_5} = 0$.

a. $d_0 = 1$

Note that $n_1 = 3 - n_3 - n_4$. By lemma E.9, $\overline{n_3} = \overline{n_4} = 0$.

Case 1: $n_3 + n_4 = 0$. Then $n_1 = 3$. By lemma E.9, the branching vector is not inferior to

$$(w_3 + 7\delta_4, 4w_3 + \delta_4).$$

Case 2: $1 \leq n_3 + n_4 \leq 3$. By lemma E.8, the branching vector is not inferior to (m_1, m_2), which is:

$(w_3 + 6\delta_4, 4w_3 + 2\delta_4)$ when $n_3 = 0$ and $n_4 = 1$,

$(w_3 + 5\delta_4, 5w_3 + 3\delta_4)$ when $n_3 = 0$ and $n_4 = 2$,

$(w_3 + 4\delta_4, 6w_3 + 4\delta_4)$ when $n_3 = 0$ and $n_4 = 3$,

$(2w_3 + 5\delta_4, 4w_3 + \delta_4)$ when $n_3 = 1$ and $n_4 = 0$,

$(2w_3 + 4\delta_4, 5w_3 + 2\delta_4)$ when $n_3 = 1$ and $n_4 = 1$,

$(2w_3 + 3\delta_4, 6w_3 + 3\delta_4)$ when $n_3 = 1$ and $n_4 = 2$,

$(3w_3 + 3\delta_4, 5w_3 + \delta_4)$ when $n_3 = 2$ and $n_4 = 0$,

$(3w_3 + 2\delta_4, 6w_3 + 2\delta_4)$ when $n_3 = 2$ and $n_4 = 1$,

$(4w_3 + \delta_4, 6w_3 + \delta_4)$ when $n_3 = 3$ and $n_4 = 0$.

b. $d_0 = 2$

Case 1: Two 2-clauses $\overline{x}y_1$ and $\overline{x}y_2$ contains literal $\overline{x}$.

Case 1.1: Two 3^+-clauses contain literal x, i.e., $n_3 = 2$ and $n_4 = 0$.

If one clause is a 4^+-clauses, i.e., $n_{1,1} = n_{1,2} = 1$, then by lemma E.8, the branching vector is not inferior to (m_1', m_2), which is

$$(3w_3 + 6\delta_4, w_3 + 3\delta_4).$$

If both are 3-clause, i.e., $n_1 = 2$, then we consider following cases:

Subcase 1.1.1: both y_1 and y_2 are 4-variable, i.e., $\overline{n_4} = 2$ and $\overline{n_3} = 0$. Then at most one of $\overline{y_1}$ and $\overline{y_2}$ co-occur with literal x. Otherwise, Case 9 in the algorithm **Reduction** would be applicable. If one of y_1 and y_2 co-occurs with literal x, i.e., $\overline{n_{4,1}} = \overline{n_{4,2}} = 1$, then by lemma E.8, the branching vector is not inferior to

$$(3w_3 + 6\delta_4, w_3 + 3\delta_4).$$

If none of y_1 and y_2 co-occurs with literal x, i.e., $\overline{n_4} = 2$ and $\overline{n_3} = 0$, then by lemma E.8, the branching vector is not inferior to

$$(w_3 + 7\delta_4, w_3 + 3\delta_4).$$

Subcase 1.1.2 y_1 is a 4-variable and y_2 is a 3-variable, i.e., $\overline{n_3} = \overline{n_4} = 1$.

If y_1 does not co-occur with literal x, then by lemma E.8, the branching vector is not inferior to (m_1', m_2), which is

$$(3w_3 + 6\delta_4, 2w_3 + 2\delta_4).$$

Otherwise, literal y_1 and x are in a clause xy_1z_1 since both clauses containing literal x are 3-clause in Subcase 1.1.1-1.1.3. Let xz_2z_3 be the other 3-clause containing literal x. Note that y_2 must be a 2-literal. Otherwise, Case 10 in the algorithm **Reduction** would be applied. Let the other two clauses containing variable y be y_2C_1 and $\overline{y_2}C_0$. By lemma E.9, $|C_0| \geq 2$. Note that variable x and y_2 are not in C_0. With these given formulas, step 4.1 in algorithm SATSolver($\mathcal{F}$) would be executed.

First we consider the case that C_1 is a 2^+-clause.

If C_0 is true, then we can set $y_2 = 1$, then $x = y_1$. Clauses $x\overline{y_1}z_1$, $\overline{x}y_1$, xy_2, and y_2C_1 are eliminated and are not in $\mathcal{F}[C_0 = \mathit{true}]$, clause xz_1z_2 in $\mathcal{F}$ becomes $y_1z_1z_2$ in $\mathcal{F}[C_0 = \mathit{true}]$, and clause $\overline{x}C_0$ in $\mathcal{F}$ becomes C_0 in $\mathcal{F}[C_0 = \mathit{true}]$. Note that y_1 is a 3-variable in $\mathcal{F}[C_0 = \mathit{true}]$. And it is obvious that only the degrees of x, y_2, y_1 and variables in C_1 in $\mathcal{F}[C_0 = \mathit{true}]$ are different to those in $\mathcal{F}$. Thus the l-value reduction from $\mathcal{F}$ to $\mathcal{F}[C_0 = \mathit{true}]$ is the contribution of $\{x, y_2, y_1\} \cup C_1$ from $\mathcal{F}$ to $\mathcal{F}[C_0 = \mathit{true}]$. The l-value reduction from $\mathcal{F}$ to reduced$_p(\mathcal{F}[C_0 = \mathit{true}])$ is the summation of the l-value reduction from $\mathcal{F}$ to $\mathcal{F}[C_0 = \mathit{true}]$ and the l-value reduction from $\mathcal{F}[C_0 = true]$ to reduced$_p(\mathcal{F}[C_0 = true])$. By lemma E.2, the l-value reduction from $\mathcal{F}[C_0 = true]$ to reduced$_p(\mathcal{F}[C_0 = true])$ is at least the contribution of $\{x, y_2, y_1\} \cup C_1$ from $\mathcal{F}[C_0 = true]$ to reduced$_p(\mathcal{F}[C_0 = true])$. So the l-value reduction from $\mathcal{F}$ to reduced$_p(\mathcal{F}[C_0 = true])$ is at least the contribution of $\{x, y_2, y_1\} \cup C_1$ from $\mathcal{F}$ to reduced$_p(\mathcal{F}[C_0 = true])$. If variable y_1 is not in C_1, the contribution of $\{x, y_2, y_1\} \cup C_1$ from $\mathcal{F}$ to reduced$_p(\mathcal{F}[C_0 = true])$ is at least $2w_3 + 4\delta_4$, no matter whether there are 3-variables or not in C_1. If variable y_1 is in C_1, the contribution of $\{x, y_2, y_1\} \cup C_1$ is still at least $2w_3 + 4\delta_4$ since y_1 is a 4-variable. For both cases, the l-value reduction from $\mathcal{F}$ to reduced$_p(\mathcal{F}[C_0 = true])$ is at least w_2 (or $2w_3$).

If C_0 is false, we require that all literals in C_0 are false, since we work on $\mathcal{F}[C_0 = false]$ only after $\mathcal{F}[C_0 = true]$ is not satisfiable. To satisfy $\mathcal{F}$, y_2 must be 0, which results in $x = 0$ from clause $\overline{x}y_2$. Hence clauses $\overline{x}y_1$, $\overline{x}y_2$ and $\overline{y_2}\mathcal{C}_0$ are not in $\mathcal{F}[C_0 = false]$. Similarly as above, we can show that the l-value reduction from $\mathcal{F}$ to reduced$_p(\mathcal{F}[C_0 = false])$ is at least the contribution of $\{x, y_2, y_1\} \cup C_0$ from $\mathcal{F}$ to reduced$_p(\mathcal{F}[C_0 = false])$. If variable y_1 is not in C_0, the contribution of $\{x, y_2, y_1\} \cup C_0$ from $\mathcal{F}$ to reduced$_p(\mathcal{F}[C_0 = false])$ is at least $4w_3 + 2\delta_4$, since (1) the contribution of x is w_4, (2) the contribution of y_2 is w_3, (3) the contribution of C_0 is at least $2w_3$ (recall that $|C_0| \geq 2$ and all variables in $\mathcal{F}$ are 3^+-variables), and (4) the contribution of y_1 is δ_4. If variable y_1 is in C_0, the contribution of $\{x, y_2, y_1\} \cup C_0$ from $\mathcal{F}$ to reduced$_p(\mathcal{F}[C_0 = false])$ is still at least $4w_3 + 2\delta_4$, since (1) the contribution of x and y_2 is $w_3 + w_4$, and (2) the contribution of C_0 is at least $w_4 + w_3$ (y_1 is a 4-variable). For both cases, the l-value reduction is at least $4w_3 + 2\delta_4$.

We conclude that when C_1 is a 2^+-clause, the l-value reduction from $\mathcal{F}$ to reduced$_p(\mathcal{F}[C_0 = true])$ is not inferior to

$$(2w_4 + 4\delta_4, 4w_3 + 2\delta_4)$$

Next we consider the case that $\mathcal{C}_1$ is a 1-clause u.

If $\mathcal{C}_0$ is true, then we can prove that the l-value reduction from $\mathcal{F}$ to the formula reduced$_p(\mathcal{F}[C_0 = true])$ is at least $2w_3 + 3\delta_4$, using similar proof for the first part of the case when C_1 is a 2^+-clause.

If $\mathcal{C}_0$ is false, we require that C_0 be false. To satisfy $\mathcal{F}$, clause $\overline{y_2}u$ requires $y_2 = 0$, which results in $x = 0$ because of clause $\overline{x}y_2$ and $u = 1$ because of clause $y_2\mathcal{C}_1$. Then clauses $\overline{x}y_1$, $\overline{y_2}C_0$ and $y_2\mathcal{C}_1$ are removed from $\mathcal{F}$. Note that only variables in $\{x, y_2, y_1, u\} \cup C_0$ change their degrees from $\mathcal{F}$ to $\mathcal{F}[C_0 = false]$. Similarly as above, we can show that the l-value reduction from $\mathcal{F}$ to reduced$_p(\mathcal{F}[C_0 = false])$ is at least the contribution of $\{x, y_2, y_1, u\} \cup C_0$ from $\mathcal{F}$ to reduced$_p(\mathcal{F}[C_0 = false])$. The contribution of x and y_2 is $w_4 + w_3 = 2w_3 + \delta_4$. Note that u is not in C_0 by lemma C.4, since y_2 is a 3-variable. Also u can not be x or y_2. If variable y_1 is not in $\{u\} \cup C_0$, then the contribution of $\{y_1, u\} \cup C_0$ is at least $3w_3 + \delta_4$ since $|C_0| \geq 2$ and y_1 is a 4-variable. If variable y_1 is in $\{u\} \cup C_0$, the contribution of $\{y_1, u\} \cup C_0$ is at least $w_4 + 2w_3 = 3w_3 + \delta_4$ since y_1 is a 4-variable. For both cases, the l-value reduction from $\mathcal{F}$ to reduced$_p(\mathcal{F}[C_0 = false])$ is at least $2w_3 + \delta_4 + 3w_3 + \delta_4 = 5w_3 + 4\delta_4$.

We conclude that when C_1 is a 1-clause, the l-value reduction from $\mathcal{F}$ to the formula reduced$_p(\mathcal{F}[C_0 = false])$ is not inferior to

$$(2w_3 + 4\delta_4, 5w_3 + 4\delta_4).$$

Subcase 1.1.3: both y_1 and y_2 are 3-variables, i.e., $\overline{n_3} = 2$ and $\overline{n_4} = 0$. By lemma E.8, the branching vector is not inferior to

$$(3w_3 + 5\delta_4, 3w_3 + \delta_4).$$

Case 1.2: One 3^+-clause contains literal x, i.e., $n_1 = 1$ and $n_3 + n_4 = 1$. Let xz be the 2-clause.

If z is a 3-variable, $n_3 = 1$ and $n_4 = 0$. By lemma E.8, the branching vector is not inferior to $(m_1, m_2) = (w_4 + 2n_1\delta_4 + (n_3 + \overline{n_3} + \overline{n_4})w_3 + n_4\delta_4, w_4 + 2\overline{n_1}\delta_4 + (n_3 + \overline{n_3} + n_4)w_3 + \overline{n_4}\delta_4) = (w_4 + 2\delta_4 + 3w_3, w_4 + (1 + \overline{n_3})w_3 + \overline{n_4}\delta_4)$, which is not inferior to

$(4w_3 + 3\delta_4, 2w_3 + 3\delta_4)$ when $\overline{n_3} = 0$ and $\overline{n_4} = 2$.

If z is a 4-variable, $n_3 = 0$ and $n_4 = 1$. By lemma C.4, variable z can not be neither variable y_1 nor y_2. Then $n_{4,1} = 0$, $n_{4,2} = 1$ and $n_3 + n_{4,1} = 1$. By lemma E.8, the branching vector is not inferior to $(m_1, m_2') =$ $(w_4 + 2n_1\delta_4 + (n_3 + \overline{n_3} + \overline{n_4})w_3 + n_4\delta_4, w_4 + (2\overline{n_{1,1}} + 3\overline{n_{1,2}})\delta_4 + (n_3 + \overline{n_3} + n_{4,1})w_3 + \overline{n_4}\delta_4 + n_{4,2}w_4) = (w_4 + 3\delta_4 + 2w_3, 2w_4 + \overline{n_3}w_3 + \overline{n_4}w_4)$, which is not inferior to

$(3w_3 + 4\delta_4, 2w_3 + 4\delta_4)$ when $\overline{n_3} = 0$ and $\overline{n_4} = 2$.

Case 1.3: Two 2-clauses contain literal x, i.e., $n_3 = 2$ and $n_4 = 0$.

By lemma E.8, the branching vector is not inferior to $(w_4 + (n_3 + \overline{n_3} + \overline{n_4})w_3 + n_4\delta_4, w_4 + (n_3 + \overline{n_3} + n_4)w_3 + \overline{n_4}\delta_4) = (w_4 + (n_3 + 2)w_3 + n_4\delta_4, w_4 + (2 + \overline{n_3})w_3 + \overline{n_4}\delta_4)$, which is not inferior to

$(3w_3 + 3\delta_4, 3w_3 + 3\delta_4)$ when $n_3 = \overline{n_3} = 0$.

Case 2: One 2-clauses $\overline{x}y_1$ and a 3^+-clause $\overline{xy_2y_3}\mathcal{C}_1$ with $|\mathcal{C}_1| \geq 0$ contain literal $\overline{x}$, i.e., $\overline{n_1} = 1$ and $\overline{n_3} + \overline{n_4} = 1$.

Case 2.1: Two 3^+-clauses contain literal x, i.e., $n_3 + n_4 = 2$.

By lemma E.8, the branching vector is not inferior to

$(2w_3 + 5\delta_4, w_3 + 4\delta_4)$ when $\overline{n_3} = 0$,

$(2w_3 + 5\delta_4, 2w_3 + 3\delta_4)$ when $\overline{n_3} = 1$.

Case 2.2: One 3^+-clause contains literal x.

We have $n_3 + n_4 = 1$ and $\overline{n_3} + \overline{n_4} = 1$. By lemma E.8, the branching vector is not inferior to $t = (w_4 + 2\delta_4 + (n_3 + \overline{n_3})w_3 + n_4\delta_4 + \overline{n_4}(w_4 - \delta_4), w_4 + 2\delta_4 + (n_3 + \overline{n_3})w_3 + n_4(w_4 - \delta_4) + \overline{n_4}\delta_4) = (w_4 + 2\delta_4 + w_3 + n_3w_3 + n_4\delta_4, w_4 + 2\delta_4 + w_3 + \overline{n_3}w_3 + \overline{n_4}\delta_4)$. It is clear that t is not inferior to

$(2w_3 + 4\delta_4, 2w_3 + 4\delta_4)$ (by setting $n_3 = \overline{n_3} = 1$).

Case 2.3: Two 2-clauses contain literal x.

This case is symmetric to Case 1.2.

Case 3: Two 3^+-clauses $\overline{x}\mathcal{C}_1$ and $\overline{x}\mathcal{C}_2$ contain literal $\overline{x}$, i.e., $\overline{n_3} = \overline{n_4} = 0$ and $\overline{n_1} = 2$.

Case 3.1: Two 3^+-clauses contain literal x.

We have $n_1 = 2$, $n_3 = n_4 = 0$. By lemma E.8, the branching vector is not inferior to

$$(w_3 + 5\delta_4, w_3 + 5\delta_4).$$

Case 3.2: One 3^+-clause contains literal x.

This case is symmetric to Case 2.1.

Case 3.3: Two 2-clauses contain literal x.

This case is symmetric to Case 1.1.

2. Analysis for Degree-5 Formulas

Suppose variable x is of degree 5, let d_1 be the degree of literal x and d_0 be the degree of $\overline{x}$. We then consider all combinations of d_1 and d_0 such that $d_1 + d_0 = 5$. W.l.o.g, it can be assumed that $d_1 \geq d_0$.

a. $d_0 = 1$

3^+-clause $\overline{x}y_1y_2\mathcal{C}$ contains literal $\overline{x}$ where $|\mathcal{C}| \geq 0$. We have $4 \geq n_3 + n_4 + n_5 \geq 0$, $n_1 = 5 - n_3 - n_4 - n_5$.

Case 1 $n_3 + n_4 + n_5 = 0$. x is a $(4, 1)$-literal. By lemma E.9, the branching vector is not inferior to

$$(w_5 + 8\delta_5, w_5 + 3w_3).$$

Case 2 $4 \geq n_3 + n_4 + n_5 = i \geq 1$. By lemma E.8, the branching vector is not inferior to $t = (w_5 + 2\delta_5(4 - n_3 - n_4 - n_5) + n_3w_3 + n_4\delta_4 + n_5\delta_5, w_5 + 2w_3 + n_3w_3 + n_4(w_4 - \delta_4) + n_5(w_5 - \delta_5)$. Note that t is not inferior to $t' = (w_5 + \delta_5(8 - 2(n_3 + n_4) - n_5) + (n_3 + n_4)\delta_4, w_5 + 2w_3 + (n_3 + n_4 + n_5)w_3 + n_5\delta_4)$.

When $n_5 = 0$, t' is

$(w_5 + 6\delta_5 + \delta_4, w_5 + 3w_3)$ when $n_3 + n_4 + n_5 = 1$,

$(w_5 + 4\delta_5 + 2\delta_4, w_5 + 4w_3)$ when $n_3 + n_4 + n_5 = 2$,

$(w_5 + 2\delta_5 + 3\delta_4, w_5 + 5w_3)$ when $n_3 + n_4 + n_5 = 3$,

$(w_5 + 4\delta_4, w_5 + 6w_3)$ when $n_3 + n_4 + n_5 = 4$.

When $n_5 = 1$, t' is

$(w_5 + 7\delta_5, w_5 + 3w_3 + \delta_4)$ when $n_3 + n_4 + n_5 = 1$,

$(w_5 + 5\delta_5 + \delta_4, w_5 + 4w_3 + \delta_4)$ when $n_3 + n_4 + n_5 = 2$,

$(w_5 + 3\delta_5 + 2\delta_4, w_5 + 5w_3 + \delta_4)$ when $n_3 + n_4 + n_5 = 3$,

$(w_5 + 3\delta_4, w_5 + 6w_3) + \delta_4$ when $n_3 + n_4 + n_5 = 4$.

When $n_5 = 2$, t' is

$(w_5 + 6\delta_5, w_5 + 3w_3 + 2\delta_4)$ when $n_3 + n_4 + n_5 = 2$,

$(w_5 + 4\delta_5 + \delta_4, w_5 + 4w_3 + 2\delta_4)$ when $n_3 + n_4 + n_5 = 3$,

$(w_5 + 2\delta_5 + 2\delta_4, w_5 + 5w_3 + 2\delta_4)$ when $n_3 + n_4 + n_5 = 4$.

When $n_5 = 3$, t' is

$(w_5 + 5\delta_5, w_5 + 3w_3 + 3\delta_4)$ when $n_3 + n_4 + n_5 = 3$,

$(w_5 + 3\delta_5 + \delta_4, w_5 + 4w_3 + 3\delta_4)$ when $n_3 + n_4 + n_5 = 4$.

When $n_5 = 4$, $n_3 = n_4 = 0$. t' is

$(w_5 + 4\delta_5, w_5 + 6w_3 + 4\delta_4)$.

b. $d_0 = 2$

Case 1: Two 2-clauses $\overline{x}y_1$ and $\overline{x}y_2$ contain literal $\overline{x}$, i.e., $\overline{n_1} = 0$ and $\overline{n_3} + \overline{n_4} + \overline{n_5} = 2$.

Case 1.1: Three 3^+-clauses contains literal x, i.e., $n_1 = 3$, $n_3 = n_4 = n_5 = 0$.

By lemma E.8, the branching vector is not inferior to

$t = (w_5 + 6\delta_5 + \overline{n_3}w_3 + \overline{n_4}(w_4 - \delta_4) + \overline{n_5}(w_5 - \delta_5), w_5 + \overline{n_3}w_3 + \overline{n_4}\delta_4 + \overline{n_5}\delta_5)$. Since $\overline{n_3} + \overline{n_4} + \overline{n_5} = 2$, $t = (w_5 + 6\delta_5 + 2w_3 + \overline{n_5}\delta_4, w_5 + \overline{n_3}w_3 + \overline{n_4}\delta_4 + \overline{n_5}\delta_5)$ and is not inferior to

$t' = (w_5 + 6\delta_5 + 2w_3 + \overline{n_5}\delta_4, w_5 + (\overline{n_3} + \overline{n_4})\delta_4 + \overline{n_5}\delta_5)$, which is:

$(w_5 + 6\delta_5 + 2w_3, w_5 + 2\delta_4)$ when $\overline{n_5} = 0$,

$(w_5 + 6\delta_5 + 2w_3 + \delta_4, w_5 + \delta_4 + \delta_5)$ when $\overline{n_5} = 1$,

$(w_5 + 6\delta_5 + 2w_3 + 2\delta_4, w_5 + 2\delta_5)$ when $\overline{n_5} = 2$.

Case 1.2: two 3^+-clauses and one 2-clause xz_1 contain literal x, i.e., $n_1 = 2$ and $n_3 + n_4 + n_5 = 1$.

By lemma E.8, the branching vector is not inferior to

$t = (w_5 + 4\delta_5 + (n_3 + \overline{n_3})w_3 + n_4\delta_4 + \overline{n_4}w_3 + n_5\delta_5 + \overline{n_5}w_4, w_5 + (n_3 + \overline{n_3})w_3 + n_4w_3 + \overline{n_4}\delta_4 + n_5w_4 + \overline{n_5}\delta_5)$. Since $\overline{n_3} + \overline{n_4} + \overline{n_5} = 2$ and $n_3 + n_4 + n_5 = 1$, we have

$t = (w_5 + 4\delta_5 + 2w_3 + n_3w_3 + n_4\delta_4 + n_5\delta_5 + \overline{n_5}\delta_4, w_5 + w_3 + n_5\delta_4 + \overline{n_3}w_3 + \overline{n_4}\delta_4 + \overline{n_5}\delta_5)$. Note that t is not inferior to

$t' = (w_5 + 4\delta_5 + 2w_3 + (1 - n_5)\delta_4 + n_5\delta_5 + \overline{n_5}\delta_4, w_5 + w_3 + n_5\delta_4 + (2 - \overline{n_5})\delta_4 + \overline{n_5}\delta_5)$, which is

$(w_5 + 4\delta_5 + 2w_3 + \delta_4, w_5 + w_3 + 2\delta_4)$ when $n_5 = 0$ and $\overline{n_5} = 0$,

$(w_5 + 4\delta_5 + 2w_3 + 2\delta_4, w_5 + w_3 + \delta_4 + \delta_5)$ when $n_5 = 0$ and $\overline{n_5} = 1$,

$(w_5 + 4\delta_5 + 2w_3 + 3\delta_4, w_5 + w_3 + 2\delta_5)$ when $n_5 = 0$ and $\overline{n_5} = 2$,

$(w_5 + 5\delta_5 + 2w_3, w_5 + w_3 + 3\delta_4)$ when $n_5 = 1$ and $\overline{n_5} = 0$,

$(w_5 + 5\delta_5 + 2w_3 + \delta_4, w_5 + w_3 + 2\delta_4 + \delta_5)$ when $n_5 = 2$ and $\overline{n_5} = 1$,

$(w_5 + 5\delta_5 + 2w_3 + 2\delta_4, w_5 + w_3 + \delta_4 + 3\delta_5)$ when $n_5 = 3$ and $\overline{n_5} = 2$.

Case 1.3: one 3^+-clauses and two 2-clauses xz_1, xz_2 contain literal x, i.e., $n_1 = 1$ and $n_3 + n_4 + n_5 = 2$.

By lemma E.8, the branching vector is not inferior to

$t = (w_5 + 2\delta_5 + (n_3 + \overline{n_3})w_3 + n_4\delta_4 + \overline{n_4}w_3 + n_5\delta_5 + \overline{n_5}w_4, w_5 + (n_3 + \overline{n_3})w_3 + n_4w_3 + \overline{n_4}\delta_4 + n_5w_4 + \overline{n_5}\delta_5)$. Since $\overline{n_3} + \overline{n_4} + \overline{n_5} = 2$ and $n_3 + n_4 + n_5 = 2$, we have

$t = (w_5 + 2\delta_5 + 2w_3 + n_3w_3 + n_4\delta_4 + n_5\delta_5 + \overline{n_5}\delta_4, w_5 + 2w_3 + n_5\delta_4 + \overline{n_3}w_3 + \overline{n_4}\delta_4 + \overline{n_5}\delta_5)$. Note that t is not inferior to

$t' = (w_5 + 2\delta_5 + 2w_3 + (2 - n_5)\delta_4 + n_5\delta_5 + \overline{n_5}\delta_4, w_5 + 2w_3 + n_5\delta_4 + (2 - \overline{n_5})\delta_4 + \overline{n_5}\delta_5)$, which is

$(w_5 + 2\delta_5 + 2w_3 + 2\delta_4, w_5 + 2w_3 + 2\delta_4)$ when $n_5 = 0$ and $\overline{n_5} = 0$,

$(w_5 + 2\delta_5 + 2w_3 + 3\delta_4, w_5 + 2w_3 + \delta_4 + \delta_5)$ when $n_5 = 0$ and $\overline{n_5} = 1$,

$(w_5 + 2\delta_5 + 2w_3 + 4\delta_4, w_5 + 2w_3 + 2\delta_5)$ when $n_5 = 0$ and $\overline{n_5} = 2$,

$(w_5 + 3\delta_5 + 2w_3 + \delta_4, w_5 + 2w_3 + 3\delta_4)$ when $n_5 = 1$ and $\overline{n_5} = 0$,

$(w_5 + 3\delta_5 + 2w_3 + 2\delta_4, w_5 + 2w_3 + 2\delta_4 + \delta_5)$ when $n_5 = 1$ and $\overline{n_5} = 1$,

$(w_5 + 3\delta_5 + 2w_3 + 3\delta_4, w_5 + 2w_3 + \delta_4 + 2\delta_5)$ when $n_5 = 1$ and $\overline{n_5} = 2$,

$(w_5 + 4\delta_5 + 2w_3, w_5 + 2w_3 + 4\delta_4)$ when $n_5 = 2$ and $\overline{n_5} = 0$,

$(w_5 + 4\delta_5 + 2w_3 + \delta_4, w_5 + 2w_3 + 3\delta_4 + \delta_5)$ when $n_5 = 2$ and $\overline{n_5} = 1$,

$(w_5 + 4\delta_5 + 2w_3 + 2\delta_4, w_5 + 2w_3 + 2\delta_4 + 2\delta_5)$ when $n_5 = 2$ and $\overline{n_5} = 2$.

Case 1.4: Three 2-clauses xz_1, xz_2 and xz_3 contain literal x, i.e., $n_1 = 0$ and $n_3 + n_4 + n_5 = 3$.

By lemma E.8, the branching vector is not inferior to

$t = (w_5 + (n_3 + \overline{n_3})w_3 + n_4\delta_4 + \overline{n_4}w_3 + n_5\delta_5 + \overline{n_5}w_4, w_5 + (n_3 + \overline{n_3})w_3 + n_4w_3 + \overline{n_4}\delta_4 + n_5w_4 + \overline{n_5}\delta_5)$. Since $\overline{n_3} + \overline{n_4} + \overline{n_5} = 2$ and $n_3 + n_4 + n_5 = 3$, we have

$t = (w_5 + 2w_3 + n_3w_3 + n_4\delta_4 + n_5\delta_5 + \overline{n_5}\delta_4, w_5 + 3w_3 + n_5\delta_4 + \overline{n_3}w_3 + \overline{n_4}\delta_4 + \overline{n_5}\delta_5)$.

Note that t is not inferior to

$t = (w_5 + 2w_3 + (3 - n_5)\delta_4 + n_5\delta_5 + \overline{n_5}\delta_4, w_5 + 3w_3 + n_5\delta_4 + (2 - \overline{n_5})\delta_4 + \overline{n_5}\delta_5)$, which is

$(w_5 + 2w_3 + 3\delta_4, w_5 + 3w_3 + 2\delta_4)$ when $n_5 = 0$ and $\overline{n_5} = 0$,

$(w_5 + 2w_3 + 4\delta_4, w_5 + 3w_3 + \delta_4 + \delta_5)$ when $n_5 = 0$ and $\overline{n_5} = 1$,

$(w_5 + 2w_3 + 5\delta_4 + 2\delta_5, w_5 + 3w_3 + 2\delta_5)$ when $n_5 = 0$ and $\overline{n_5} = 2$,

$(w_5 + 2w_3 + 2\delta_4 + \delta_5, w_5 + 3w_3 + 3\delta_4)$ when $n_5 = 1$ and $\overline{n_5} = 0$,

$(w_5 + 2w_3 + 3\delta_4 + \delta_5, w_5 + 3w_3 + 2\delta_4 + \delta_5)$ when $n_5 = 1$ and $\overline{n_5} = 1$,

$(w_5 + 2w_3 + 3\delta_4 + \delta_5, w_5 + 3w_3 + \delta_4 + 2\delta_5)$ when $n_5 = 1$ and $\overline{n_5} = 2$,

$(w_5 + 2w_3 + \delta_4 + 2\delta_5, w_5 + 3w_3 + 4\delta_4)$ when $n_5 = 2$ and $\overline{n_5} = 0$,

$(w_5 + 2w_3 + 2\delta_4 + 2\delta_5, w_5 + 3w_3 + 3\delta_4 + \delta_5)$ when $n_5 = 2$ and $\overline{n_5} = 1$,

$(w_5 + 2w_3 + 3\delta_4 + 2\delta_5, w_5 + 3w_3 + 2\delta_4 + 2\delta_5)$ when $n_5 = 2$ and $\overline{n_5} = 2$,

$(w_5 + 2w_3 + 3\delta_5, w_5 + 3w_3 + 5\delta_4)$ when $n_5 = 3$ and $\overline{n_5} = 0$,

$(w_5 + 2w_3 + 3\delta_5 + \delta_4, w_5 + 3w_3 + 4\delta_4 + \delta_5)$ when $n_5 = 3$ and $\overline{n_5} = 1$,

$(w_5 + 2w_3 + 3\delta_5 + 2\delta_4, w_5 + 3w_3 + 3\delta_4 + 2\delta_5)$ when $n_5 = 3$ and $\overline{n_5} = 2$.

Case 2: One 2-clause $\overline{x}y_1$ and a 3^+-clause $\overline{x}y_2y_3\mathcal{C}_1$ with $|\mathcal{C}_1| \geq 0$ contain literal $\overline{x}$, i.e., $\overline{n_1} = 1$ and $\overline{n_3} + \overline{n_4} + \overline{n_5} = 1$.

Case 2.1: Three 3^+-clauses contains literal x, i.e., $n_1 = 3$, $n_3 = n_4 = n_5 = 0$.

By lemma E.8, the branching vector is not inferior to

$t = (w_5 + 6\delta_5 + \overline{n_3}w_3 + \overline{n_4}w_3 + \overline{n_5}w_4, w_5 + 2\delta_5 + \overline{n_3}w_3 + \overline{n_4}\delta_4 + \overline{n_5}\delta_5)$. Since $\overline{n_3} + \overline{n_4} + \overline{n_5} = 1$, $t = (w_5 + 6\delta_5 + w_3 + \overline{n_5}\delta_4, w_5 + 2\delta_5 + \overline{n_3}w_3 + \overline{n_4}\delta_4 + \overline{n_5}\delta_5)$, not inferior to $t' = (w_5 + 6\delta_5 + 2w_3 + \overline{n_5}\delta_4, w_5 + 2\delta_5 + (1 - \overline{n_5})\delta_4 + \overline{n_5}\delta_5)$. t' is:

$(w_5 + 6\delta_5 + w_3, w_5 + 2\delta_5 + \delta_4)$ when $\overline{n_5} = 0$

$(w_5 + 6\delta_5 + w_3 + \delta_4, w_5 + 3\delta_5)$ when $\overline{n_5} = 1$.

Case 2.2: two 3^+-clauses and one 2-clause xz_1 contain literal x, i.e., $n_1 = 2$ and $n_3 + n_4 + n_5 = 1$.

By lemma E.8, the branching vector is not inferior to

$t = (w_5 + 4\delta_5 + (n_3 + \overline{n_3})w_3 + n_4\delta_4 + \overline{n_4}w_3 + n_5\delta_5 + \overline{n_5}w_4, w_5 + 2\delta_5 + (n_3 + \overline{n_3})w_3 + n_4w_3 + \overline{n_4}\delta_4 + n_5w_4 + \overline{n_5}\delta_5)$. Since $\overline{n_3} + \overline{n_4} + \overline{n_5} = 1$ and $n_3 + n_4 + n_5 = 1$, we have $t = (w_5 + 4\delta_5 + w_3 + n_3w_3 + n_4\delta_4 + n_5\delta_5 + \overline{n_5}\delta_4, w_5 + 2\delta_5 + w_3 + n_5\delta_4 + \overline{n_3}w_3 + \overline{n_4}\delta_4 + \overline{n_5}\delta_5)$. Note that t is not inferior to

$t' = (w_5 + 4\delta_5 + w_3 + (1 - n_5)\delta_4 + n_5\delta_5 + \overline{n_5}\delta_4, w_5 + 2\delta_5 + w_3 + n_5\delta_4 + (1 - \overline{n_5})\delta_4 + \overline{n_5}\delta_5)$, which is

$(w_5 + 4\delta_5 + w_3 + \delta_4, w_5 + \delta_5 + w_3 + 2\delta_4)$ when $n_5 = 0$ and $\overline{n_5} = 0$,

$(w_5 + 4\delta_5 + w_3 + 2\delta_4, w_5 + 2\delta_5 + w_3 + \delta_4 + \delta_5)$ when $n_5 = 0$ and $\overline{n_5} = 1$,

$(w_5 + 5\delta_5 + w_3, w_5 + \delta_5 + w_3 + 3\delta_4)$ when $n_5 = 1$ and $\overline{n_5} = 0$,

$$(w_5 + 5\delta_5 + w_3 + \delta_4, w_5 + 2\delta_5 + w_3 + 2\delta_4) \text{ when } n_5 = 1 \text{ and } \overline{n_5} = 1.$$

Case 2.3: one 3^+-clauses and two 2-clauses xz_1, xz_2 contain literal x, i.e., $n_1 = 1$ and $n_3 + n_4 + n_5 = 2$.

By lemma E.8, the branching vector is not inferior to

$t = (w_5 + 2\delta_5 + (n_3 + \overline{n_3})w_3 + n_4\delta_4 + \overline{n_4}w_3 + n_5\delta_5 + \overline{n_5}w_4, w_5 + 2\delta_5 + (n_3 + \overline{n_3})w_3 + n_4w_3 + \overline{n_4}\delta_4 + n_5w_4 + \overline{n_5}\delta_5)$. Since $\overline{n_3} + \overline{n_4} + \overline{n_5} = 1$ and $n_3 + n_4 + n_5 = 2$, we have

$t = (w_5 + 2\delta_5 + w_3 + n_3w_3 + n_4\delta_4 + n_5\delta_5 + \overline{n_5}\delta_4, w_5 + 2\delta_5 + w_3 + n_5\delta_4 + \overline{n_3}w_3 + \overline{n_4}\delta_4 + \overline{n_5}\delta_5)$.

Note that t is not inferior to

$t' = (w_5 + 2\delta_5 + w_3 + (2 - n_5)\delta_4 + n_5\delta_5 + \overline{n_5}\delta_4, w_5 + 2\delta_5 + 2w_3 + n_5\delta_4 + (1 - \overline{n_5})\delta_4 + \overline{n_5}\delta_5)$,

which is

$$(w_5 + 2\delta_5 + w_3 + 2\delta_4, w_5 + 2\delta_5 + 2w_3 + \delta_4) \text{ when } n_5 = 0 \text{ and } \overline{n_5} = 0,$$

$$(w_5 + 2\delta_5 + w_3 + 3\delta_4, w_5 + 2\delta_5 + 2w_3 + \delta_5) \text{ when } n_5 = 0 \text{ and } \overline{n_5} = 1,$$

$$(w_5 + 3\delta_5 + w_3 + \delta_4, w_5 + 2\delta_5 + +2w_3 + 2\delta_4) \text{ when } n_5 = 1 \text{ and } \overline{n_5} = 0,$$

$$(w_5 + 3\delta_5 + w_3 + 2\delta_4, w_5 + 3\delta_5 + 2w_3 + \delta_4) \text{ when } n_5 = 1 \text{ and } \overline{n_5} = 1,$$

$$(w_5 + 4\delta_5 + w_3, w_5 + 2\delta_5 + 2w_3 + 3\delta_4) \text{ when } n_5 = 2 \text{ and } \overline{n_5} = 0,$$

$$(w_5 + 4\delta_5 + w_3 + \delta_4, w_5 + 3\delta_5 + 2w_3 + 2\delta_4) \text{ when } n_5 = 2 \text{ and } \overline{n_5} = 1.$$

Case 2.4: Three 2-clauses xz_1, xz_2 and xz_3 contain literal x, i.e., $n_1 = 0$ and $n_3 + n_4 + n_5 = 3$.

By lemma E.8, the branching vector is not inferior to

$t = (w_5 + (n_3 + \overline{n_3})w_3 + n_4\delta_4 + \overline{n_4}w_3 + n_5\delta_5 + \overline{n_5}w_4, w_5 + 2\delta_5 + (n_3 + \overline{n_3})w_3 + n_4w_3 + \overline{n_4}\delta_4 + n_5w_4 + \overline{n_5}\delta_5)$. Since $\overline{n_3} + \overline{n_4} + \overline{n_5} = 1$ and $n_3 + n_4 + n_5 = 3$, we have

$t = (w_5 + w_3 + n_3w_3 + n_4\delta_4 + n_5\delta_5 + \overline{n_5}\delta_4, w_5 + 2\delta_5 + 3w_3 + n_5\delta_4 + \overline{n_3}w_3 + \overline{n_4}\delta_4 + \overline{n_5}\delta_5)$.

Note that t is not inferior to

$t = (w_5 + w_3 + (3 - n_5)\delta_4 + n_5\delta_5 + \overline{n_5}\delta_4, w_5 + 2\delta_5 + 3w_3 + n_5\delta_4 + (1 - \overline{n_5})\delta_4 + \overline{n_5}\delta_5)$,

which is

$$(w_5 + w_3 + 3\delta_4, w_5 + 2\delta_5 + 3w_3 + \delta_4) \text{ when } n_5 = 0 \text{ and } \overline{n_5} = 0,$$

$$(w_5+w_3+4\delta_4, w_5+3\delta_5+3w_3) \text{ when } n_5=0 \text{ and } \overline{n_5}=1,$$

$$(w_5+w_3+2\delta_4+\delta_5, w_5+2\delta_5+3w_3+2\delta_4) \text{ when } n_5=1 \text{ and } \overline{n_5}=0,$$

$$(w_5+w_3+3\delta_4+\delta_5, w_5+3\delta_5+3w_3+\delta_4) \text{ when } n_5=1 \text{ and } \overline{n_5}=1,$$

$$(w_5+w_3+\delta_4+2\delta_5, w_5+2\delta_5+3w_3+3\delta_4) \text{ when } n_5=2 \text{ and } \overline{n_5}=0,$$

$$(w_5+w_3+2\delta_4+2\delta_5, w_5+3\delta_5+3w_3+2\delta_4+) \text{ when } n_5=2 \text{ and } \overline{n_5}=1,$$

$$(w_5+w_3+3\delta_5, w_5+2\delta_5+3w_3+4\delta_4) \text{ when } n_5=3 \text{ and } \overline{n_5}=0,$$

$$(w_5+w_3+3\delta_5+\delta_4, w_5+3\delta_5+3w_3+3\delta_4) \text{ when } n_5=3 \text{ and } \overline{n_5}=1.$$

Case 3: Two 3^+-clauses $\overline{x}\mathcal{C}_1$ and $\overline{x}\mathcal{C}_2$ contain literal $\overline{x}$, i.e., $\overline{n_1}=2$ and $\overline{n_3}=\overline{n_4}=\overline{n_5}=0$.

Case 3.1: Three 3^+-clauses contains literal x, i.e., $n_1=3$, $n_3=n_4=n_5=0$.

By lemma E.8, the branching vector is not inferior to

$$(w_5+6\delta_5, w_5+4\delta_5).$$

Case 3.2: two 3^+-clauses and one 2-clause xz_1 contain literal x, i.e., $n_1=2$ and $n_3+n_4+n_5=1$.

By lemma E.8, the branching vector is not inferior to

$t=(w_5+4\delta_5+n_3w_3+n_4\delta_4+n_5\delta_5, w_5+4\delta_5+n_3w_3+n_4w_3+n_5w_4)$. Since $n_3+n_4+n_5=1$, $t=(w_5+4\delta_5+n_3w_3+n_4\delta_4+n_5\delta_5, w_5+4\delta_5+w_3+n_5\delta_4)$. Note that t is not inferior to

$t'=(w_5+4\delta_5+(1-n_5)\delta_4+n_5\delta_5, w_5+4\delta_5+w_3+n_5\delta_4)$, which is

$$(w_5+5\delta_5+\delta_4, w_5+4\delta_5+w_3) \text{ when } n_5=0,$$

$$(w_5+5\delta_5, w_5+\delta_5+w_3+\delta_4) \text{ when } n_5=1.$$

Case 3.3: one 3^+-clauses and two 2-clauses xz_1, xz_2 contain literal x, i.e., $n_1=1$ and $n_3+n_4+n_5=2$.

By lemma E.8, the branching vector is not inferior to

$t=(w_5+2\delta_5+n_3w_3+n_4\delta_4+n_5\delta_5, w_5+2\delta_5+n_3w_3+n_4w_3+n_5w_4)$. Since $n_3+n_4+n_5=2$,

$t = (w_5 + 2\delta_5 + 2w_3 + n_3w_3 + n_4\delta_4 + n_5\delta_5, w_5 + 4\delta_5 + 2w_3 + n_5\delta_4)$. Note that t is not inferior to

$t' = (w_5 + 2\delta_5 + w_3 + (2 - n_5)\delta_4 + n_5\delta_5, w_5 + 4\delta_5 + 2w_3 + n_5\delta_4)$, which is

$$(w_5 + 2\delta_5 + w_3 + 2\delta_4, w_5 + 4\delta_5 + 2w_3) \text{ when } n_5 = 0,$$
$$(w_5 + 3\delta_5 + w_3 + \delta_4, w_5 + 4\delta_5 + 2w_3 + \delta_4) \text{ when } n_5 = 1,$$
$$(w_5 + 4\delta_5 + w_3, w_5 + 4\delta_5 + 2w_3 + 2\delta_4) \text{ when } n_5 = 2.$$

Case 3.4: Three 2-clauses xz_1, xz_2 and xz_3 contain literal x, i.e., $n_1 = 0$ and $n_3 + n_4 + n_5 = 3$.

By lemma E.8, the branching vector is not inferior to

$t = (w_5 + n_3w_3 + n_4\delta_4 + n_5\delta_5, w_5 + 4\delta_5 + n_3w_3 + n_4w_3 + n_5w_4)$. Since $n_3 + n_4 + n_5 = 3$, $t = (w_5 + 2w_3 + n_3w_3 + n_4\delta_4 + n_5\delta_5, w_5 + 4\delta_5 + 2w_3 + n_5\delta_4)$. Note that t is not inferior to

$t' = (w_5 + w_3 + (3 - n_5)\delta_4 + n_5\delta_5, w_5 + 4\delta_5 + 2w_3 + n_5\delta_4)$, which is

$$(w_5 + w_3 + 3\delta_4, w_5 + 4\delta_5 + 2w_3) \text{ when } n_5 = 0,$$
$$(w_5 + w_3 + 2\delta_4 + \delta_5, w_5 + 4\delta_5 + 2w_3 + \delta_4) \text{ when } n_5 = 1,$$
$$(w_5 + w_3 + \delta_4 + 2\delta_5, w_5 + 4\delta_5 + 2w_3 + 2\delta_4) \text{ when } n_5 = 2,$$
$$(w_5 + w_3 + 3\delta_5, w_5 + 4\delta_5 + 2w_3 + 3\delta_4) \text{ when } n_5 = 3.$$

3. Analysis for Formulas of Degree Larger Than 5

Let x be a (d_1, d_0)-literal. Then $d = d_1 + d_0 \geq 6$. Suppose there are s_1 2-clauses containing literal x, and s_0 2-clauses containing literal $\overline{x}$. Let l_1 be the l-value reduction and c_1 be the value of c from $\mathcal{F}$ to $\mathcal{F}[x]$). Let l_0 be the l-value reduction and c_0 be the c-value from $\mathcal{F}$ to $\mathcal{F}[\overline{x}]$. By lemma E.5, c_1 (c_0) is not larger than l_1 (l_0). Thus $l_1 + l_2 \geq c_1 + c_2$. It can be verified that each 2-clause (containing either x or $\overline{x}$) adds at least 0.5 to both c_1 and c_0 by the calculation of c-value. Thus the $s_1 + s_2$ 2-clauses add at least $s_1 + s_0$ to $c_1 + c_2$. Moreover, the $d_1 - s_1$ 3^+-clauses containing literal x add at least $2(d_1 - s_1)\delta_i \geq d_1 - s_1$ to c_1 since $i \geq 3$ (all variables in $\mathcal{F}$ are 3-variables), and the $d_0 - s_0$ 3^+-clauses containing literal $\overline{x}$ add at least $d_0 - s_0$ to c_0. Thus the 3^+-clauses add at least $(d_1 - s_1) + (d_0 - s_0)$ to $c_1 + c_2$. Finally, x adds $w_d = 0.5d \geq 3$ to both c_1 and c_0 since x is a d-variable where $d \geq 6$. Thus x add at least $2w_d \geq 6$ to $c_1 + c_0$. Therefore, we have that $l_1 + l_0 \geq c_1 + c_0 \geq (s_1 + s_0) + (d_1 - s_1) + (d_0 - s_0) + 6 \geq 12$.

Next we prove that both of l_1 and l_0 is greater than $0.5 + d/2 = 3.5$. As shown above, the $s_1 + s_0$ 2-clauses add at least $0.5(s_1 + s_0)$ to both c_1 and c_0, the $(d_1 - s_1)$ 3^+-clauses add at least $d_1 - s_1$ to c_1, the $(d_0 - s_0)$ 3^+-clauses add at least $d_0 - s_0$ to c_0, and x add at least 3 to both c_1 and c_0. Thus $c_1 \geq 0.5(s_1 + s_0) + (d_1 - s_1) + 3$ and $c_0 \geq 0.5(s_1 + s_0) + (d_0 - s_0) + 3$. Note that $d_0 - s_1 \geq 0$ and $d_0 - s_0 \geq 0$. If $s_1 + s_0 = 1$, then both c_1 and c_0 are not less than 3.5. If $s_1 + s_0 = 0$, then $s_1 = s_0 = 0$. Since both d_1 and d_0 are not less than 1, we have that both c_1 and c_0 are not less than 4. By lemma E.5, both l_1 and l_0 are not less than 3.5.

So the branching vector in this case is at least (l_1, l_2), not inferior to $(3.5, 12 - 3.5) = (3.5, 8.5)$, which leads to $O(1.1313^{l(\mathcal{F})})$.

4. Branching Vector for the Main Algorithm

Summarizing all the above discussion, we can verify that the worst case occurs in the Subcase 1.1.3 in subsection E.1. The branching vector for this worst case is $t_0 = (3w_3 + 5\delta_4, 3w_3 + \delta_4) = (7.491, 3.891)$, which is inferior to the branching vectors for all other cases. The root of the polynomial corresponding to this worst branching vector is $\tau(t_0) \leq 1.1346$. In conclusion, we derive that the time complexity of the algorithm **SATSolver** is bounded by $O(1.1346^{l(\mathcal{F})})$ on an input formula $\mathcal{F}$. Let L be the total length of the formula $\mathcal{F}$, and observe that $l(\mathcal{F}) \leq L/2$, we finally conclude the running time of the algorithm **SATSolver** bounded by $O(1.0652^{L})$, which completes the proof of Theorem D.1.

F. Final Remarks

Our main algorithm is very simple. Though our algorithm has detailed analysis of numerous cases, its analysis is quite straightforward and simple. The previous algorithm by Wahlstöm is also simple. But its analysis is quite complicated. It is interesting that both our algorithm and the algorithm by Wahlstöm deal with formulas with low degree and formulas with large degree differently.

```
**Algorithm Reduction**($\mathcal{F}$)
INPUT: a non-empty formula $\mathcal{F}$
OUTPUT: an equivalent formula on which no further reduction is
        applicable

change = true;
while change do
    Case 1.  a clause $C$ is a subset of a clause $D$: remove $D$;
    Case 2.  a clause $C$ contains both $x$ and $\overline{x}$: remove $C$;
    Case 3.  a clause $C$ contains multiple copies of a literal $z$:
                 remove all but one $z$ in $C$;
    Case 4.  there is a variable $x$ with at most one non-trivial resolvent:
                 $\mathcal{F} \leftarrow DP_x(\mathcal{F})$;
    Case 5.  there is a 1-clause $(z)$ or a monotone literal $z$: $\mathcal{F} \leftarrow \mathcal{F}[z]$;
    Case 6.  there exist a 2-clause $z_1z_2$ and a clause $z_1\overline{z_2}C$:
                 remove $\overline{z_2}$ from the clause $z_1\overline{z_2}C$;
    Case 7.  there are clauses $z_1z_2C_1$ and $z_1\overline{z_2}C_2$ and $z_2$ is a $(2,1)$-literal:
                 remove $z_1$ from the clause $z_1z_2C_1$;
    Case 8.  there are clauses $z_1z_2$ and $\overline{z_1z_2}C$ such that literal $\overline{z_1}$ is a
             1-literal: remove the clause $z_1z_2$;
    Case 9.  there is a $(2,2)$-variable $x$ with clauses $\overline{x}z_1$, $\overline{x}z_2$ and two
             3-clauses $xC_1$ and $xC_2$ such that both $\overline{z_1}$ and $\overline{z_2}$ are 4-variables
             in either $C_1$ or $C_2$: $\mathcal{F} \leftarrow DP_x(\mathcal{F})$. Apply Case 2, if possible.
    Case 10. there is a 2-clause $z_1z_2$ where $z_1$ is a 1-literal, or there are two
             2-clauses $z_1z_2$ and $\overline{z_1z_2}$:
                 replace $z_1$ with $\overline{z_2}$. Apply Case 2, if possible;
    Case 11. there are two clauses $CD_1$ and $CD_2$ with $|C| > 1$:
                 replace $CD_1$ and $CD_2$ with $\overline{x}C$, $xD_1$, and $xD_2$, where $x$ is
                 a new variable. Apply Case 2 on variables in $C$, if possible.
    default: change = false;
```

Fig. 12. The reduction algorithm

```
Algorithm SATSolver($\mathcal{F}$)
INPUT: a CNF formula $\mathcal{F}$
OUTPUT: a report whether $\mathcal{F}$ is satisfiable
1. Reduction($\mathcal{F}$);
2. pick a $d(\mathcal{F})$-variable $x$;
3. if $d(\mathcal{F}) > 5$ then
     return SATSolver($\mathcal{F}[x]$) $\vee$ SATSolver($\mathcal{F}[\overline{x}]$);
4. else if $d(\mathcal{F}) > 3$ then
4.1    if $x$ is a $(2,2)$-variable with clauses $x\overline{y}_1z_1$, $xz_2z_3$, $\overline{x}y_1$, and $\overline{x}y_2$ such
        that $y_1$ is a 4-variable and $y_2$ is a 3-variable then
          let $\overline{y}_2C_0$ be the clause containing $\overline{y}_2$;
          return SATSolver($\mathcal{F}[C_0 = \text{true}]$) $\vee$ SATSolver($\mathcal{F}[C_0 = \text{false}]$);
4.2    if both $x$ and $\overline{x}$ are $2^+$-literals then
          return SATSolver($\mathcal{F}[x]$) $\vee$ SATSolver($\mathcal{F}[\overline{x}]$);
4.3    else (* assume that $\overline{x}$ occurs in a single clause $(\overline{x} \vee z_1 \vee \cdots \vee z_h)$ *)
          return SATSolver($\mathcal{F}[x]$) $\vee$ SATSolver($\mathcal{F}[\overline{x}, \overline{z}_1, \ldots, \overline{z}_h]$);
5. else if $d(\mathcal{F}) = 3$ then
     Apply the algorithm by Wahlström [81];
6. else return true;
```

Fig. 13. Algorithm for the SATISFIABILITY problem

CHAPTER VIII

SUMMARY AND FUTURE RESEARCH

A. Dissertation Summary

In this dissertation, we study a new approach—measure driven algorithm design and analysis. In Chapters II to VI, we present improved fpt-algorithms for several NP-hard problems. In Chapter VII, we present an improved exact algorithm for the well-known SATISFIABILITY problem. For those problems, we pick multiple measures and then find properties which help to design better algorithms. The traditional choice of single measure for a problem often restricts the application of structural properties of that problem. The discussions in previous chapters reveal that proper choice of measures do allow more structural properties to be applied to design better algorithms.

In Chapter II, we consider two measures for the 3-D MATCHING and 3-SET PACKING problems: the number of colors needed in the coloring step and the number of elements in the dynamic programming step. With the choice of these measures, we find it is better to search for a matching (packing) M' of size $i+1$ when a matching (packing) M of size i is given, according to the following property—every tuple in M contains at least two symbols in M'. This property reduces both the number of colors needed in the coloring step and the number of elements in the dynamic programming step, thus resulting in an improved deterministic algorithm of running time $O^*(4.61^{3k})$ for both the 3-D MATCHING and 3-SET PACKING problems. Moreover, the 3-D MATCHING problem can be solved with elements in two columns of M' in the dynamic programming step. By taking this advantage of the 3-D MATCHING problem, we can further reduce the number of colors needed in the coloring step and the number of elements in the dynamic programming step. The further reduction results in an improved deterministic algorithm of running time $O^*(2.32^{3k})$ for the 3-D MATCHING problem.

In Chapter III, we study the MULTIWAY CUT problem. Besides considering the size of the multiway cut to search for as a measure, we also take as a measure the minimum cut from a terminal to the other terminals. These two measures are effective because of three properties: (1) there is a vertex which either decreases the size of the multiway cut to search for or increase the minimum cut from a terminal to the other terminals, (2) the MULTIWAY CUT problem can be answered negatively when the minimum cut from a terminal to the other terminals is larger than the size of the multiway cut to search for, and (3) the MULTIWAY CUT problem can be solved in polynomial time when the size of the multiway cut to search for is zero. These properties and measures lead to an faster algorithm for the MULTIWAY CUT problem.

In Chapter IV, the FEEDBACK VERTEX SET problem on undirected graphs is studied. We apply the iterative compression approach to this problem. It is more effective to consider as measures both the size of the feedback vertex set (fvs) to search for and the number of connected components (ncc) in another feedback vertex set which does not intersect with fvs. It turns out there is a vertex which either decrease fvs or ncc. Moreover, the FEEDBACK VERTEX SET problem can be solved when $ncc = 1$ or $|fvs| = 0$. With these properties and measures, we design an fpt-algorithm of running time $O^*(5^k)$ for the FEEDBACK VERTEX SET problem on undirected graphs. Further investigations result in an fpt-algorithms of running time $O^*(5^k)$ for the weighted FEEDBACK VERTEX SET problem on undirected graphs.

In Chapter V, we study the FEEDBACK VERTEX SET problem on directed graphs, which had been an important open problem for 16 years before our algorithm. Similarly as to the FEEDBACK VERTEX SET problem on undirected graphs, we still apply the iterative compression approach to the FEEDBACK VERTEX SET problem on directed graphs. However, the concept of connected components makes no sense in directed graphs. Thus the measures for the FEEDBACK VERTEX SET problem on undirected graphs can not be used for the FEEDBACK VERTEX SET problem on directed graphs. However, the FEEDBACK VERTEX SET problem on directed graphs can be transformed into $O(k!)$ SKEW SEPARATOR problems. For the SKEW SEPARATOR problem, we consider two measures: the size of the skew separator cut to search for and the minimum cut from the *last* source to all the sinks. Properties similar to those for the MULTIWAY CUT problem can be proved for the SKEW SEPARATOR problem. An fpt-algorithm of running time $O^*(4^k)$ follows from these properties and measures for the SKEW SEPARATOR. Then an fpt-algorithm of running time $O^*(k!4^k)$ for the FEEDBACK VERTEX SET problem on digraphs follows from the algorithm for the SKEW SEPARATOR problem and the transformation from the FEEDBACK VERTEX SET problem to the $O(k!)$ SKEW SEPARATOR problems.

In Chapter VI, the MAX LEAF problem on directed graphs is studied. We focus on a special case of this problem—the root r of an out-branching $\mathcal{T}$ with at least k vertices is already given. For this special case, we try to extend an out-tree $\mathcal{T}'$ rooted at r to $\mathcal{T}$. During the processing of extending $\mathcal{T}'$ to $\mathcal{T}$, there are two important measures: (1) a subset L_1 of leaves in $\mathcal{T}'$ such that every leaf in L_1 can reach exactly one leaf in $\mathcal{T}$, and (2) the subset L_2 of the remaining leaf in $\mathcal{T}'$. A leaf in $\mathcal{T}'$ either belong to L_1 or L_2. Moreover, a leaf of $\mathcal{T}'$ with only one out-neighbour in $G - \mathcal{T}'$ must be in L_1. Then a leaf either is in L_1 or extends $\mathcal{T}'$ to a new out-tree which has one more leaf than $\mathcal{T}'$. The MAX LEAF problem can be solved when $|L_1| = k$ or $|L_1| + |L_2| \geq k$. With these measures and properties, we design an improved fpt-algorithm of running time $O^*(4^k)$ for the MAX LEAF problem on directed graphs. This algorithm can also be easily applied to solve the MAX LEAF problem on undirected graphs.

In Chapter VII, we study the well know SATISFIABILITY problem by considering its time complexity related to the total length L of input formulas. Most previous algorithms study this problem by analyzing how the length L changes for various cases. The previous best algorithm considers a complicated function of L and other factors. The algorithm is fast. But its analysis is daunting because of the complicated function. Our algorithm takes a simple function of the number n_i of variables of degree i and the weights w_i associated with variables of degree i. Our function does not depend on L directly. Instead, it is bounded by $L/2$ by careful choices of w_i's. With this function, we design reduction rules which make our main algorithm for the SATISFIABILITY problem quite simple.

B. Future Work

There are several interesting questions related to the new approach and the problems studied in this dissertation: further study of this new approach, randomized and algebraic algorithms, and kernelization.

1. Further Study of the New Approach

Normally, it is difficult to design fpt-algorithms of upper bound better than $O^*(4^k)$ with our new approach. For example, we present $O^*(4^k)$ algorithms for the MULTIWAY CUT problem, the FEEDBACK VERTEX SET problem, the SKEW SEPARATOR problem, and the MAX LEAF problem in this dissertation. These problems are studied with the consideration of two measures m_1 and m_2. The changes of m_1 and m_2 for these problems are not identical. For example, either m_1 or m_2 increases by 1 for the MULTIWAY CUT problem, while either m_1 increases by 1 and m_2 decreases by 1 or

m_2 increases by 1 for the MAX LEAF problem. However, analyses show that the algorithms for both problems are of running time $O^*(4^k)$.

To have better algorithms, we can study our new approach in two directions: (1) find new structural properties which ensure larger measure changes, or (2) design better measures for known properties. For example, if some property requires that m_1 increase by at least 2, we can have much faster algorithms for the MULTIWAY CUT problem and the MAX LEAF problem. On the other hand, new measures may also improve analyses and result in better algorithms. Overall, we should consider both measures and properties together. This consideration allows more opportunities for better algorithms.

2. Randomized and Algebraic Algorithms

There are faster randomized algorithms than our algorithms for the FEEDBACK VERTEX SET problem on undirected graphs [6], the 3-D MATCHING and 3-SET PACKING problems [72]. However, there are no randomized algorithms faster than our algorithms for the MULTIWAY CUT problem, the FEEDBACK VERTEX SET problem on directed graphs, and the MAX LEAF problem. Simple randomization of our deterministic algorithms do not lead to faster randomized algorithms for the MULTIWAY CUT problem, the FEEDBACK VERTEX SET problem on directed graphs, and the MAX LEAF problem. Can we speed up our deterministic algorithms by randomizing our algorithms for those problems? A natural and interesting question is what properties of a problem result in randomized algorithms faster than deterministic algorithms.

The best algorithms for the 3-D MATCHING and 3-SET PACKING problems are randomized algebraic algorithms [72]. Can these randomized algebraic algorithms be derandomized? Can we have algebraic algorithms faster than our deterministic algorithms for the MULTIWAY CUT problem, or for the FEEDBACK VERTEX SET problems?

3. Kernelization

Fpt-algorithms are of running time $O(f(k)n^{O(1)})$, which can be very practical when both n and k are small. There are applications whose parameter k is small. But their input size n can be very large. It is of practical interests to reduce their input size n significantly. Formally, given any instance I of a problem, we in polynomial time reduced the the instance to another instance I' such that (1) I has a solution if and only if I' has a solution, (2) we can construct a solution to I from a solution to I' in polynomial time, and (3) If the size of I' is larger than some function $g(k)$, then we can find a solution for I'. Such reduction is called *kernelization* and we say the problem has a $g(k)$ *kernel*. Theoretically, a parameterized problem has a kernel if and only if it is fixed-parameter tractable [39].

Currently, the 3-D MATCHING and the 3-set packing problems have $O(k^3)$ kernels [44], and the FEEDBACK VERTEX SET problem on undirected graphs has an $O(k^2)$ kernel [98]. It is challenging to have better kernels for these problems. Moreover, can we find a polynomial kernel for the MULTIWAY CUT problem or for the FEEDBACK VERTEX SET problem on directed graphs?

While the MAX LEAF problem has no polynomial kernel, it has polynomial number of polynomial kernels [51]. That is, any instance of this problem can be reduced to $O(n^{O(1)})$ number of smaller instances whose size is bounded by $O(k^{O(1)})$. Can we find many polynomial kernels for the MULTIWAY CUT problem or for the FEEDBACK VERTEX SET problem on directed graphs?

REFERENCES

[1] N. Alon, F. Fomin, G. Gutin, M. Krivelevich, and S. Saurabh, "Better algorithms and bounds for directed maximum leaf problems," in *Proc. of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science*, New Delhi, India, Dec. 2007, pp. 316–327.

[2] ——, "Parameterized algorithms for directed maximum leaf problems," in *Proc. of the 34th International Colloquium on Automata, Languages and Programming*, Wroclaw, Poland, July 2007, pp. 352–362.

[3] N. Alon, R. Yuster, and U. Zwick, "Color-coding," *Journal of ACM*, vol. 42, no. 2, pp. 844–856, 1995.

[4] D. Applegate, R. Bixby, V. Chvatal, and W. Cook, "On the solution of traveling salesman problems," *Documenta Mathematica*, pp. 645–656, 1998.

[5] V. Bafna, P. Berman, and T. Fujito, "A 2-approximation algorithm for the undirected feedback vertex set problem," *SIAM Journal on Discrete Mathematics*, vol. 12, no. 3, pp. 289–297, 1999.

[6] A. Becker, R. Bar-Yehuda, and D. Geiger, "Randomized algorithms for the loop cutset problem," *Journal of Artificial Intelligence Research*, vol. 12, pp. 219–234, 2000.

[7] H. Bodlaender, "A cubic kernel for feedback vertex set," in *Proc. of the 24th Annual Symposium on Theoretical Aspects of Computer Science*, Aachen, Germany, Feb. 2007, pp. 320–331.

[8] H. Bodlaender and E. Penninkx, "A linear kernel for planar feedback vertex set," in *Proc. of the 3rd International Workshop on Parameterized and Exact Computation*, Victoria, Canada, May 2008, pp. 160–171.

[9] H. L. Bodlaender, "On linear time minor test and depth-first search," in *Lecture Notes in Computer Science.* New York, USA: Heidelberg Springer, 1989, vol. 382, pp. 577–590.

[10] ——, "On disjoint cycles," *International Journal of Foundation of Computer Science*, vol. 5, no. 1, pp. 59–68, 1994.

[11] P. Bonsma, T. Brueggemann, and G. Woeginger, "A faster FPT algorithm for finding spanning trees with many leaves," in *Proc. of the 28th International Symposium on Mathematical Foundations of Computer Science*, Český Krumlov, Czech Republic, Aug. 2003, pp. 259–268.

[12] P. Bonsma and F. Dorn, "An FPT algorithm for directed spanning k-leaf," Corr abs/0711.4052, 2007.

[13] ——, "Tight bounds and faster algorithms for directed max-leaf problems," in *Proc. of the 16th Annual European Symposium on Algorithms*, Universität Karlsruhe, Germany, Sept. 2008, pp. 222–233.

[14] P. Bonsma and F. Zickfeld, "Spanning trees with many leaves in graphs without diamonds and blossoms," in *Proc. of the 8th Latin American Theoretical Informatics*, Búzios, rio de Janeiro, Brazil, Apr. 2008, pp. 531–543.

[15] Y. Boykov, O. Veksler, and R. Zabih, "Markov random fields with efficient approximations," in *Proc. of IEEE 1998 Conference on Computer Vision and Pattern Recognition*, Santa Barbara, CA, USA, Jun. 1998, pp. 648–655.

[16] G. Calinescu, H. Karloff, and Y. Rabani, “An improved approximation algorithm for multiway cut,” *Journal of Computer and System Sciences*, vol. 60, pp. 564–574, 2000.

[17] G. Chartrand and L. Lesniak, *Graphs & Digraphs*, 4th ed. Boca Raton: Chapman & Hall/CRC, 2004.

[18] J. Chen, F. V. Fomin, Y. Liu, S. Lu, and Y. Villanger, “Improved algorithms for the feedback vertex set problems,” in *Proc. of the 10th Workshop on Algorithms and Data Structures*, Halifax,Canada, Aug. 2007, pp. 422–433.

[19] J. Chen, D. K. Friesen, W. Jia, and I. A. Kanj, “Using nondeterminism to design efficient deterministic algorithms,” *Algorithmica*, vol. 40, pp. 83–97, 2004.

[20] J. Chen, I. Kanj, and W. Jia, “Vertex cover: further observations and further improvements,” *Journal of Algorithms*, vol. 41, pp. 280–301, 2001.

[21] J. Chen, I. A. Kanj, and G. Xia, “Improved parameterized upper bounds for vertex cover,” in *Proc. of the 31st International Symposium on Mathematical Foundations of Computer Science*, Bratislava, Slovak Republic, Aug. 2006, pp. 238–249.

[22] J. Chen and Y. Liu, “On the parameterized max-leaf problems: digraphs and undirected graphs,” Texas A&M University, Tech. Rep. 2008-12-1, 2008.

[23] J. Chen, Y. Liu, and S. Lu, “An improved parameterized algorithm for the minimum node multiway cut problem,” in *Proc. of the 10th Workshop on Algorithms and Data Structures*, Halifax,Canada, Aug. 2007, pp. 495–506.

[24] J. Chen, Y. Liu, S. Lu, B. O’Sullivan, and I. Razgon, “A fixed-parameter algorithm for the directed feedback vertex set problem,” in *Proc. of the 40th Annual*

ACM symposium on Theory of Computing, Victoria, Canada, May 2008, pp. 177–186.

[25] J. Chen, S. Lu, S. Sze, and F. Zhang, “Improved algorithms for path, matching and packing problems,” in *Proc. of the 18th Annual ACM Symposium on Discrete Algorithms*, New Orleans, Louisiana, USA, Jan. 2007, pp. 298–307.

[26] C. Chu and Y. Wong, “FLUTE: fast lookup table based rectilinear steiner minimal tree algorithm for VLSI design,” *IEEE Transaction on Computer-aided Design of Integrated Circuits and Systems*, vol. 27, no. 1, pp. 70–83, 2007.

[27] J. Cong, W. Labio, and N. Shivakumar, “Multi-way VLSI circuit partitioning based on dual net representation,” in *Proc. of IEEE 1994 International Conference on Computer-aided Design*, San Jose, California, USA, Nov. 1994, pp. 56–62.

[28] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. MIT Press, Cambridge, MA, 2001.

[29] W. Cunningham, “The optimal multiterminal cut problem,” *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 5, pp. 105–120, 1991.

[30] E. Dahlhaus, D. Johnson, C. Papadimitriou, P. Seymour, and M. Yannakakis, “The complexity of multiterminal cuts,” *SIAM Journal on Computing*, vol. 23, pp. 864–894, 1994.

[31] M. Davis and H. Putnam, “A computing procedure for quantification theory,” *Journal of the ACM*, vol. 7, pp. 201–215, 1960.

[32] F. Dehne, M. Fellows, F. Rosamond, and P. Shaw, “Greedy localization, iterative compression, and modeled crown reductions: new FPT techniques, and improved algorithm for set splitting, and a novel $2k$ kernelization for vertex cover,” in *Lecture Notes in Computer Science.* New York, USA: Heidelberg Springer, 2004, vol. 3162, pp. 271–280.

[33] F. K. H. A. Dehne, M. R. Fellows, M. A. Langston, F. A. Rosamond, and K. Stevens, “An $O(2^{O(k)}n^3)$ FPT algorithm for the undirected feedback vertex set problem,” in *Lecture Notes in Computer Science.* New York, USA: Heidelberg Springer, 2005, vol. 3595, pp. 859–869.

[34] R. Downey and M. Fellows, “Fixed-parameter tractability and completeness i: basic results,” *SIAM Journal on Computing*, vol. 24, pp. 873–921, 1995.

[35] R. G. Downey and M. R. Fellows, “Fixed parameter tractability and completeness,” in *Complexity Theory: Current Research.* New York, USA: Cambridge University Press, 1992, pp. 191–225.

[36] ——, “Parameterized computational feasibility,” in *Feasible Mathematics II.* Boston, USA: Birkhäuser, 1995, pp. 219–244.

[37] ——, *Parameterized Complexity.* New York, USA: Heidelberg Springer, 1999.

[38] R. G. Downey, M. R. Fellows, and M. A. Langston, “Forward by the guest editors,” *The Computer Journal*, vol. 51, no. 1, pp. 1–6, 2008.

[39] R. G. Downey, M. R. Fellows, and U. Stege, “Parameterized complexity: a framework for systematically confronting computational intractability,” in *DIMACS Series in Discrete Mathematics and Theoretical Computer Science.* Providence, RI, USA: American Mathematical Society, 1999, pp. 49–99.

[40] M. Drescher and A. Vetta, "An approximation algorithm for the maximum leaf spanning arborescence problem," Manuscript, http://digitool.Library.McGill.CA:8881/R/-?func=dbin-jump-full&object_id=18462&current_base=GEN01.

[41] G. Even, J. Naor, B. Schieber, and M. Sudan, "Approximating minimum feedback sets and multicuts in directed graphs," *Algorithmica*, vol. 20, pp. 151–174, 1998.

[42] M. R. Fellows, "Blow-ups, win/win's, and crown rules: some new directions in FPT," in *Graph-Theoretic Concepts in Computer Science.* New York, USA: Heidelberg Springer, 2003, pp. 1–12.

[43] M. R. Fellows, F. R. C. McCartin, and U. Stege, "Coordinated kernels and catalytic reductions: an improved FPT algorithm for max leaf spanning tree and other problems," in *Proc. of the 20th International Conference on Foundation of Software Technology and Theoretical Computer Science*, New Delhi, India, Dec. 2000, pp. 240–251.

[44] M. R. Fellows, C. Knauer, N. Nishimura, F. R. P. Ragde, U. Stege, D. M. Thilikos, and S. Whitesides, "Faster fixed-parameter tractable algorithms for matching and packing problems," *Algorithmica*, vol. 52, no. 2, pp. 167–176, 2008.

[45] M. Fellows, P. Heggernes, F. Rosamond, C. Sloper, and J. Telle, "Finding k disjoint triangles in an arbitrary graph," in *Lecture Notes in Computer Science.* New York, USA: Heidelberg Springer, 2004, vol. 3353, pp. 235–244.

[46] Q. Feng, Y. Liu, S. Lu, and J. Wang, "Improved deterministic algorithms for weighted matching and packing problems," in *Proc. of the 6th Annual Confer-*

ence on Theory and Applications of Models of Computation, Changsha, China, May 2009, pp. 211–220.

[47] P. Festa, P. M. Pardalos, and M. G. Resende, "Feedback set problems," in *Handbook of Combinatorial Optimization, Supplement Vol. A.* Dordrecht Kluwer Acad. Publ., 1999, pp. 209–258.

[48] F. V. Fomin, S. Gaspers, and A. V. Pyatkin, "Finding a minimum feedback vertex set in time $O(1.7548^n)$," in *Lecture Notes in Computer Science.* New York, USA: Heidelberg Springer, 2006, vol. 4169, pp. 184–191.

[49] F. V. Fomin, F. Grandoni, and D. Kratsch, "Measure and conquer: domination - a case study," in *Lecture Notes in Computer Science*, 2005, vol. 3580, pp. 191–203.

[50] ——, "Measure and conquer: a simple $O(2^{0.288n})$ independent set algorithm," in *Proc. of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, Miami, Florida, USA, Jan. 2006, pp. 18–25.

[51] F. Fomin, d. Lokshtanov, D. Raible, S. Saurabh, and Y. Villanger, "Kernel(s) for problems with no kernel: on out-trees with many leaves," in *Proc. of the 26th International Symposium on Theoretical Aspects of Computer Science*, Freiburg, Germany, Feb. 2009, pp. 421–432.

[52] G. Gardarin and S. Spaccapietra, "Integrity of databases: a general lockout algorithm with deadlock avoidance," in *Modeling in Data Base Management System.* Amsterdam: North-Holland, 1976, pp. 395–411.

[53] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* San Francisco, Freeman, 1979.

[54] A. V. Gelder, "A satisfiability tester for non-clausal propositional calculus," *information and Computation*, vol. 79, pp. 1–21, 1988.

[55] J. Gu, P. Purdom, and W. Wah, "Algorithms for the satisfiability (SAT) problem: a survey," in *Satisfiability Problem: Theory and Applications.* Providence, RI, USA: American Mathematical Society, 1997, pp. 19–152.

[56] S. Guha and S. Khuller, "Approximation algorithms for connected dominating sets," in *Proc. of the 4th Annual European Symposium on Algorithms*, Barcelona, Spain, Sept. 1996, pp. 179–193.

[57] J. Guo, F. H. J. Gramm, R. Niedermeier, and S. Wernicke, "Compression-based fixed-parameter algorithms for feedback vertex set and edge bipartization," *Journal of Computer and System Sciences*, vol. 72, no. 8, pp. 1386–1396, 2006.

[58] G. Gutin and A. Yeo, "Some parameterized problems on digraphs," *The Computer Journal*, vol. 51, no. 3, pp. 363–371, 2008.

[59] F. Henglein and H. Mairso, "The complexity of type inference for higher-order typed lambda calculi," in *Proc. of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Orlando, Florida, USA, Jan. 1991, pp. 119–130.

[60] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach.* San Francisco, CA: Morgan Kaufmann, 2007.

[61] E. Hirsh, "New worst-case upper bounds for sat," *Journal of Automated Reasoning*, vol. 24, pp. 397–420, 2000.

[62] ——, "Two new upper bounds for sat," in *Proc. of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, San Francisco, California, USA, Jan. 1998, pp. 521–530.

[63] D. Huston, A. L. Halpern, Z. Lai, E. W. Myers, K. Reinert, and G. G. Sutton, "Comparing assemblies using fragment and mate-pairs," in *Proc. of the 1st International Workshop on Algorithms in Bioinformatics*, University of Aarhus, Denmark, Aug. 2001, pp. 294–306.

[64] W. Jia, C. Zhang, and J. Chen, "An efficient parameterized algorithm for m-set packing," *Journal of Algorithms*, pp. 106–117, 2004.

[65] I. A. Kanj, M. J. Pelsmajer, and M. Schaefer, "Parameterized algorithms for feedback vertex set," in *Lecture Notes in Computer Science.* New York, USA: Heidelberg Springer, 2004, vol. 3162, pp. 235–247.

[66] D. Karger, P. Klein, C. Stein, M. Thorup, and N. Young, "Rounding algorithms for a geometric embedding of minimum multiway cut," in *Proc. of the 31th Annual ACM Symposium on Theory of Computing*, Atlanta, Georgia, USA, May 1999, pp. 668–678.

[67] D. Karger and M. Levine, "Finding maximum flows in undirected graphs seems easier than bipartite matching," in *Proc. of the 30th Annual ACM Symposium on Theory of Computing*, Dallas, Texas, USA, 1998, pp. 69–78.

[68] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations.* Plenum, 1972, pp. 85–103.

[69] J. Kneis, A. Langer, and P. Rossmanith, "A new algorithm for finding trees with many leaves," in *Lecture Notes in Computer Science.* New York, USA:

Heidelberg Springer, 2008, vol. 5369, pp. 270–281.

[70] J. Kneis, D. Molle, S. Richter, and P. Rossmanith, "Divide-and-color," in *Graph-Theoretic Concepts in Computer Science.* New York, USA: Heidelberg Springer, 2006, pp. 58–67.

[71] I. Koutis, "A faster parameterized algorithm for set packing," *Information Processing Letters*, vol. 94, pp. 7–9, 2005.

[72] ——, "Faster algebraic algorithms for path and packing problems," in *Proc. of the 35th International Colloquium on Automata, Languages and Programming*, Reykjavik, Iceland, July 2008, pp. 575–586.

[73] O. Kullmann and H. Luckhardt, "Deciding propositional tautologies: algorithms and their complexity," Manuscript, 1997, https://eprints.kfupm.edu.sa/33648/1/33648.pdf.

[74] J. L. Ford and D. Fulkerson, *Flows in Networks.* Princeton University Press, Princeton, NJ, 1962.

[75] T. Leighton and S. Rao, "An approximation max-flow min-cut theorem for uniform multi-commodity flow problems with applications to approximation algorithms," *Journal of the ACM*, vol. 46, pp. 787–832, 1999.

[76] C. Leiserson and J. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, pp. 5–35, 1991.

[77] O. Lichtenstein and A. Pneuli, "Checking the finite-state concurrent programs satisfy their linear specification," in *Proc. of the 12th ACM Symposium on Principles of programming Languages*, New Orleans, Louisiana, USA, Jan. 1985, pp. 97–107.

[78] Y. Liu, S. Lu, J. Chen, and S. Sze, “Greedy localization and color-coding: improved matching and packing algorithms,” in *Proc. of 2nd International Workshop on Parameterized and Exact Computation*, Zürich, Switzerland, Sept. 2006, pp. 85–95.

[79] H.-I. Lu and R. Ravi, “Approximation maximum leaf spanning trees in almost linear time,” *Journal of Algorithms*, vol. 29, pp. 132–141, 1998.

[80] M. Wahlstöm, “An algorithm for the sat problem for formulae of linear length,” in *Lecture Notes in Computer Science.* New York, USA: Heidelberg Springer, 2005, vol. 3669, pp. 107–118.

[81] ——, “Faster exact solving of sat formulae with a low number of occurrences per variable,” in *Lecture Notes in Computer Science.* New York, USA: Heidelberg Springer, 2005, vol. 3569, pp. 309–323.

[82] D. Mark, “Parameterized graph separation problems,” *Theoretical Computer Science*, vol. 351, pp. 394–406, 2006.

[83] L. Mathieson, E. Prieto, and P. Shaw, “Packing edge disjoint triangles: a parameterized view,” in *Lecture Notes in Computer Science.* New York, USA: Heidelberg Springer, 2004, vol. 3162, pp. 127–137.

[84] N. Alon and O. Goldreich and J. Hästad and R. Peralta, “Simple constructions of almost k-wise independent random variables,” *Random Structures and Algorithms*, vol. 3, pp. 289–304, 1992.

[85] J. Naor and L. Zosin, “A 2-approximation algorithm for the directed multiway cut problem,” *SIAM Journal on Computing*, vol. 31, pp. 477–482, 2001.

[86] E. Prieto and C. Sloper, "Looking at the stars," *Theoretical Computer Science*, vol. 351, pp. 437–445, 2006.

[87] V. Raman, S. Saurabh, and C. R. Subramanian, "Faster fixed parameter tractable algorithms for undirected feedback vertex set," in *Lecture Notes in Computer Science.* New York, USA: Heidelberg Springer, 2002, vol. 2518, pp. 241–248.

[88] ——, "Faster fixed parameter tractable algorithms for finding feedback vertex set," *ACM Transaction on Algorithms*, vol. 2, no. 3, pp. 403–415, 2006.

[89] I. Razgon, "Exact computation of maximum induced forest," in *Lecture Notes in Computer Science.* New York, USA: Heidelberg Springer, 2006, vol. 4059, pp. 160–171.

[90] B. Reed, K. Smith, and A. Vetta, "Finding odd cycle transversals," *Operations Research Letters*, vol. 32, no. 4, pp. 299–301, 2004.

[91] J. M. Robson, "Finding a maximum independent set in time $O(2^{n/4})$," Manuscript, 2001, http://www.labri.fr/perso/robson/mis/techrep.html.

[92] J. Schmidt and A. Siegel, "The spatial complexity of oblivious k-probe hash functions," *SIAM Journal on Computing*, vol. 19, pp. 775–786, 1990.

[93] A. Schrijver, *Combinatorial Optimization.* Berlin: Springer-Verlag, 2003.

[94] A. Silberschatz and P. Galvin, *Operating System Concepts*, 4th ed. Boston, Massachusetts, USA: Addison-Wesley Longman Publishing Co., Inc., 1994.

[95] R. Solis-Ob, "2-approximation algorithm for finding a spanning tree with maximum number of leaves," in *Lecture Notes of Computer Science*, 1998, vol. 1461, pp. 441–452.

[96] H. Stone, “Multiprocessor scheduling with the aid of network flow algorithms,” *IEEE Transaction on Software Engineering*, vol. 3, pp. 85–93, 1977.

[97] M. Thai, F. Wang, D. Liu, S. Zhu, and D. Du, “Connected dominating sets in wireless networks with different transmission range,” *IEEE Transaction on Mobile Computing*, vol. 6, pp. 721–730, 2007.

[98] S. Thomassé, “A quadratic kernel for feedback vertex set,” in *Proc. of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms*, New York, USA, Jan. 2009, pp. 115–119.

[99] U. Schöning, “Algorithms in exponential time,” in *Lecture Notes in Computer Science*. New York, USA: Heidelberg Springer, 2005, vol. 3404, pp. 36–43.

[100] D. M. Warme, “A new exact algorithm for rectilinear steiner trees,” System Simulation Solutions, Inc., Alexandria, VA, Tech. Rep. 22153, 1997.

[101] G. J. Woeginger, “Exact algorithms for NP-hard problems: a survey,” in *Combinatorial Optimization - Eureka, You Shrink!* New York, USA: Heidelberg Springer, 2003, pp. 185–207.

[102] B. Wu and K. Chao, *Spanning Trees and Optimization Problems.* Boca Raton: Chapman & Hall/CRC, 2003.

[103] M. Xie, J. Wang, and J. Chen, “A model of higher accuracy for the individual haplotyping problem based on weighted snp fragments and genotype with errors,” *Bioinformatics*, vol. 24, no. 13, pp. 105–113, 2008.

VITA

Name: Yang Liu

Address: Department of Computer Science, University of Texas-Pan American,
Edinburg, TX 78539

Email: yliu@cs.panam.edu

Education: Ph.D. in Computer Science, Texas A&M University, 2009
M.S. in Electrical and Computer Engineering, Rose-Hulman Institute of Technology, 2005
B.S. in Electrical Engineering, Zhejiang University, China, 1997