MAXIMUM SATISFIABILITY AND SEPARATOR DECOMPOSITION OF GRAPHS

by

Cong Wang

B.S., Simon Fraser University, 2007

a Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in the School
of
Computing Science

APPROVAL

Name:	Cong Wang
Degree:	Master of Science
Title of Thesis:	Maximum Satisfiability and Separator Decomposition of Graphs

Examining Committee: Dr. Art Liestman
Chair

____
Dr. Andrei Bulatov, Senior Supervisor

____
Dr. Binay Bhattacharya, Supervisor

____
Dr. Qianping Gu, Examiner

Date Approved: August 4, 2009

Abstract

The Satisfiability (SAT) problem asks to find a satisfying assignment to a Boolean formula in CNF, while in its optimization version, MAXSAT, the object is to satisfy the maximum number of clauses. Much effort has been dedicated to design efficient solvers for these problems. In this thesis, we focus on designing algorithms for solving MAX2SAT problem where all clauses contain at most two literals. We use graphs to represent Boolean formulas, and consider problem instances with different structural restrictions. We compare two exact algorithms for MAX2SAT of bounded tree width. Based on our observations, we define a new measure, separator width, which can be less than tree width by up to a logarithmic factor. Moreover, we also design an approximation algorithm using the elimination ordering of variables. We show both experimentally and theoretically that our approximation algorithm works well on a large class of graphs, namely, d-degenerate graphs.

To my dearest grandmother, Song Li

Acknowledgments

I would like to acknowledge the considerable academic and financial support of my senior supervisor Dr. Andrei Bulatov. As well, I would like to thank Dr. Binay Bhattacharya for all the advices and encouragements. I would also like to thank Dr. Qianping Gu for his suggestions and patience. The experiment on the model in Section 4.3 is completed with the help from Evgeny Skvortsov. My deepest gratitude goes to my family for their love and support throughout my life. I would like to thank my father, Shiwen Wang, for all the sacrifices he made for me. And also, my mother, Xinmei Li, for all the encouragements and care.

Contents

Approval ii

Abstract iii

Dedication iv

Acknowledgments v

Contents vi

List of Tables viii

List of Figures ix

1 Introduction 1

1.1 Overview 2

1.2 Basic Definitions 3

1.3 Previous results 5

1.3.1 MAXSAT 5

1.3.2 Tree decomposition 9

2 Exact Algorithms 13

2.1 Dynamic programming 13

2.2 Divide and conquer 15

2.3 Modified Divide and conquer 16

2.4 Balancing path decomposition 18

3 Separators **21**

3.1 Separator Width 21

3.2 Relationship of separator width and tree width 24

3.3 Hardness of determining an optimal balanced separator tree 26

3.4 Applications 29

4 Approximation Algorithm **32**

4.1 Algorithm 32

4.2 Experimental analysis 35

4.3 Mathematical model 37

5 Conclusion **46**

Bibliography **48**

List of Tables

3.1 Time and space complexity 31
3.2 Time and space complexity 31

4.1 Performance of the three approximation algorithms 36
4.2 Result of formulas with 100 variables 44
4.3 Result of formulas with 1000 variables 44

List of Figures

1.1 A graph and its tree decomposition 5

2.1 Transform a path to a tree 19

3.1 A balanced separator tree 23
3.2 (1) A graph. (2), (3) Two different balanced separator trees. 23
3.3 (1) A graph. (2) Tree decomposition of the graph. (3) Balanced separator tree. 24
3.4 (1) A graph. (2) An optimal balanced separator tree. (3)A balanced separator tree. 27
3.5 Approximation algorithm 28

4.1 Four sets of clauses 33
4.2 Flow Diagram 40

Chapter 1

Introduction

The Satisfiability problem (SAT) is an important problem both in theory and in practice. In this problem, we are asked to find a satisfying assignment to a Boolean formula in CNF. Despite the fact that it can not be solved efficiently in the worst case unless $P = NP$, many SAT solvers are shown to perform well in practice. Researchers are investigating this phenomenon from many different aspects, but so far, there is no clear result explaining it. In order to design a robust algorithm for SAT, it is important to distinguish instances which can be solved efficiently from others which are not polynomial time solvable. It is known in graph theory that structural parameters such as, tree width, play an important role in classifying solvable instances of many NP-complete problems. Inspired by these results, in this thesis, we explore different structural parameters, and design algorithms for solving SAT instances using these parameters.

This work is to a large extent motivated by a problem encountered by D-wave company that is developing a working prototype of a quantum computer. The difficulty is that the device they are working on cannot possibly allocate real life instances, and therefore such instances should be split into subproblem. Then each subproblem can be solved by a quantum algorithm. However, the interface between subproblems remain classical. Assuming the quantum device works properly, one implication of that is every subproblem can be solved in a matter of nanoseconds, the amount of data that has to be carried between subproblems becomes a bottleneck.

Many decomposition methods can be used to split a problem into subproblems. The tree decomposition method is one of the most standard one. In classical setting, many NP-complete problem instances of bounded tree width can be solved efficiently using dynamic programming algorithm. However, this algorithm doesn't work well in quantum setting because the amount of data that needs to be carried between subproblems is usually exponential in terms of tree width. Therefore, when designing algorithms, we are willing to trade the time for memory.

1.1 Overview

SAT is a well known NP-complete problem, and therefor any problem in NP can be reduced to SAT in polynomial time. In practice, many of these NP problems are often solved more efficiently by reducing to SAT and then using one of the existing SAT solvers. This property makes SAT especially interesting and important to study. After 1971 when the problem was first proved to be NP-complete by Stephen Cook [14], numerous studies were carried out in SAT and related field. In the standard version of SAT, we are asked to find a satisfying assignment of a boolean formula. The currently known best upper bound for SAT is up to a polynomial factor of $2^{n(1-\frac{1}{\alpha})}$ where $\alpha = \ln(m/n) + O(\ln \ln m)$ and m, n are the number of clauses and the number of variables in a given formula respectively. This result is obtained by a randomized algorithm which was proposed by Dantsin and Wolpert [17] in 2005.

Although SAT is hard to solve in general, there are also special cases of SAT which are polynomial time solvable. For example, 2SAT which is SAT with restrictions of at most two literals in each clause is one of these polynomial solvable cases. As shown by Aspvall, Plass & Tarjan [6] in 1979, 2SAT is satisfiable if and only if every variable and its negation belong to different strongly connected components in the implication graph. An implication graph of a boolean formula is a directed graph $G(V, E)$, where V represents the truth status of a Boolean literal, and each directed edge from vertex

u to vertex v represents the implication "If the literal u is true then the literal v is also true". Since strongly connected components of a graph can be found in linear time, 2SAT is also solvable in linear time. However, limiting the number of literals in each clause doesn't necessarily make the problem easier to solve. 3SAT which has at most three literals in each clause is known to be NP-complete.

There are also some special structural properties that makes some instances of SAT easy to solve. For example, if the graph of the formula (implication graph without orientations) of SAT is a tree, then the formula is always satisfiable. A satisfying assignment can be found by recursively assigning truth values to the leave nodes of the tree. There are also many NP-complete combinatorial problems in graph theory, such as, Maximum Independent Subset problem, Graph coloring problem, and so on, that are easy to solve on trees. In 1983, Robertson and Seymour [29] introduced a structural parameter called treewidth which measures how much does a graph resemble a tree structure. Later, several researchers [5], [7], [8], [15] observed independently that some NP-complete combinatorial problems that are easy to solve on trees are solvable in polynomial time when restricted to graphs of bounded treewidth. This is what motivates us to investigate the bounded treewidth instances of SAT.

There are several different versions and extensions of SAT. For example, there is the decision version of SAT which answers the question whether a given formula is satisfiable or not; there is the counting version of SAT which counts the number of satisfying assignments of a given formula; there is also the optimization version of SAT which determines an assignment that satisfies the maximum number of clauses in a given formula. In this thesis, we are focusing on the optimization version of SAT which is also known as MAXSAT.

1.2 Basic Definitions

MAXSAT

A Boolean formula is in conjunctive normal form (CNF) if it is a conjunction of clauses and each clause is a disjunction of variables or their negations. A formula is in kCNF if it is in CNF and each clause of the formula contains no more than k variables. Let $F := C_1 \wedge C_2 \wedge ... \wedge C_m$ be a Boolean CNF formula with variables $\{x_1, x_2, ...x_n\}$. An assignment of F is a mapping $\phi : V \rightarrow \{0, 1\}$ where 0, 1 stand for false and true respectively. Let $S(\phi, F)$ denote the number of clauses in F satisfied by ϕ. The maximum satisfiability problem is an optimization problem that determines a truth assignment which satisfy the maximum number of clauses in F. If F is in kCNF form, then the problem is known as MAXkSAT. In the MAX2SAT problem, the boolean formula is given in 2CNF form. Instead of solving the general MAXkSAT, we focus on solving MAX2SAT in this thesis.

Treewidth, pathwidth and partial k-tree

According to Robertson and Seymour [29], a tree decomposition of $G(V, E)$ is a pair $(X = \{X_i | i \in I\}, T = (I, H))$ with X a family of subsets of V(also known as bags), one for each node of T, and T a tree such that

- $\bigcup_{i \in I} X_i = V$.
- for all edges $(v, w) \in E$, $\exists i \in I$ with $v \in X_i$ and $w \in X_i$.
- for all i, j, $l \in I$: if j is on the path from i to l in T then $X_i \cap X_l \subseteq X_j$

In other words, a tree decomposition satisfies the following two properties:

- **P1** All vertices in V and all edges in E are contained in some bags of T.
- **P2** If a vertex v appears in some bags X_i and X_l then v must appear in any bag X_j that is along the path from X_i to X_l

The graph $G_F(V, E)$ of a formula F is a graph with vertex set $V := \{x_1, x_2, ..., x_n\}$, and edge set $E := \{\{x_i, x_j\} | x_i, x_j \in V$ and x_i, x_j appear in the same clause in $F\}$. The tree decomposition of $G_F(V, E)$ can be defined in exactly the same way as above.

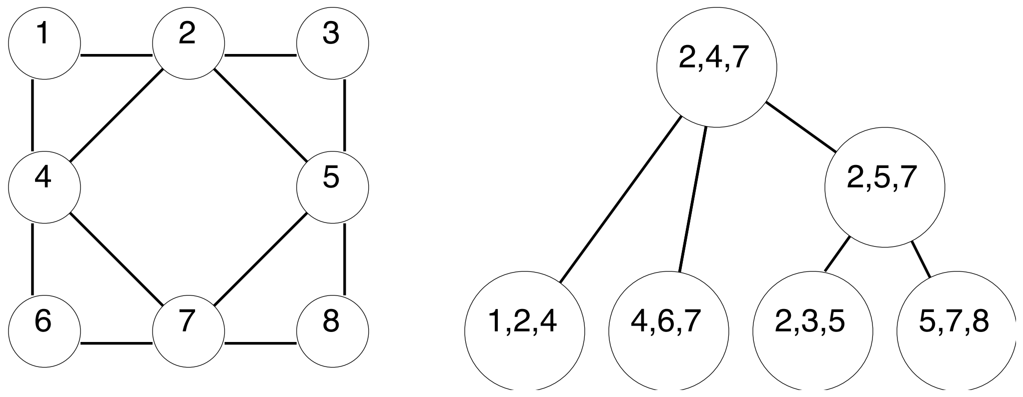

Figure 1.1: A graph and its tree decomposition

For a given tree decomposition (X, T), the width of the decomposition, $tw((X, T))$, is $max_{i \in I}\{|X_i| - 1\}$. The treewidth of a graph G, $tw(G)$, is the minimum width over all tree decompositions of G. A graph of treewidth at most k is also known as partial k-tree.

A tree decomposition of width k is smooth [9] if and only if for any adjacent bags X_i, X_j, we have $|X_i \cap X_j| = k$. It is also known that any tree decomposition can be transformed into a smooth tree decomposition with the same width in linear time.

A path decomposition is a tree decomposition (X, T) such that T is a path. Similarly, the pathwidth of a graph G, $pw(G)$, is the minimum pathwidth over all path decompositions of G. From the definition, we know every path decomposition is also a tree decomposition, therefore, $pw(G)$ is at least $tw(G)$ for any graph G.

1.3 Previous results

1.3.1 MAXSAT

As mentioned in Section 1.2, MAXkSAT is the optimization version of SAT that determines an assignment which satisfies maximum number of clauses. MAX2SAT which is known to be NP-hard [20] is especially interesting because 3SAT can be reduced to it in polynomial time. Hence, if we can solve MAX2SAT more efficiently, it is possible to improve the running time for 3SAT as well.

Exact algorithm

Some of the studies on MAX2SAT are focused on designing exact algorithms. Let n denote the number of variables in a formula and m denote the number of clauses. The first nontrivial upper bounded on MAX2SAT is $2^{n/1.261}$ which was proved by Williams [34]. Later, a new upper bound of $2^{m/5.5}$ is given by Kojevnikov and Kulikov [24]. Very Recently, Raible and Fernau [27] refined the algorithm given by Kojevnikov and Kulikov [24] and show an upper bound of $2^{m/6.2158}$.

Backtracking and BB (branch and bound) are the two commonly used approaches in designing exact algorithms for solving MAX2SAT. DPLL (Davis-Putnam-Logemann-Loveland) [18] is a backtracking based algorithm which is enhanced by the following two rules,

- **Pure literal elimination** A literal is pure if and only if the formula contains only one polarity of the literal. All clauses that contain a pure literal can be satisfied by assigning a truth value that satisfies the pure literal.

- **Unit propagation** A unit clause contains a single unassigned literal. All unit clauses can be satisfied by assign a truth value that satisfies the single literal. The algorithm backtracks if the resulting formula contains a contradiction.

Besides these two rules, other more sophisticated techniques were introduced in later studies to improve the performance of the DPLL algorithm.

BB is another approach used in designing algorithms for solving MAX2SAT. In BB based algorithm, we systematically enumerate all candidate solutions and use a lower bound function to discard subsets of infeasible candidates. A lower bound function is the function that takes a problem instance as input and determines the threshold such that a candidate solution is infeasible if its value is under the threshold. Studies of BB based algorithms are mostly focused on either improving the lower bound function or simplifying the problem instance. By improving the lower bound, we tighter the search space which leads to improvements of the performance. There are trivial lower bound functions, such as the number of conflicting clauses of the current partial assignment [12]. There are also more sophisticated ones that take the minimum over several lower bound functions [1]. To simplify the problem instance, we need to introduce rules that reduce the size of the formula. For example, one of the rules defined in [32] says, if $F = \{x \vee y\} \wedge \{x \vee \overline{y}\} \wedge \{\overline{x} \vee z\} \wedge \{\overline{x} \vee \overline{z}\} \wedge F'$, then the OPT($F$)=OPT($F'$)+1 where OPT($F$) is the number of unsatisfied clauses in the optimal assignment for formula F. Using this rule, we can reduce the problem of solving F to solving F'.

Approximation algorithm

In an optimization problem, we are given a cost function f, and asked to find a solution to the problem that minimize or maximize the cost function. This solution is also know as the optimal solution. Approximation algorithms are used to find approximate solutions to NP-hard optimization problems. An approximation algorithm A to a minimization (maximization) problem has an approximation ratio β if the value of the approximate solution $A(x)$ to an instance x is no more (less) than β times the optimal solution.

Approximation algorithms for MAX2SAT is another area that generates many interesting results. The cost function of MAX2SAT is the number of unsatisfied clauses. It was shown by Hastad that MAX2SAT can not be approximated within an approximation ratio that is better than $\frac{21}{22}$ in polynomial time. The current best known approximation ratio is 0.935 which is given by an approximation algorithm designed by Matuura and Matsui [25]. Two of the most well known approximation algorithms for MAX2SAT are local search (LS) and GSAT.

Among many of the approximation algorithms, the LS algorithm plays a fundamental role. The LS heuristic uses the idea that the current solution can be improved by making small changes, for instance, flipping a single variable in the assignment. LS algorithm usually starts with an random solution and keeps flipping the solution if its value can be improved. The algorithm terminates when no improvement can be made. For any MAXkSAT problem, it was proven by Hansen and Jaumard [21] that a lower bound for local optimum is $\frac{km}{k+1}$ where m is the number of clauses. Since LS stops that a local optimum, it implies that the solution of a LS algorithm satisfies at least $\frac{km}{k+1}$ clauses. Although the idea of LS seems very simple, surprisingly, its expected performance is much better than the $\frac{km}{k+1}$ bound.

Besides LS, GSAT is another widely used approximation algorithm for MAXSAT. Unlike classic LS algorithms, not all flips of variables in GSAT result in improvements of the current solution. Instead of making improvements at each step of the algorithm and terminating when no improvement can be made, GSAT starts with an initial assignment, and at each step, it evaluates all possible moves, selects the best move (even if it worsens the assignment) which minimizes the cost function to avoid local maximum. There is no natural condition that can be used for GSAT to terminate. Instead, it usually runs for a specified number of steps (flips of variables) given by the parameter max_flip.

Another parameter which plays an important role in both LS and GSAT algorithms is max_try. It allows the algorithm to restart for certain number of times with a different initial assignment after termination. However, this parameter is not required for all algorithms. For example, the one-pass local search (OLS) algorithm which is used in the experiment in Section 4.2 doesn't need to restart. It evaluates each variable of a given formula according to some predefined order. Each variable is considered once and set to a truth value according to some predefined conditions.

Random formulas

Random formulas are often considered for the purpose of testing the performance of SAT solvers. In recent years, increasing number of researches are focused on designing good models for generating random formulas. Uniformly random formulas is one of the most common models. In Section 4, we use a modified uniformly random model to generate instances for our experiments.

The uniformly random model usually take three parameters, the number of variables n, the number of clauses m, and the number of variables per clause k. Each clause is generated by selecting k distinct variables uniformly at random. Each variable in the clause is negated with probability $\frac{1}{2}$. The process stops after generating m clauses. In this model, all formulas with n variables and m clauses appear with equal probability.

Beside the classic uniformly random model, there are many other models for different experimental purposes. There are models that generate hard instances of MAXSAT problem which can be used for testing worst case performance. Formulas generated by these models consist two parts: random and adversarial. While the random part might be polynomial time solvable, the adversarial part ensures that the formula is hard to solve with high probability. There are models that generate random instances such that a given optimal solution is hidden in the formula. Random instances generated by these models are useful in experimental analysis of the distances between approximated solutions and real solutions. In the experiments in Section 4.2, we use a model similar to the uniformly random model to generate testing formulas.

1.3.2 Tree decomposition

Given an arbitrary graph G, it is NP-complete to determine its tree width and to find its tree decomposition with minimum width [4]. For a given fixed k, many works have been done on determining a tree decomposition with width at most k. It is known that for small $k = 1, 2, 3, 4$, this problem is linear time solvable. For constant k, there are algorithms that construct a tree decomposition of width at most k if it exists. Such algorithms are usually polynomial in n, but exponential in k. We should notice that k is not the minimum tree width of G in most cases.

Decomposition algorithm

For a fixed constant k, the algorithm designed by Arnborg, Corneil, and Proskurowski [4], is one of the earliest polynomial time algorithms that constructs a tree decomposition of width k. In the first step of the algorithm, it determines all size k vertex separators of a graph G which partition the graph into disjoint components. Each separator divides the graph into several disconnected components of various sizes. In the next step, it lists all components of all separators from the smallest in size to the largest, then uses dynamic programming to determine the decomposable components. Since there are approximately $O(n^k)$ size k separators, and it takes $O(n^2)$ time to determine the decomposable components for each separator, the total running time of the algorithm is $O(n^{k+2})$. Although the running time of this algorithm is polynomial in n, but for graphs with large tree width or large size, it is infeasible to construct the tree decomposition.

A much more efficient tree decomposition algorithm was developed by Robertson and Seymour [31] which achieves the running time of $O(n^2)$ for fixed k. Later, Bodlaender and Kloks [11] improved the running time of the algorithm to $O(n \log^2 n)$. Reed [28] further improved the running time to $O(n \log n)$ based on his linear time algorithm for determining approximately balanced separators. Bodlaender [9] also designed a linear time algorithm for finding minimum-width tree decomposition for graphs of bounded tree width. However, there is a large constant hiding in the running time of the algorithm, so the algorithm is not practical. In practice, there are also many heuristic algorithms for finding tree decompositions [2], [26], [33].

Balanced Separator

Given an arbitrary graph with n vertices, an α-separator of the graph is a vertex set such that the removal of the set disconnects the graph into connected components of size no more than αn. When α is at most $\frac{1}{2}$, it is called balanced separator. The problem of finding an α-separator of minimum size is NP-hard for any $\alpha < 1$. Since finding a minimum α-separator is important for improving the efficiency of many algorithms that are based on divide and conquer techniques, many studies have been done on designing approximation algorithms. The approximation may be in terms of the size of the separator, in terms of the value of α, or even both. In our case, we are interested in finding small α-separator with some relaxation of balance requirement. For $\frac{2}{3} \leq \alpha < 1$, Feige and Mahdian [19] developed an $O(2^k n)$ randomized algorithm for finding an α-separator of size k.

Given a fixed k, the linear time algorithm given by Reed [28], determines a $\frac{3}{4}$-separator of size no more than k. As mentioned above, we can determine a tree decomposition of a graph in $O(n \log n)$ by recursively applying this separator algorithm. In Section 4, we also use this algorithm as a subroutine for constructing a separator tree. This approximately balanced separator algorithm proceeds as follows. In the first step, it finds a partition of the graph into disjoint rooted trees $T = \{t_1, t_2, ..., t_s\}$ with roots $R = \{r_1, r_2, ..., r_s\}$, such that $s \leq 24k$ and the size of connected components of $t_i \setminus \{r_i\}$ is most $\frac{n}{24k}$. In the next step, we check all possible partitions of R for an approximately balanced separator since if a cut set partitions R, then it also partitions G.

Other Decomposition

Besides tree decomposition, there are many other decomposition techniques that partition a graph according to different constraints. Similar to the notion of tree width, different structural parameters can be defined based on these decompositions. These structural parameters are usually used to identify classes of structures or graphs which are algorithmically well behaved. It was shown that some NP-complete problems are fixed parameter tractable or polynomial time solvable under some of these structural parameters. Some of the well known width measures are branch width, clique width, Kelly width.

The notion of branch decomposition which was introduced by Robertson and Seymour [30] is closely related to tree decomposition. It was shown to be effective for solving some combinatorial optimization problems, such as, general minor containment, ring-routing problem, and the traveling salesman problem. A branch decomposition of a graph G is a pair (T, v), such that T is a ternary tree with $|E(G)|$ leaves, and v is a bijection from the edges of G to the leaves of T. By removing an edge e from T, we can partition the edges of G into two sets A_e and B_e. The width of e is the number of vertices that are incident to both edges in A_e and B_e. The width of a branch decomposition is the maximum width among all edges of the decomposition. The branch width of a graph G, $bw(G)$, is the minimum width over all branch decompositions of G. It is known [30] that $bw(G) \leq tw(G) + 1 \leq \lfloor \frac{3}{2} bw(G) \rfloor$.

Another well known width measure is clique width, $cw(G)$, which was introduced by Courcelle and Olariu [16]. It is known to be effective for solving problems such as, edge domination set, graph coloring. As defined in [16], $cw(G)$ is the minimum integer k such that G can be constructed by means of repeated application of the following four operations: introduce, disjoint union, relabel, and join. For any graph with tree width k, it is clique width is at most $3 * 2^{k-1}$. For any k, there is a graph with tree width k, where its clique width is at least $2^{\lfloor k/2 \rfloor - 1}$ [16].

Width measure can also be defined for directed graphs. Kelly width of a digraph G, $kw(G)$, is the minimum width of Kelly decomposition which was introduced by Hunter and Kreutzer [23]. Since we are not going to use the notion of kelly decomposition, so we are not defining it here. It is known that problems such as parity game, weighted hamiltonian cycle, can be solved efficiently on small Kelly width instances. It is also known [23] that $kw(G) \leq tw(G) + 1$.

When a new width parameter is defined, it is usually compared against tree width. As we can observe from above, both branch width and Kelly width are within constant factor of tree width, and clique width can be exponentially worse than tree width. However, the separator width which is defined in Section 3 is no more than the tree width and can differ from the tree width by a logarithm factor.

Chapter 2

Exact Algorithms

In this chapter, we investigate two different approaches in designing exact algorithms for solving MAX2SAT instances with bounded tree width. In both approaches, we take a tree decomposition of the formula as input. In the dynamic programming approach, we explore the tree from bottom-up, whereas, in the divide and conquer approach we explore the tree from top-down. We also give an algorithm that transforms a path decomposition of width k into a balanced tree decomposition of width at most $3k$.

2.1 Dynamic programming

Dynamic programming is the most common approach in designing tree decomposition based algorithms. Dynamic programming algorithms often take advantages of bounded tree width to build tables for storing partial results and avoiding redundant computations. Problems such as independent sets, dominating sets, graph coloring, optimal subgraphs can all be solved in polynomial time for graphs with bounded tree width. The fundamental result of [15] shows that problems that are expressible in existential monadic second order logic are solvable by dynamic programming algorithm in polynomial time on classes of bounded tree width. A MAX2SAT instance has bounded tree width if the graph of the formula has bounded tree width. Because the MAX2SAT problem is expressible in monadic second order logic, same concept can be used in algorithms for solving MAX2SAT problem.

Let k denote the width a tree decomposition. Since the size of each bag in the tree decomposition is at most $k+1$, we can determine all possible assignments of a bag in $O(2^{k+1})$ time. For each bag X_i in the tree decomposition, we can determine the optimal assignment of the subtree rooted at X_i from the information of all solutions of its left and right subtrees.

For a given tree decomposition (X, T), let L_i denote the set of all possible assignments of variables in X_i and their costs on the subtree rooted at X_i. The cost of an assignment on a subtree is determined by the minimum number of unsatisfied clauses containing only variables from the subtree. The dynamic programing algorithm first starts from a leaf bag X_i, and stores all possible assignments of X_i and their costs in L_i. Secondly, L_i for an internal bag X_i with children X_j and X_l can be obtained by combining each assignment of X_i, with assignments from L_j and L_l separately. The following rule defines how to combine assignments.

Rule 2.1.1 (Combining an assignment) *Let ϕ_i be an assignment in X_i. Let S be the set of assignments ϕ_j in L_j such that ϕ_i and ϕ_j coincide on variables in $X_i \cap X_j$. Similarly, let Q be the set of assignments ϕ_l in L_l such that ϕ_i and ϕ_l coincide on variables in $X_i \cap X_l$. Let $\phi_j \in S$ be the assignment which have the minimum cost c_j,and $\phi_l \in Q$ be the assignment which have the minimum cost c_l. We store ϕ_i in L_i and the cost of ϕ_i is $c + c_i + c_j$ where c is the number of unsatisfied clauses containing only variables in X_i.*

From property P2 of tree decomposition defined in Section 1.2, we know that variables in $(X_j \cup X_l) \setminus X_i$ can never appear in any of the unprocessed bags. Therefore, in L_i, we only need to store the assignments of variables in X_i. The algorithm terminates when we reach the root bag X_r of (X, T), the best assignment in L_r is the partial assignment corresponds to the optimal assignment of (X, T).

For a bag X_i with children X_j and X_l, the total time it takes to determine the list L_i is $2^{|X_i \setminus X_j|} * 2^{|X_j|} + 2^{|X_i \setminus X_l|} * 2^{|X_l|}$. Since each bag of a tree decomposition contains at most $k+1$ variables, in the worst case, it takes $O(2^{2(k+1)+1})$ time to combine all solutions in an internal bag with its two children. We know that a tree decomposition contains at most n bags, so the total running time of the algorithm is $O(2^{2k}n)$. We should also notice, when the tree decomposition is smooth, the algorithm have time complexity $O(2^{k+3}n)$. Since we need to store all assignments of the child nodes, and the list can be discarded only after their parent nodes are processed, the total memory space used by this algorithm is $O(2^{k+1})$ (and the constant factor is relatively small). This algorithm is very efficient when k is small. However, when k is large, a huge amount of partial solutions is carried in between each pair of adjacent bags. Most importantly, many of the partial solutions stored in the algorithm can not be extended to an optimal solution at all. In next section, we look at the divide and conquer algorithm which is less efficient than dynamic programming algorithm, but uses only linear memory space.

2.2 Divide and conquer

Instead of using a bottom-up dynamic programming algorithm, we can also use a top-down divide and conquer algorithm to solve this problem. By assigning a truth value to all the variables in the root bag, we can partition the formula into two independent sub-formulas. Therefore, we can solve this problem by assigning each of the 2^k possible assignments to the root bag and recursively solving the two sub-formulas. This algorithm takes advantages of the tree like structure of the decomposition. Once a particular assignment is given to the root, we are left with several smaller instances of the problem which can be solved independently.

Unlike dynamic programming algorithm, at each step in the divide and conquer algorithm, we only store the current optimal assignment. This implies that the memory space used by the algorithm is linear. This also means when a different assignment is given to a root bag, we will need to recompute the optimal solutions of each subproblems. Although this requires a lot of recomputation, we can still efficiently solve the problem if we can split the problem into more or less equal

Algorithm 2.1 DCSolver(r)

```
Require: A tree decomposition $(X, T)$ with root $r$
Ensure: An optimal assignment $\phi$ on the subtree rooted at $r$
 1: $\phi$:=null
 2: for All $s \in$ possible assignments of $X_r$ do
 3:    if r.leftChild!=null then
 4:       $l$:=DCSolver(r.leftChild);
 5:    end if
 6:    if r.rightChild!=null then
 7:       $r$:=DCSolver(r.rightChild);
 8:    end if
 9:    $\phi$ :=best($\phi$, $s \cup l \cup r$);
10: end for
11: return $\phi$
```

sized subproblems. Let $T(n)$ denote the total running time of this algorithm, then $T(n) = (2^{k+1})T(n/c) + (2^{k+1})T(n - n/c) + O(n)$ where $c > 1$. Since the number of edges in a tree decomposition of width k is no more than kn, we can split the problem and update the solution in O(n) time. It is known that [22], in linear time, we can find a centroid bag which is a bag such that the removal of this bag gives us subtrees of size no more half of the size of the original tree. Instead of picking the left child and the right child of the root as roots of the new subtrees, we can pick the centroid bag as the root at each step. Hence, we get $T(n) = (2^{k+2})T(n/2) + O(n) = O(n^k)$. Although this algorithm is less efficient than dynamic programming algorithm in terms of time complexity, it is advantage is that the memory space used by the algorithm is always linear even for graphs with unbounded width.

2.3 Modified Divide and conquer

Recall that, in a smooth tree decomposition of width k, each pair of adjacent bags has exactly k vertices in common. Using this property, we can improve the performance of the divide and conquer algorithm described in Section 2.2.

Lemma 2.3.1 *If the graph of a formula has a balanced smooth tree decomposition of* *width k, then the running time of Algorithm 2.1 is $O(2^k n)$ with linear memory space.*

Proof In the smooth tree decomposition, by assigning a value to each variable in the parent bag, we reduce the number of unprocessed variables in the child bag to one. Therefore, if we apply the divide and conquer algorithm to a smooth tree decomposition, then we only need to spend 2^k time for the root bag. Once an assignment to the root bag is given, there are only 2 possible choices for each of the descendant bags. For each of the 2^k possible assignments of the root bag, we only need to spend $T(n) = 2T(n/c) + 2T(n - n/c) + O(n)$ time to determine its corresponding optimal assignment. In exactly the same way as Algorithm 2.1, we only store the current best assignment at each time, so the algorithm takes only $O(n)$ memory space. Hence, if the smooth tree decomposition is balanced, the algorithm yields a total running time of $O(2^k n)$ using only linear memory. ∎

However, we should also notice that the algorithm performs badly on unbalanced decompositions. In order to use the property that adjacent bags differ by only one vertex, we can no longer use centroid bags to reduce the height of the tree decomposition. In the worst case, when the tree decomposition is a path, the tree can have height as large as $n - k$, then the over all running time is $O(2^n)$. If we can transform a tree decomposition into a smooth and balanced tree decomposition with the same width, then the divide and conquer algorithm can achieve the same time complexity as the dynamic programming algorithm while using only linear memory. This motivates us to try to balance a tree decomposition.

Given a tree decomposition of a graph, we can always transform it into a smooth tree decomposition with the same width in linear time [9]. In the next section, we discuss a possible technique which can be used to balance a path decomposition. However, it is not always possible to transform a path decomposition into a balanced smooth tree decomposition with only a small amount increase in its width. To balance a path decomposition, we increase the size of each bag by a constant factor of 3. If two adjacent bags in the decomposition have no vertices in common, then we need to add k bags in between these two bags to make it smooth where k is the maximum size of these two bags. By doing so, we destroy the balance of the decomposition. Of course, we can rebalance the decomposition and then smooth it again. However, repeatly applying these procedures can lead to large increases in the width of the decomposition. In this case, we can not improve the performance of Algorithm 2.1. Hence, we decide not to put any more efforts into extending the technique for balancing path decomposition to the general tree decomposition.

2.4 Balancing path decomposition

As discussed in Section 2.3, the performance of the divide and conquer algorithm on a smooth tree decomposition depends on the height and the width of a decomposition. For a graph with bounded treewidth, its tree decomposition could be a path which have height as large as $n - k$. In order to improve the performance in the worst case, we could want to balance the path decomposition so that its height is minimal while its width stays within a constant factor of the original decomposition.

Path decomposition which is a special case of the tree decomposition usually have large height. In this section, we present an algorithm that balances a given path decomposition into a balanced tree decomposition while only increases the width by a constant factor of 3.

Let $(X = \{X_i | i \in I\}, P = (I, H))$ denote the path decomposition and $I = 1, 2, ..., h$. Without loss of generality, let's assume that $i - 1$ and $i + 1$ are the left and right neighbors of i in P respectively. Our algorithm can be described by the following two steps.

- **Construct a balanced tree.** Let $T = (I, H')$ be a tree such that $\lfloor \frac{h}{2} \rfloor$ is the root of the tree. For any node i in the tree, $\lfloor \frac{i}{2} \rfloor$ and $i + \lfloor \frac{i}{2} \rfloor$ are the left and right child of i in T respectively.
- **Satisfy the property P2.** For all i and $i + 1$ in P, if i and $i + 1$ are not adjacent in T, add $X_i \cap X_{i+1}$ to X_j where j is on the path from i to $i + 1$ in

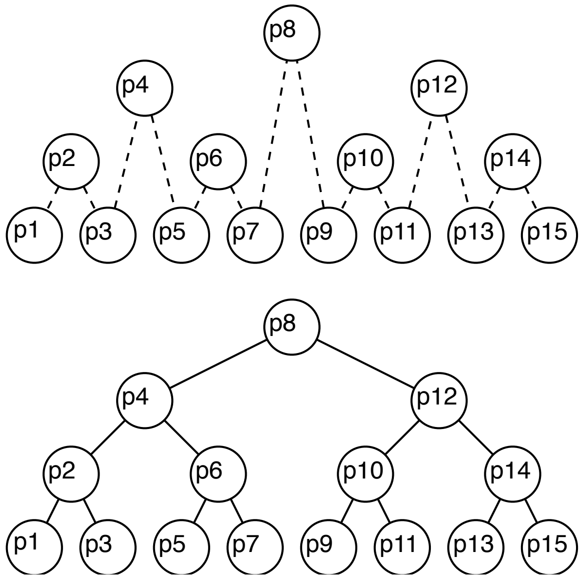

Figure 2.1: Transform a path to a tree

T. Pair $(X = \{X_i | i \in I\}, T = (I, H'))$ is a balanced tree decomposition of $(X = \{X_i | i \in I\}, P = (I, H))$.

From the first step, we get a partial tree decomposition (X, T) of height $\log(h)$. However, this partial tree decomposition may violates property P2 of the tree decomposition as defined in Section 1.2. In the second step, we modify the bags of that partial tree decomposition to satisfy all the constraints. Since each step takes $O(n)$ time, we can transform a path decomposition into a balanced tree decomposition in linear time.

Observation 2.4.1 *As shown in Figure 2.1 we can retrieve the ordering of bags in P from T by inorder traverse. Hence, for an arbitrary internal bag i in T, $i-1$ and $i+1$ are located at the rightmost leaf in the left subtree rooted at i and the leftmost leaf in the right subtree rooted at i respectively. Starting from i, we can reach $i-1$ by first picking a left child, then keep picking the right child. Similarly, we can reach $i+1$ by first picking a right child, then keep picking the left child.*

Lemma 2.4.2 *Given a path decomposition (X, P) of width k, we can transform it into a balanced tree decomposition (X, T) of width not exceeding $3k$.*

Proof To proof the lemma, we first show that each modification increases the size of a bag by at most k, then we show that each bag can be modified at most twice.

- $|X_i \cap X_{i+1}| \leq k$

  Since the width of (X, P) is k, we know both $|X_i|$ and $|X_{i+1}|$ are no more than $k+1$. Since X_i and X_{i+1} are different bags, so their intersection is at most k.

- X_j **are modified at most twice**

  Let l be the parent of j and let i be the parent of l in T. From the above observation we know that j appears on either the paths from l to $l+1$ and i to $i-1$ or the paths from l to $l-1$ and i to $i+1$. In the first case, l is the left child of i and j is the right child of l. In the second case, l is the right child of i and j is the left child of l. As mentioned earlier, for any bag r, the paths from r to $r-1$ and r to $r+1$ always pick the children on the same side except the first one. Therefore, j can not appear on any other r to $r-1$ or r to $r+1$ paths.

Since for an arbitrary j, X_j can be modified at most twice in our algorithm and each modification increases the size of X_j by at most k, (X, T) is a balanced tree decomposition of width no more than $3k$. ∎

Chapter 3

Separators

3.1 Separator Width

As shown in previous sections, in order to use the divide and conquer algorithm, we want to separate the original problem into approximately equal sized independent subproblems. Although all bags in a tree decomposition of the graph can be used as separators, they are usually not optimal separators due to the repetition of vertices. In fact, some of the properties of tree decomposition is not working in the advantage of divide and conquer algorithm. For example, divide and conquer algorithm requires the subproblems are independent of each other, but requiring that all edges covered by some bags in the tree decomposition has nothing to do with the independence of the subproblems. On the contrary, it can increase the size of the bags, hence increase the complexity of the divide and conquer algorithm. In this section, we are going to define a new notion of width, separator width, which is based on the size of separators in the optimal balanced separator tree. We show that separator width of a graph doesn't exceed its tree width. Most importantly, unlike other width measure, separator width of a graph can be logarithmically smaller than its tree width. Therefore, the divide and conquer algorithm is more efficient when applying on an optimal separator tree instead of a tree decomposition.

Definition

- **Balanced separator.** A set $S \subset V$ of a graph $G(V, E)$ is a balanced separator of G if and only if the removal of S partitions G into connected components of size no more αn where $n = |V|$ and $\alpha \leq \frac{1}{2}$. However, often a relaxed notion of balanced separator is used. For example, in [28] a balanced separator is a separator with $\alpha \leq \frac{3}{4}$. We use the relaxed notion. For any graph $G(V, E)$, a balanced separator of G always exits. If G is a clique, then the balanced separator of G is V.

- **Balanced separator tree.** A balanced separator tree of graph $G(V, E)$ is a pair $(X = \{X_i | i \in I\}, T = (I, F))$ with X a family of subset of V, one for each node of T, and T a tree such that

  - $\bigcup_{i \in I} X_i = V$.
  - for all i, $j \in I$: $X_i \cap X_j = \emptyset$
  - for all $i \in I$: X_i is a a balanced separator of the induced subgraph of G on the set $\bigcup_{k \in T_i} X_k$ where T_i is the subtree of T rooted at i.

- **Width of a balanced separator tree.** The width of a balanced separator tree (X, T) is $sw((X, T)) = max_{i \in I} |X_i|$.

- **Separator width.** The separator width of a graph G, $sw(G)$ is the minimum width over all possible balanced separator trees of G.

Given a graph G, by recursively determining the balanced separator of each component, we can construct a balanced separator tree. As shown in Figure 3.1, each internal node r is a balanced separator of the graph which is induced by the nodes in the subtree rooted at r. However, such a tree of balanced separators is not unique. As shown in Figure 3.2, for the given graph, we have two different trees of balanced separators. In later sections, we would discuss algorithms for finding a tree of balanced separators. Depending on the algorithm, we may get trees of different width.

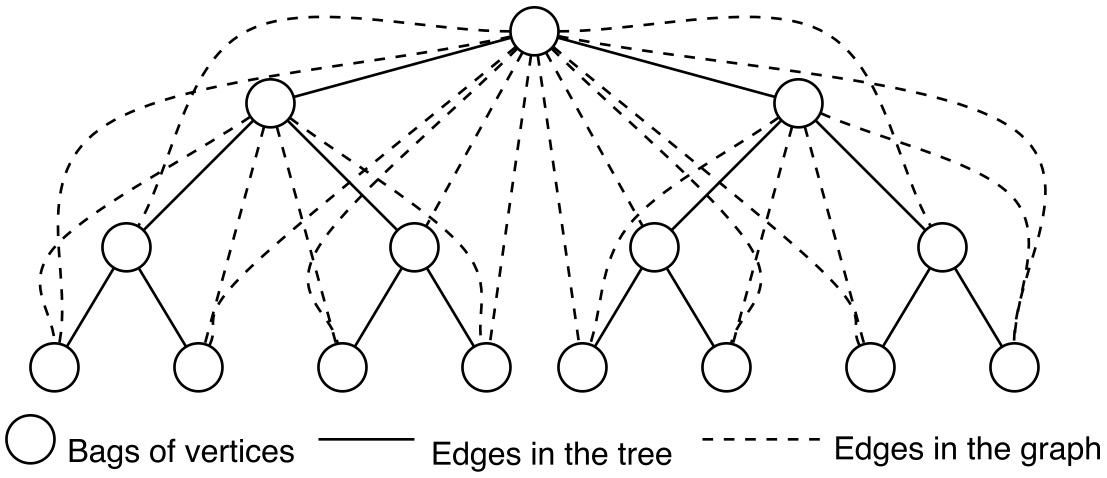

Figure 3.1: A balanced separator tree

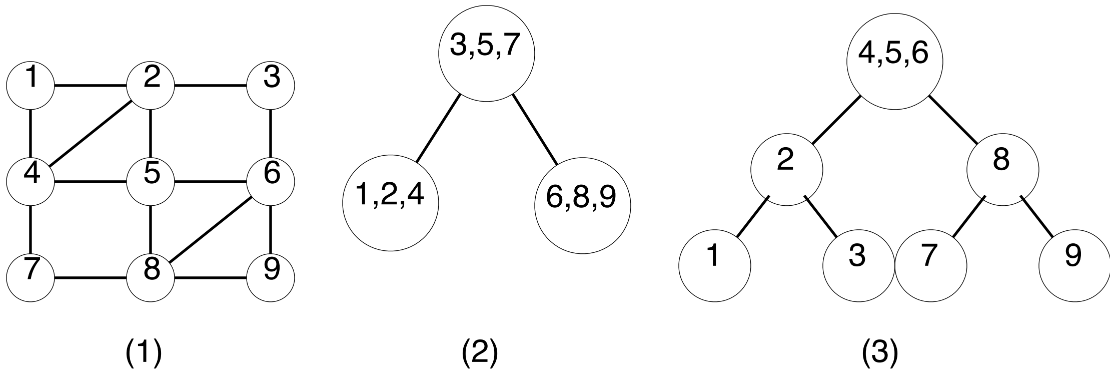

Figure 3.2: (1) A graph. (2), (3) Two different balanced separator trees.

3.2 Relationship of separator width and tree width

From the definition of separator width, we know it is very similar to tree width. However, the notion of a tree decomposition is very different from a balanced separator tree. First of all, tree decomposition requires all edges to be covered by some bags in the decomposition, but a balanced separator tree only covers all vertices of the graph. Secondly, from the definition, we know that in a balanced separator tree, each vertex in the graph belongs to a unique bag. Last, but most importantly, the height of balanced separator tree is at more $\log(n)$, yet the height of a tree decomposition can be as large as n.

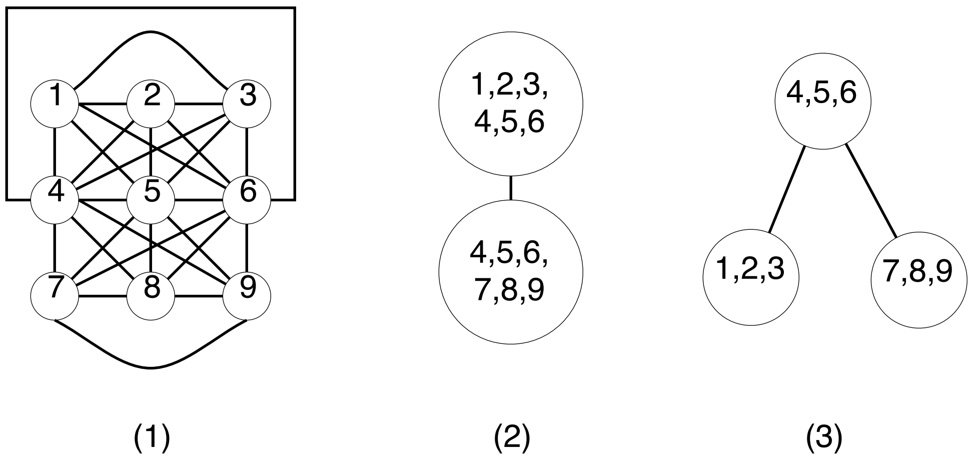

Figure 3.3: (1) A graph. (2) Tree decomposition of the graph. (3) Balanced separator tree.

Theorem 3.2.1 *For an arbitrary graph G with n vertices, $sw(G) \leq tw(G)$, and $tw(G)$ can be as large as* $\log(n)sw(g)$

We prove the theorem using the following two lemmas,

Lemma 3.2.2 *For any graph G of tree width k, there is a balanced separator tree of the same width.*

Proof It is known [10] that in any graph G of tree width k, there is a balanced separator of size at most k. Since any subgraph of G also has tree width at most k, we can obtain a balanced separator tree by recursively picking a size k balanced separator of each component. By definition, the width of this balanced separator tree is k. ▮

From Lemma 3.2.2, we know that $sw(G) \leq tw(G)$.

Lemma 3.2.3 *There is a graph G contains a clique of size $\log(n)sw(G)$ where n is the number of vertices in G.*

We prove the lemma by constructing a separator tree of width $sw(G)$ such that G contains a clique of size $\log(n)sw(G)$.

Proof As shown in Figure 3.1, Let $T = (I, H)$ be a full binary tree. We include $sw(G)$ vertices in each bag X_i, $i \in I$. The vertex set V of G is $\bigcup_{i \in I} X_i$. To construct the edge set E of G, we first add an edge (v, u) for all pairs of vertices $v, u \in X_i$ and all X_i, $i \in I$. The vertices in each bag form a clique in G. Then, for all pairs of bags X_i and X_j such that j is an ancestor of i in T, we add an edge (v, u) between all vertices $v \in X_i$ and $u \in X_j$. For each root to leaf path p in T, $K = \bigcup_{i \in p} X_i$ is a clique in G. The pair $(X = \{X_i | i \in I\}, T = (I, F))$ is a balanced separator tree of $G = (V, E)$ which have width $sw(G)$, and G contains a clique K of size $\log(n)sw(G)$. ▮

From Lemma 3.2.3, we know the $tw(G)$ can be as large as $\log(n)sw(G)$ since the tree width of a graph is at least the size of its largest clique. Hence, the theorem is proved.

However, we should also notice that the size of cliques in a graph G is at most $\log(n)sw(G)$ where n is the number of vertices in G.

Lemma 3.2.4 *For any graph G, the largest clique in G is at most $\log(n)sw(G)$ where n is the number of vertices in G.*

Proof Let p be the longest root to leaf path in a balanced separator tree. As shown in Lemma 3.2.3, $\bigcup_{i \in p} X_i$ is a clique in G, and the size of this clique is at most $\log(n)sw(G)$ since p contains at most $\log(n)$ bags and each bag is of size at most $sw(G)$. Moreover, for any $v \in X_i$ such that X_i is not a bag on p, v cannot connect to all vertex u in all bags X_j on p. Therefore, the largest clique in G is at most $\log(n)sw(G)$. ▮

3.3 Hardness of determining an optimal balanced separator tree

An *optimal balanced separator tree* is a separator tree with minimum width. As discussed in Section 1.3.2, the problem of finding an exact minimum separator of a graph is NP-complete. To find a balanced separator tree, we need to find a minimum balanced separator at each stage. Therefore, we cannot determine the optimal balanced separator tree in polynomial time.

As mentioned in Section 1.3.2, there are good approximation algorithms for finding fixed size k balanced separators. Using the approximation algorithm described in that section, we can recursively construct a balanced separator tree by greedily picking the size k balanced separator returned by the approximation algorithm. However, this greedy algorithm does not guarantee to determine a balanced separator tree of width k.

It is possible to recursively construct a graph G such that the separator decomposition determined by the greedy algorithm can have width equal to the size of the largest clique in G while $sw(G)$ is much smaller. Let $H(i)$ be a graph that contains two parts, a size 2 clique K_2 and a size i path P_i. Vertices $u_i, v_i \in K_2$ are connected to the end points $w_i, z_i \in P_i$ respectively. Starting with a graph $G_0 = H(2)$, we can obtain G_i from G_{i-1} by combining G_{i-1} with $H(|G_{i-1}|)$. A graph G_{i-1} is combined with $H(|G_{i-1}|)$ by connecting all vertices in K_2 of $H(|G_{i-1}|)$ with all vertices in the clique of G_{i-1}. From the construction, we know G_{i-1} have a unique clique which is the union of vertices in K_2 of each $H(j)$.

For any graph G_i, If we greedily pick $\{u_j, v_j\}$ in K_2 of each $H(j)$ as the balanced separator at each step, then eventually we get a balanced separator tree of width 2 since every path has a balanced tree decomposition of width 1. However, if we greedily pick the end points $\{w_j, z_j\}$ in P_j of each $H(j)$ as the balanced separator at each step, then eventually we end up with a clique of size $2i + 2$. In this case, the balanced separator tree has width $2i + 2$.

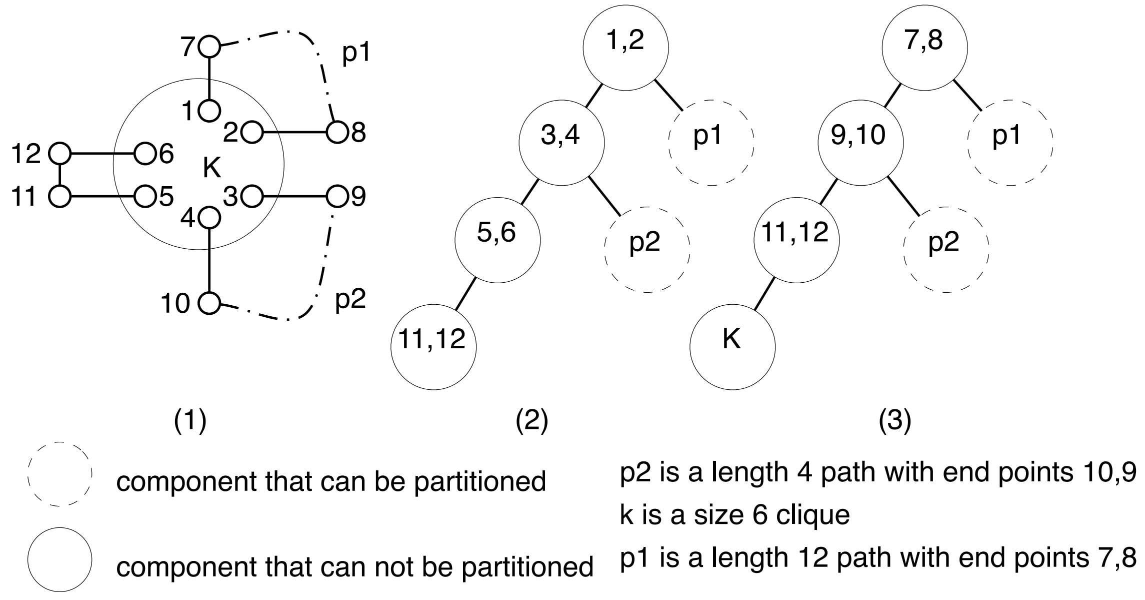
7
p1
1
2
8
12
6
K
11
5
3
9
4
10
p2
1,2
3,4
p1
5,6
p2
11,12
7,8
9,10
p1
11,12
p2
K
(1)
(2)
(3)
component that can be partitioned
component that can not be partitioned
p2 is a length 4 path with end points 10,9
k is a size 6 clique
p1 is a length 12 path with end points 7,8

Figure 3.4: (1) A graph. (2) An optimal balanced separator tree. (3)A balanced separator tree.

In Figure 3.4, we take G_2 as an example. When $k = 2$, if we greedily pick $\{7, 8\}$, $\{9, 10\}$, and $\{11, 12\}$ as the balanced separator at each step respectively, than eventually we end up with a component, K, which is a clique so that the separator width of this balanced separator tree is the size of the clique. In this case it is 6. On the other hand, if we pick $\{1, 2\}$, $\{3, 4\}$, and $\{5, 6\}$ as the balanced separator at each step respectively, then the balanced separator tree has separator width 2 which is optimal. In general, the width of the separator tree determined by the greedy algorithm can be as large as $\log(n)sw(G)$.

Approximation Algorithm

Although the greedy algorithm does not guarantee to return an optimal solution, we can modify the separator tree determined by the greedy algorithm to obtain a good approximation. For a given graph G, and a constant k, let (X, T) be the separator tree determined by the greedy algorithm. The approximation algorithm takes T as an input, and reshapes it in the following two steps.

- **Extension.** Since the separator tree returned by the greedy algorithm may contain short paths, we need to extend T into a full binary tree of height $h = O(\log(n))$ by adding dummy nodes to short paths. For each dummy node i, the corresponding set X_i is initialized to be empty. Furthermore, we push the leaf nodes in the original tree, all the way down to one of the leaf nodes in the extended tree.

- **Propagation.** In the propagation step, we first traverse the tree from bottom-up starting at the leaf nodes. For each leaf node i, we count how many vertices in X_i need to be propagated to its parent if we only allow to keep $k + j$ vertices in the current node. Let's denote this number by c_i. For each internal node i with children a and b, $c_i = c_a + c_b - j$. We look for the smallest j such that $c_r = 0$. Once we reach the root r, if $c_r \neq 0$, then we need to increase j, and repeat this process. When $c_r = 0$, we know that we have obtain the right value of j, so we revisit the tree from bottom-up, at each node i we propagate c_i vertices to its parent.

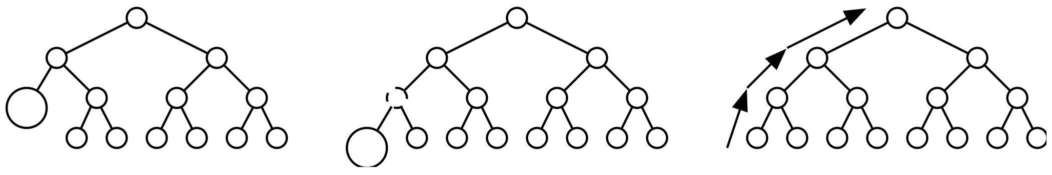

Figure 3.5: Approximation algorithm

Observation 3.3.1 *Let v be an arbitrary vertex in any bag X_i, propagating v to any bag X_j such that j is an ancestor of i in T, then the resulting tree is still a separator tree.*

From Observation 3.3.1, we know that the tree we obtained after the bottom-up propagation step is a separator tree.

Lemma 3.3.2 *Let K be a clique in G, for all i such that X_i contains some vertices in K, they all must belong to the same leaf to root path in T.*

Proof Without loss of generality, assume X_i and X_j are two bags that contain some vertices in K, and i, j belongs to different leaf to root paths in T. There is a $l \in I$ such that l is the common ancestor of i, j. By definition we know X_l separates X_i and X_j. Hence, X_l must contain all vertices in K which contradicts the requirement that X_l, X_i and X_j are pairwise disjoint. ▮

Recall that in Lemma 3.2.4, we showed cliques in G can be as large as $\log(n)sw(G)$. We also know that the balanced separator decomposition returned by the greedy algorithm can contain such cliques in its leaf nodes. In order to reduce the size of the clique in a leaf node, we need to spread it along a leaf to root path which increases the size of each bag along the path by $sw(G)$. Since each internal bag already contains $k \geq sw(G)$ vertices, we know for some graphs the separator tree returned by the approximation algorithm could have width greater than or equal to $2k$. However, we are not going to put any effort in improving the algorithm to obtain a balanced separator tree of width at most $2k$ in these cases. In next subsection, we show that improving the algorithm to obtain a balanced separator tree of width $2k$ doesn't improve the time complexity for solving our problem.

3.4 Applications

Given a balanced separator tree (X, T) of a graph G, we can apply the divide and conquer algorithm to solve MAX2SAT problem. Instead of taking a tree decomposition as input, we take a balanced separator tree as input. The algorithm proceeds in the same way as described in Section 2.2. In general, the running time $T(n)$ of the algorithm becomes $T(n) = 2^{sw(G)+1}T(n/2) + O(n) = O(n^{sw(G)})$.

We should also notice that in the original balanced separator tree returned by the greedy algorithm described in previous section, each internal bag has size k for some k, while each leaf bag has size at most $\log(n)k$. When we apply Algorithm 2.1 on this tree, its time complexity is $O(n^{2k})$. First of all, since the size of a leaf bag is no more than $\log(n)k$, each of the $\frac{n}{2}$ leaf bags can be processed independently in $O(n^k)$ time. Secondly, the running time on the internal bags of the balanced separator tree is at most $T(n) = 2^k T(n/2) + O(n) = O(n^k)$. Therefore, the total running time of the algorithm is $O(n^{2k})$. Hence, we know that the approximation algorithm described in previous section is not useful unless it can reduce the width of a balanced separator tree returned by the greedy algorithm to a value less than $2k$.

The performance of Algorithm 2.1 can be improved when the input is a balanced separator tree instead of the tree decomposition. Moreover, the balanced separator decomposition is specially designed for divided and conquer algorithm, so dynamic programming algorithm doesn't work on this decomposition. The properties of the tree decomposition ensure that every edge in the graph is covered in a bag. Therefore, we can correctly calculate the number of unsatisfied clauses by summing up unsatisfied clauses in each bag. However, in a balanced separator decomposition there is no such restriction. In fact, many edges are not covered by any bag at all. Therefore, if we apply dynamic programming algorithm on a balanced separator decomposition, many unsatisfied clauses are missed by the algorithm.

Next, we compare the performance of algorithm 2.1 on a balanced separator decomposition with the performance of dynamic programming algorithm on a smooth tree decomposition. We consider the case when $tw(G)$ is close to $sw(G)$ and the case when $tw(G)$ is logarithmically larger than $sw(G)$.

Dynamic programming is the most standard approach that is used in designing algorithms for solving problem instances of bounded tree width. This implies that it is hard to design an algorithm that beats the time complexity of dynamic programming algorithm. Indeed, as shown in Table 3.1, when the separator width is less than the tree width by only a small constant c, the performance of Algorithm 2.1 on a balanced separator tree is still worse than the performance of dynamic programming algorithm.

Table 3.1: Time and space complexity

	Divide and conquer	Dynamic Programming
Time complexity	$O(n^{sw(G)}) = O(2^{\log(n)k})$	$O(2^{tw(G)+3}n) = O(2^{ck+3+\log(n)})$
Space complexity	$O(n)$	$O(2^{tw(G)}) = O((2)^{ck})$

But still, we are happy to notice that there are also cases in which the performance of Algorithm 2.1 is superior than dynamic programming algorithm. For a graph G with $sw(G) = k$ and $tw(G) = \log(n)k$, we compare the time and space complexity of the divide and conquer algorithm on its balanced separator decomposition with the dynamic programming algorithm on its smooth tree decomposition. As shown in Table 3.2, the running time of both algorithm is comparable in this case, but the divide and conquer algorithm uses much less memory space.

Table 3.2: Time and space complexity

	Divide and conquer	Dynamic Programming
Time complexity	$O(n^{sw(G)}) = O(n^k)$	$O(2^{tw(G)+3}n) = O(n^{k+4})$
Space complexity	$O(n)$	$O(2^{tw(G)}) = O(n^k)$

Moreover, when $tw(G) \geq \log(n)$, dynamic programming algorithm uses much more memory space then divided and conquer algorithm. By scarifying the performance of the algorithm, we are able to save the memory space.

Chapter 4

Approximation Algorithm

The algorithms described in Section 2 are exact algorithms for solving MAX2SAT problem. As we know MAX2SAT is NP-hard, there are also many interesting studies focusing on designing efficient approximation algorithms. In this section, we use the elimination ordering of vertices as a tool to design an approximation algorithm for d-degenerate instances of MAX2SAT problem. We analyze the performance of our approximation algorithm both experimentally and mathematically.

4.1 Algorithm

A graph G is *d-degenerate* if every subgraph of G has a vertex with degree at most d. Alternatively, vertices of G can be arranged in an ordering $\pi = \{v_1, v_2, v_3, ...v_n\}$ such that for every vertex v_i, v_i has at most d neighbors v_j with $j > i$. For a d-degenerate graph, the ordering π can be determined in linear time by greedily picking vertices of degree no more than d. For the sake of connivence, we call π an *elimination ordering* of G, and d the *elimination width*. It was shown by Arnborg [3] that all graphs with bounded tree width k have an elimination ordering π of width k. Moreover, the class of tree width k graphs is not the only class of graphs with an elimination width k. For example, planar graphs do not have a fixed tree width, but they are 5-degenerate by Euler's Theorem, so they have a elimination width 5.

To describe the approximation algorithm, we need some terminology. Let ϕ_i denote a *partial assignment*, where variables $\{v_1, v_2, ..., v_{i-1}\}$ are assigned to some truth values, and variables $\{v_i, v_{i+1}, ..., v_n\}$ are not assigned to any value. Let F_i denote the formula corresponding to ϕ_i. It is obtained from F by removing all clauses containing a variable in $\{v_1, v_2, ..., v_{i-1}\}$ and satisfied by its value, and substituting a unit clause v_j ($\overline{v_j}$) for unsatisfied clauses containing v_j ($\overline{v_j}$) such that $v_j \in \{v_i, v_{i+1}, ..., v_n\}$. Formula F_i contains two types of clauses, namely, the unprocessed clauses and the processed clauses. An unprocessed clause in F_i is a clause containing two variables such that both variables belong to $\{v_i, v_{i+1}, ..., v_n\}$. A processed clause in F_i is a unit clause containing a variable in $\{v_i, v_{i+1}, ..., v_n\}$. By giving a truth value to v_i, we can extend ϕ_i to a partial assignment ϕ_{i+1}. As shown in Figure 4.1, let A_i, B_i, D_i, and E_i denote the set of unprocessed clauses that are satisfied by the value assigned to v_i, the set of processed clauses that are satisfied by the value assigned to v_i, the set of unprocessed clauses that are unsatisfied by the value assigned to v_i, and the set of processed clauses that are unsatisfied by the value assigned to v_i respectively. When we eventually extend a partial assignment to a total assignment ϕ, the number of unsatisfied clauses is $\sum_{i=1}^{n} |E_i|$. Since $\sum_{i=1}^{n} |E_i| = \sum_{i=1}^{n} |D_i| - \sum_{i=1}^{n} |B_i| = \sum_{i=1}^{n}(|D_i| - |B_i|)$, in order to maximize the number of satisfied clauses we want to minimize the total difference of $|D_i|$ and $|B_i|$.

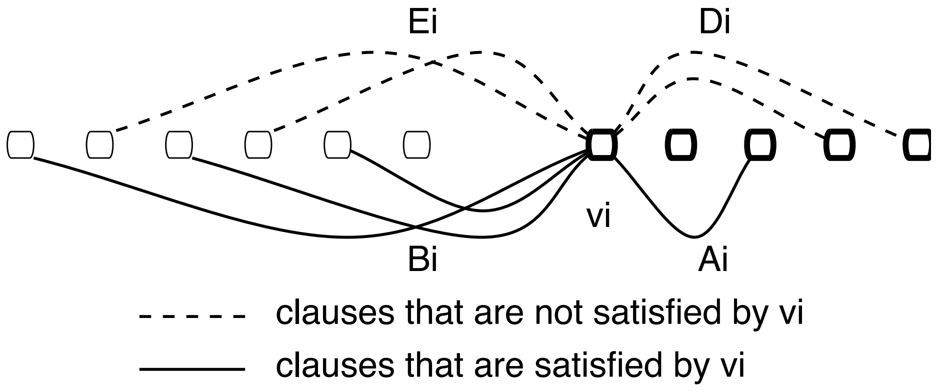
Ei
Di
vi
Bi
Ai
clauses that are not satisfied by vi
clauses that are satisfied by vi

Figure 4.1: Four sets of clauses

Based on the above observation, we can define the following rule,

Rule 4.1.1 (Flipping condition) *For a given value of v_i, if $|A_i| - |E_i| < |D_i| - |B_i|$, then flip the value of v_i*

Using the flipping condition, we design our approximation algorithm in the following way. Let's consider the vertex in the given elimination ordering from the lowest to the highest. At each step, take the vertex v_i, and set it to be true. After determining the sets A_i, B_i, D_i, E_i, if $|A_i| - |E_i| < |D_i| - |B_i|$, then flip the value of v_i.

Algorithm 4.1 GreedySearch(F, $\pi = \{v_1, v_2, v_3...v_n\}$)

```
Require: An elimination ordering $\pi = \{v_1, v_2, v_3...v_n\}$ of variables in $F$
Ensure: an assignment $\phi$ of $F$
 1: for currentV = $v_1$ to $v_n$ do
 2:     $\phi(currentV) = 1$;
 3:     A:= the set of unprocessed clauses satisfied by $\phi(currentV)$;
 4:     B:= the set of processed clauses satisfied by $\phi(currentV)$;
 5:     D:= the set of unprocessed clauses unsatisfied by $\phi(currentV)$;
 6:     E:= the set of processed clauses unsatisfied by $\phi(currentV)$;
 7:     if $|A| - |E| < |D| - |B|$ then
 8:         $\phi(currentV) = 0$;
 9:     end if
10: end for
11: return $\phi$;
```

This algorithm is very similar to local search. But instead of setting each vertex to the value that minimizes the number of unsatisfied clauses, we minimize the difference between the number of unprocessed clauses that are unsatisfied and the number of processed clauses that are satisfied. Most importantly, the vertices are not processed arbitrarily. The elimination ordering ensures that after a vertex is considered, at more d processed clauses are introduced.

The advantage of this algorithm is that it is time efficient. The algorithm processes each variable only once, and for each variable it takes c time to compute the four related numbers where c is the number of clauses incident to the vertex. In total, the running time of this algorithm is O(δn) where δ is the maximum degree over all vertices.

The disadvantage of this algorithm is that it can be fooled by some specially constructed instances. For example, let $H_0 = \{v_1 \vee \overline{v_2}\} \wedge \{v_1 \vee v_3\} \wedge \{\overline{v_1} \vee v_4\} \wedge \{v_2 \vee \overline{v_3}\} \wedge \{\overline{v_2} \vee \overline{v_4}\} \wedge \{v_3 \vee \overline{v_4}\} \wedge \{v_4 \vee v_5\}$. Formula H_0 has an optimal assignment $\phi_0 = (1, 1, 1, 1, 1)$ satisfies 6 out of 7 clauses. Now we can recursively construct a formula H_i from H_{i-1} by adding a variable v_{i+5} and clauses $\{\overline{v_i} \vee v_{i+5}\}$, $\{\overline{v_{i+1}} \vee v_{i+5}\}$, $\{\overline{v_{i+2}} \vee v_{i+5}\}$, $\{v_{i+3} \vee \overline{v_{i+5}}\}$, and $\{v_{i+4} \vee v_{i+5}\}$. Each formula H_i contains $i + 5$ variables and $7 + 5i$ clauses. $\phi_i = (1, 1, 1, ..., 1)$ is an optimal assignment of H_i which satisfies $6 + 5i$ clauses. The elimination ordering $\pi_i = \{v_1, v_2, ..., v_{i+4}\}$ has bounded width 5. For $i \geq 5$, the best solution returned by GreedySearch algorithm is $(0, 0, 0, ...0, 1, 1, 1, 1)$ which satisfies $9 + 4i$ out of $7 + 5i$ clauses.

Proposition 4.1.2 *For $i \geq 5$, we can construct a formula H_i with $i+5$ variables and $7 + 5i$ clauses, such that the optimal solution of the formula satisfies $6 + 5i$ clauses while the best solution of GreedySearch algorithm satisfies $9 + 4i$ clauses.*

In general, there is no known method to determine exactly how many clauses in a formula are satisfied by an optimal assignment. So it is difficult or even impossible to determine the exact approximation ratio of the GreedySearch algorithm. To analyze the performance of the algorithm, we first experimentally compare its performance with two well-known approximation algorithms. we also use a mathematical model to determine its expected performance on random formulas.

4.2 Experimental analysis

Even though it is possible to construct instances for which GreedySearch (GS) algorithm approximates badly, we still can expect a good performance of the algorithm in a typical case. In this section, we conduct an experiment that compares the performance of GS with one-pass local search (OLS) and GSAT on random instances. We choose these two algorithm because OLS have the same time complexity as GS, and GSAT is one of the best approximation algorithms for solving MAX2SAT. In our experiment, we allow GSAT to restart 10 times and flip each assignment $100n$ times.

The random instance generator in this experiment takes three parameters, n, m, and d. Let n denotes the number of variables, m denotes the number of clauses, and d denotes the elimination width. The output is a formula over n variables with m clauses, such that the vertex ordering $\{1, 2, 3, 4, ...n\}$ has elimination width at most d. This random instance generator is based on the uniform random model described in Section 1.3.1. Each clause C_k with variable v_i and v_j, $j > i$ is considered with equal probability. For each variable v_i, we count the number of clauses C_k that is already included in the formula, if it exceeds the threshold d, C_k is discarded. Notice, in order to produce a formula of m clauses on n variables, we need to have the elimination width $d \geq \frac{m}{n}$. In order to ensure the generator terminates in a reasonable amount of time, we usually set d to be a value that is slightly larger than the density $\frac{m}{n}$.

In the following table, we compare the performance of these three algorithms on instances that generated by different parameters. The performance of an algorithm is measured by the number of unsatisfied clauses in the best assignment determined by the algorithm. The experimental results in the table are averages over 20 randomly generated formulas.

Table 4.1: Performance of the three approximation algorithms

Input parameter			Number of unsatisfied clauses		
n	m	d	GS	OLS	GSAT
100	800	10	92	151	90
100	2500	30	374	560	436
100	4000	60	547	894	755
500	3000	10	311	530	296
500	12500	30	1304	2194	1647
500	100000	300	13974	24558	22264
1000	6000	10	599	1079	598
1000	100000	150	13844	24131	21102

From this table, we observe that GS performs better than OLS on all instance. Its performance is comparable to GSAT on low density instances. For very sparse instances, GSAT is slightly better than GS. However, in high density cases, GS produces better approximations than GSAT. We should also notice that the time complexity of GSAT is much larger than GS. It takes much longer time for GSAT to find a best solution.

4.3 Mathematical model

In this section, we construct a model of GS using a system of differential equations based on a similar analysis in [13], and show that the performance of our model does agree with the expected performance of the algorithm. Before we start to model GS, we need to introduce a theorem given by Wormald [35] which provides the mathematical tool for analyzing GS.

As defined in [35], let's denote discrete time random processes by $(Q_0, Q_1, ...)$, where Q_i takes values in some set S. For $n = 1, 2, ...$, a sequence Q_n of random processes contains elements $(q_0(n), q_1(n), ...)$ where each $q_i \in S$. The little oh and big Oh notation denote asymptotic for $n \to \infty$, but uniform over all other variables. If $max\{x|P(X = x) \neq 0\} = o(f(n))$, then $X = o(f(n))$ *always*. An event occurs almost surely if its probability in Q_n is $1 - o(1)$. A function $f(u_1, ..., u_j)$ satisfies a Lipschitz condition on $D \subset \mathbb{R}^j$ if a constant $L > 0$ exists with the property that

$$|f(u_1, ..., u_j) - f(v_1, ..., v_j)| \leq L \sum_{i=1}^{j} |u_j - v_j|$$

for all $(u_1, ..., u_j)$ and $(v_1, ..., v_j)$ in D

Theorem 4.3.1 (Wormald) *Let $Y_i(t)$ be a sequence of real-valued random variables, $1 \leq i \leq k$ for some fixed k, such that for all i, all t and all n, $|Y_i(t)| \leq Cn$ for some constant C. Let $H(t)$ be the history of the sequence. Let $I = \{(y_1, ...y_k) : Pr[Y(0) = (y_1n, ...y_kn)\} \neq 0$ for some $n\}$. Let D be some bounded connected open set containing the intersection of $\{(s, y_1, ...y_k) : s \geq 0\}$ with a neighborhood of $\{(0, y_1, ..., y_k) :$*

$(y_1, ...y_k) \in I\}$. Let $f_i : \mathbb{R}^{k+1} \to \mathbb{R}$, $1 \le i \le k$, and suppose that for some function $m = m(n)$,

(i) *for all i and uniformly over all $t < m$,*

$$E(Y_i(t+1) - Y_i(t)|H(t)) = f_i\left(\frac{t}{n}, \frac{Y_0(t)}{n}, ..., \frac{Y_k(t)}{n}\right) + o(1), always;$$

(ii) *for all i and uniformly over all $t < m$,*

$$Pr[|Y_i(t+1) - Y_i(t)| > n^{\frac{1}{5}}|H(t)] = o(n^{-3}), always;$$

(iii) *for each i, the function f_i is continuous and satidfies a Lipschitz condition on D.*

Then

(a) *for $(0, \hat{z}(0), ...\hat{z}(k)) \in D$ the system of differential equations*

$$\frac{dz_i}{ds} = f_i(s, z_0, ...z_k), 1 \le i \le k$$

has a unique solution in D fo $z_i : \mathbb{R} \to \mathbb{R}$ passing through $z_i(0) = \hat{z}(i)$, $1 \le i \le k$, and wchich extends to points arbitrarily close to the boundary of D;

(b) *almost surely*

$$Y_i(t) = z_i\left(\frac{t}{n}\right)n + o(n),$$

uniformly for $0 \le t \le min\{\sigma n, m\}$ and for each i, where $z_i(s)$ is the solution in (a) with $\hat{z}^{(i)} = \frac{Y_i(0)}{n}$, and $\sigma = \sigma(n)$ is the supremum of those s to which the solution can be extended.

With the theoretical foundation provided by Theorem 4.3.1., we can now start the analysis of our algorithm. Given a random formula with n variables, the algorithm produces values $q_0(n), q_1(n), ...$ where 0, 1, ... are steps of the algorithm, and we are going to define the values accurately. At each step of the algorithm, we can partition all clauses into five sets, and the performance of the algorithm can be analyzed by monitoring the dynamics of these five sets. Let F_t denote the formula obtained at time step t. Variables $\{v_1, v_2....v_{t-1}\}$ are processed variables at the current step. Likewise, variables $\{v_t, v_{t+1}....v_n\}$ are unprocessed variables at the current step. At time step t, we define the five sets as follows,

- $E_\emptyset$ is the set of all clauses in F_t that contain only unprocessed variables.
- E_1 is the set of all clauses that contain at least one processed variable such that its value satisfies the clause.
- E_0 is the set of all clauses that contain two processed variables such that none of their values satisfies the clause.
- E_+ is the set of all clauses in F_t that contain a processed variable and a positive unprocessed variable.
- E_- is the set of all clauses in F_t that contain a processed variable and a negative unprocessed variable.

Let $e_\emptyset$, e_1, e_0, e_+, and e_- denote the size of the five sets respectively, we use $\mathbf{e}$ to denote the vector ($e_\emptyset$, e_1, e_0, e_+, e_-). Also, let m_t denote the number of clauses that contain two unprocessed variables such that one of them is v_t, and denote this set of clauses by S. The relationship between these five sets can be represented by Figure 4.2. Initially, $E_\emptyset$ contains all clauses, but E_1, E_0, E_+, and E_- are all empty. As the algorithm processes, a clause in $E_\emptyset$ is either satisfied and goes to E_1, or transformed into unit clauses that contains only positive variables or negative variables and so goes to E_+ and E_- respectively. At the same time, a clause in E_+ or E_- is either satisfied and goes to E_1 or unsatisfied and goes to E_0. Also notice, clauses in E_1 and E_0 never leave these sets. Having understood the relationships between these five set, we can model the flow by the probability for a clause to move from one set to another. Let $E_\star$ and $E_\#$ denote any two of these sets. We denote the probability for a clause to move from $E_\star$ to $E_\#$ by $P(E_\star \rightarrow E_\#)$.

$$\begin{tikzcd}
& E_- \arrow[r] \arrow[dddr] & E_0 \\
E_\emptyset \arrow[ur] \arrow[dr] \arrow[ddrr] & & \\
& E_+ \arrow[uur] \arrow[dr] & \\
& & E_1
\end{tikzcd}$$

Figure 4.2: Flow Diagram

Before we compute $P(E_\star \to E_\#)$, we need to know what is the probability that a variable v_t is flipped. Let m_{10} denote the number of clauses in S such that the first literal is positive and the second one is negative. Similarly, let m_{00} denote the number of clauses in S such that both literals are negative, m_{11} denote the number of clauses in S such that both literals are positive, and m_{10} denote the number of clauses in S such that the first literal is negative and the second one is positive. Also, let l_{u0} denote the number of unit clauses that contain the negation of v_t, and l_{u1} denote the number of unit clauses that contain the positive form of v_t. We know a variable is fliped if $m_{10} + m_{11} - l_{u0} < m_{01} + m_{00} - l_{u1}$.

Let $F(m_{10}, m_{11}, l_{u0}, m_{01}, m_{00}, l_{u1})$ denote the event that we get exactly m_{10}, m_{11}, l_{u0}, m_{01}, m_{00} and l_{u1}, clauses of the corresponding type at time t, then the probability this event occurs is

$$\begin{aligned}
P(F(m_{10}, m_{11}, l_{u0}, m_{01}, m_{00}, l_{u1})) &= \binom{e_\emptyset}{m_{10}}\left(\tfrac{1}{4(n-t)}\right)^{m_{10}}\binom{e_\emptyset - m_{10}}{m_{11}}\left(\tfrac{1}{2(n-t)}\right)^{m_{11}} \\
&\quad \times \binom{e_\emptyset - m_{10} - m_{11}}{m_{01}}\left(\tfrac{1}{4(n-t)}\right)^{m_{01}} \\
&\quad \times \binom{e_\emptyset - m_{10} - m_{11} - m_{01}}{m_{00}}\left(\tfrac{1}{2(n-t)}\right)^{m_{00}} \\
&\quad \times \binom{e_+}{l_{u1}}\left(\tfrac{1}{n-t}\right)^{l_{u1}}\binom{e_-}{l_{u0}}\left(\tfrac{1}{n-t}\right)^{l_{u0}}
\end{aligned}$$

As n tends to infinity, the binomial distribution can be approximated by Poisson distribution. Thus, we have

$$\begin{aligned} P(F(m_{10}, m_{11}, l_{u0}, m_{01}, m_{00}, l_{u1})) &= (\tfrac{e_\emptyset}{4(n-t)})^{m_{10}} (\tfrac{e_\emptyset - m_{10}}{2(n-t)})^{m_{11}} (\tfrac{e_\emptyset - m_{10} - m_{11}}{4(n-t)})^{m_{01}} \\ &\times (\tfrac{e_\emptyset - m_{10} - m_{11} - m01}{2(n-t)})^{m_{00}} (\tfrac{e_+}{n-t})^{l_{u1}} (\tfrac{e_-}{n-t})^{l_{u0}} \\ &\times \frac{e^{-(\frac{e_\emptyset}{4(n-t)} + \frac{e_\emptyset - m_{10}}{2(n-t)} + \frac{e_\emptyset - m_{10} - m_{11}}{4(n-t)} + \frac{e_\emptyset - m_{10} - m_{11} - m01}{2(n-t)} + \frac{e_+}{n-t} + \frac{e_-}{n-t})}}{m_{10}! m_{11}! m_{01}! m_{00}! l_{u1}! l_{u0}!} \\ &+ O(\tfrac{1}{n}) \end{aligned}$$

The probability that v_t is flipped is

$$\begin{aligned} P(v_t\_is\_flipped) &= P(m_{10} + m_{11} - l_{u0} < m_{01} + m_{00} - l_{u1}) \\ &= \sum_{\substack{m_{10}+m_{11}-l_{u0}<m_{01}+m_{00}-l_{u1} \\ m_{10}+m_{11}+m_{10}+m_{00}<d}} P(F(m_{10}, m_{11}, l_{u0}, m_{01}, m_{00}, l_{u1})) \end{aligned}$$

To determine the probability $P(E_\emptyset \to E_+)$, we need to consider the following two cases. Firstly, v_t is negative, the other variable is positive, and v_t is not flipped. Secondly, v_t is positive, the other variable is also positive, and v_t is flipped. The probability of the first event is $\frac{1}{4(n-t)}(1 - P(v_t\_is\_flipped))$ and the probability of the second event is $\frac{1}{2(n-t)}P(v_t\_is\_flipped)$. Thus

$$P(E_\emptyset \to E_+) = \frac{1}{2(n-t)}(1 - P(v_t\_is\_flipped)) + \frac{1}{4(n-t)}e_\emptyset(t-1)$$

Similarly, the probabilities of other directions of flow are the following,

$$P(E_\emptyset \to E_-) = \frac{1}{4(n-t)}(1 - P(v_t\_is\_flipped)) + \frac{1}{2(n-t)}P(v_t\_is\_flipped)$$

$$P(E_\emptyset \to E_1) = \frac{1}{2(n-t)} + \frac{1}{4(n-t)}$$

$$P(E_+ \to E_1) = \frac{1}{(n-t)}(1 - P(v_t\_is\_flipped))$$

$$P(E_- \to E_1) = \frac{1}{(n-t)}P(v_t\_is\_flipped)$$

$$P(E_+ \to E_0) = \frac{1}{(n-t)} P(v_t\_is\_flipped)$$

$$P(E_- \to E_0) = \frac{1}{(n-t)} (1 - P(v_t\_is\_flipped))$$

Using above probabilities, we get,

$$\begin{aligned}
&\mathbf{E}(e_\star(t) - e_\star(t-1)|H_t) \\
&\quad = \sum_{E_\# \neq E_\star} (\sum_{C \in E_\#(t-1)} P(C \in E_\star(t) - \sum_{C \in E_\star(t-1)} P(C \in E_\#(t)) \\
&\qquad = \sum_{e_\# \neq e_\star} (e_\# P(E_\# \to E_\star) - e_\star P(E_\star \to E_\#)) \qquad (4.1)
\end{aligned}$$

From Equation 4.1, we get,

$$\mathbf{E}(e_\emptyset(t) - e_\emptyset(t-1)|H_t) = -m(t), \quad \mathbf{E}(m(t)) = min\{\tfrac{2e_\emptyset}{n-t}, d\}$$

$$\begin{aligned}
\mathbf{E}(e_+(t) - e_+(t-1)|H_t) = {} & \tfrac{e_\emptyset(t-1)}{2(n-t)}(1 - P(v_t\_is\_flipped)) \\
& + \tfrac{e_\emptyset(t-1)}{4(n-t)} P(v_t\_is\_flipped) - \tfrac{e_+(t-1)}{n-t}
\end{aligned}$$

$$\begin{aligned}
\mathbf{E}(e_-(t) - e_-(t-1)|H_t) = {} & \tfrac{e_\emptyset(t-1)}{4(n-t)}(1 - P(v_t\_is\_flipped)) \\
& + \tfrac{e_\emptyset(t-1)}{2(n-t)} P(v_t\_is\_flipped) - \tfrac{e_-(t-1)}{n-t}
\end{aligned}$$

$$\begin{aligned}
\mathbf{E}(e_1(t) - e_1(t-1)|H_t) = {} & \tfrac{3e_\emptyset(t-1)}{4(n-t)} + \tfrac{e_+(t-1)}{n-t}(1 - P(v_t\_is\_flipped) \\
& + \tfrac{e_-(t-1)}{n-t} P(v_t\_is\_flipped)
\end{aligned}$$

$$\begin{aligned}
\mathbf{E}(e_0(t) - e_0(t-1)|H_t) = {} & \tfrac{e_+(t-1)}{n-t} P(v_t\_is\_flipped) \\
& + \tfrac{e_-(t-1)}{n-t}(1 - P(v_t\_is\_flipped)
\end{aligned}$$

Finally, let's check that the random process $(\mathbf{e}(1), \mathbf{e}(2), \mathbf{e}(3)...)$ satisfies all three conditions of Theorem 4.2.1.

(i) To show that $\mathbf{E}(e_\star(t) - e_\star(t-1)|H_t)$ can be expressed by function

$$f_\star\left(\frac{t}{n}, \frac{e_\emptyset(t)}{n}, \frac{e_0(t)}{n}, \frac{e_1(t)}{n}, \frac{e_+(t)}{n}, \frac{e_-(t)}{n}\right) + o(1),$$

it is suffices to normalize n and express $P(v_t\_is\_flipped)$ by function $p(\frac{t}{n})$.

For any component $e_\star$ of $\mathbf{e}$, let $s = \frac{t}{n}$, $f_\star(s) = \frac{1}{n}e_\star(t)$, then we know $\frac{1}{n-t} = \frac{1}{n(1-s)}$, and

$$\begin{aligned}
&p(s, m_{10}, m_{11}, l_{u0}, m_{01}, m_{00}, l_{u1}) \\
&\quad = (\tfrac{f_{e_\emptyset}(s)}{4})^{m_{10}}(\tfrac{f_{e_\emptyset}(s)-m_{10}}{2})^{m_{11}}(\tfrac{f_{e_\emptyset}(s)-m_{10}-m_{11}}{4})^{m_{01}} \\
&\times(\tfrac{f_{e_\emptyset}(s)-m_{10}-m_{11}-m01}{2})^{m_{00}}(\tfrac{1}{1-s})^{m_{10}+m_{11}+l_{u0}+m_{01}+m_{00}+l_{u1}} \\
&\qquad\times f_+^{l_{u1}}(s) f_-^{l_{u0}}(s) \\
&\times\frac{e^{\frac{-(\frac{f_\emptyset(s)}{4}+\frac{f_\emptyset(s)-m_{10}}{2}+\frac{f_\emptyset(s)-m_{10}-m_{11}}{4}+\frac{f_\emptyset-m_{10}-m_{11}-m01}{2}+f_++f_-)}{1-s}}}{m_{10}!m_{11}!m_{01}!m_{00}!l_{u1}!l_{u0}!}
\end{aligned}$$

Let's define p(s) as,

$$p(s) = \sum_{\substack{m_{10}+m_{11}-l_{u0}<m_{01}+m_{00}-l_{u1} \\ m_{10}+m_{11}+m_{10}+m_{00}<d}} p(s, m_{10}, m_{11}, l_{u0}, m_{01}, m_{00}, l_{u1})$$

Finally, we can obtain the required system of differential equations from Equation 4.1 by substituting $P(v_t\_is\_flipped)$ by $p(s)$.

(ii) For any component $e_\star$ of $\mathbf{e}$, we have $|e_\star(t-1) - e_\star(t)|$ is no more than the number of clauses containing v_t. Since the probability that v_t is contained in a clause is $\frac{2}{n}$, we know the probability that there are k clauses containing v_t is $\binom{\sigma n}{k}(\frac{2}{n})^k$. For large enough n, we get

$$P(v_t \text{ appears in more than } n^{\frac{1}{5}} \text{ clauses}) = \sum_{k=n^{\frac{1}{5}}}^{n} \binom{\sigma n}{k}(\frac{2}{n})^k$$

$$= \sum_{k=n^{\frac{1}{5}}}^{n} \frac{\sigma n(\sigma n-1)...(\sigma n-k+1)2^k}{k!n^k} \leq \frac{(2\sigma)^{n^{\frac{1}{5}}}n}{n^{\frac{1}{5}}!} = o(n^{-3})$$

(iii) Although functions defined in **(i)** are not well defined at $s = 1$, we can use a standard method to overcome this problem. For each $\varepsilon > 0$, we define set D by points with $s \leq 1 - \varepsilon$ only. Since probability is nonnegative and bounded by 1, we know our functions are not only continuous but also converge uniformly in D. Therefore, it satisfies Lipschitz condition.

As shown above, our model satisfies all three conditions of Theorem 4.3.1. Therefore, we conclude the following theorem by applying Theorem 4.3.1.

Theorem 4.3.2 *For any positive σ, there is a constant c such that for a random 2-CNF $F(n, \sigma n)$ with elimination width d almost surely the GreedySearch algorithm finds an assignment which satisfies $cn + o(n)$ clauses.*

In the following tables, we look at the relationship between (σ, d) and constant c from Theorem 4.3.2. We compare the value of c given by our model with the empirical result obtained by GS, and show indeed our model gives a good approximation of the empirical result. The empirical result is the average over ten random formulas.

Table 4.2: Result of formulas with 100 variables

(σ, d)	(6,10)	(10,15)	(15,20)	(20,25)	(25,30)
c (experiment)	5.24	8.77	12.8	17.1	21.2
c (model)	5.24	8.71	12.8	16.8	20.9

Table 4.3: Result of formulas with 1000 variables

(σ, d)	(6,10)	(10,15)	(15,20)	(20,25)	(25,30)
c (experiment)	5.36	8.78	12.97	17.16	21.33
c (model)	5.24	8.65	12.74	16.78	20.92

As we shown in above tables, the values of c given by our model are comparable to the values obtained by GS. However, they do not agree completely. The small differences between these two values are caused by the approximation in numerical calculations. Limited by the current computational power, it is impossible to determine in reasonable time the exact value of probability function $P(v_t\_is\_flipped)$ for m as small as 20. In order to find a good approximation for the function, we have to explore many different sampling methods in different ranges. After numerous experiments, we finally found a range in which the approximation is good enough to give us a stable value of c.

Chapter 5

Conclusion

In this thesis, we first considered exact algorithms for solving MAX2SAT instances with bounded treewidth k. We know the standard dynamic programming algorithm has time complexity $O(2^{k+3}n)$ and space complexity $O(2^k)$. Since the space complexity is exponential in k, the algorithm takes too much memory space when k is large. To save memory space, we designed a divide and conquer algorithm that has time complexity $O(n^k)$ with linear space complexity. For $k > \log(n)$, the algorithm uses much less memory space than the dynamic programming algorithm. Moreover, we showed that when the tree decomposition is smooth and balanced, the divide and conquer algorithm can be modified so that it have the same time complexity as the dynamic program algorithm while using only linear memory space. We also improved the running time of the divide and conquer algorithm using balanced separator decomposition. We showed that the divide and conquer algorithm works well when the separator width of a graph is logarithmically less than its tree width. In this case, the time complexity of the algorithm is comparable or better to the time complexity of the dynamic programming algorithm, but uses much less memory space.

Besides exact algorithms, we also designed an approximation algorithm, GS, which is shown to perform well on d-degenerate graphs. The class of d-degenerate graphs contains many graph families such as planar graphs, bounded tree width graphs, and so on. We conducted an experiment that compares the performance of GS with OLS and GSAT on random formulas. We also prove rigorously that the algorithm with high probability finds an assignment which satisfies $cn + o(n)$ clauses for a random 2-CNF formula with density σ and elimination width d. We showed experimentally that the value of c obtained from the model agrees with the experimental results.

Bibliography

[1] Teresa Alsinet, Felip Manya, and Jordi Planes. Improved exact solvers for weighted Max-SAT. In *SAT*, pages 371–377, 2005.

[2] Eyal Amir. Approximation algorithms for treewidth, 2002.

[3] Stefan Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded, decomposability—a survey. *BIT*, 25(1):2–23, 1985.

[4] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic and Discrete Methods*, 8(2):277–284, 1987.

[5] Stefan Arnborg and Andrzej Proskurowski. Linear time algorithms for NP-hard problems restricted to partial k-trees. *Discrete Applied Mathematics*, 23(1):11–24, 1989.

[6] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Inf. Process. Lett.*, 8(3):121–123, 1979.

[7] Marshall W. Bern, Eugene L. Lawler, and A. L. Wong. Linear-time computation of optimal subgraphs of decomposable graphs. *J. Algorithms*, 8(2):216–235, 1987.

[8] Hans L. Bodlaender. Dynamic programming on graphs with bounded treewidth. In *ICALP*, pages 105–118, 1988.

[9] Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996.

[10] Hans L. Bodlaender. A partial -arboretum of graphs with bounded treewidth. *Theor. Comput. Sci.*, 209(1-2):1–45, 1998.

[11] Hans L. Bodlaender and Ton Kloks. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *J. Algorithms*, 21(2):358–402, 1996.

[12] Brian Borchers and Judith Furman. A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. *J. Comb. Optim.*, 2(4):299–306, 1998.

[13] Andrei A. Bulatov and Evgeny S. Skvortsov. Efficiency of local search. In *SAT*, pages 297–310, 2006.

[14] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.

[15] Bruno Courcelle. The monadic second-order logic of graphs III: Tree-decompositions, minor and complexity issues. *ITA*, 26:257–286, 1992.

[16] Bruno Courcelle and Stephan Olariu. Upper bounds to the clique width of graphs. *Discrete Applied Mathematics*, 101(1-3):77–114, 2000.

[17] Evgeny Dantsin and Alexander Wolpert. An improved upper bound for SAT. In *SAT*, pages 400–407, 2005.

[18] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[19] Uriel Feige and Mohammad Mahdian. Finding small balanced separators. In *STOC '06: Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, pages 375–384, New York, NY, USA, 2006. ACM.

[20] M. R. Garey, David S. Johnson, and Larry J. Stockmeyer. Some simplified np-complete graph problems. *Theor. Comput. Sci.*, 1(3):237–267, 1976.

[21] Pierre Hansen and Brigitte Jaumard. Algorithms for the maximum satisfiability problem. *Computing*, 44(4):279–303, 1990.

[22] Frank Harary. *Graph Theory.* Addison-Wesley, Massachusetts, 1969.

[23] Paul Hunter and Stephan Kreutzer. Digraph measures: Kelly decompositions, games, and orderings. In *SODA*, pages 637–644, 2007.

[24] Arist Kojevnikov and Alexander S. Kulikov. A new approach to proving upper bounds for MAX-2-SAT. In *SODA*, pages 11–17, 2006.

[25] Shiro Matuura and Tomomi Matsui. New approximation algorithms for MAX 2SAT and MAX DICUT. *Journal of Operations Research Society of Japan*, 46(2):178–188, 2003.

[26] Nysret Musliu. An iterative heuristic algorithm for tree decomposition. In *Recent Advances in Evolutionary Computation for Combinatorial Optimization*, pages 133–150. 2008.

[27] Daniel Raible and Henning Fernau. A new upper bound for Max-2-SAT: A graph-theoretic approach. In *MFCS*, pages 551–562, 2008.

[28] Bruce A. Reed. Finding approximate separators and computing tree width quickly. In *STOC*, pages 221–228, 1992.

[29] Neil Robertson and Paul D. Seymour. Graph minors. I. Excluding a forest. *J. Comb. Theory, Ser. B*, 35(1):39–61, 1983.

[30] Neil Robertson and Paul D. Seymour. Graph minors. X. Obstructions to tree-decomposition. *J. Comb. Theory, Ser. B*, 52(2):153–190, 1991.

[31] Neil Robertson and Paul D. Seymour. Graph minors. XIII. The disjoint paths problem. *J. Comb. Theory, Ser. B*, 63(1):65–110, 1995.

[32] Haiou Shen and Hantao Zhang. Improving exact algorithms for MAX-2-SAT. In *AMAI*, 2004.

[33] Stan P. M. van Hoesel and Bert Marchal. Finding good tree decompositions by local search. *Electronic Notes in Discrete Mathematics*, 32:43–50, 2009.

[34] Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theor. Comput. Sci.*, 348(2):357–365, 2005.

[35] Nicholas C. Wormald. Differential equations for random processes and random graph. *The Annals of Applied Probability*, 5(4):1217–1235, 1995.