An Integer Programming Approach for the Satisfiability Problems

by

LUI Oi Lun Irene

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Philosophy

in

Department of Systems Engineering And Engineering Management

©The Chinese University of Hong Kong

August 2001

To All I love

Abstract

Satisfiability problem is a well-known NP-complete problem. It consists of testing whether the clauses in a Conjuctive Normal Form can all be satisfied by certain consistent assignment of binary values to variables. If it is consistent, the problem is said to be satisfiable; otherwise, it is unsatisfiable. The 3-SAT randomized problem is the smallest NP-complete problem in SAT. In the literature, many transformations have been proposed in converting the satisfiability problem into an integer programming problem. These transformations usually create extra variables and constraints that would enlarge the problem size.

In this thesis, we propose a new transformation method with no extra variables introduced and a single surrogate constraint is resulted at the end of the process. This singly-constrained zero-one polynomial problem can be then solved by certain solution techniques in integer programming problem, such as branch-and-bound methods. We suggest some branch-and-bound algorithms to tackle the resulted singly-constrained zero-one polynomial problem. Revised branch and bound rules are proposed to improve the efficiency of a basic algorithm. Analytical results show that the revised branch-and-bound algorithm has a great improvement compared with the basic one, in terms of the computation time and the number of backtracking.

摘要

可滿足性問題（Satisfiability problem）是一個著名的NP 完全問題，它是指是否存在一種對一組布爾變量的賦值使所給的由若干個子句組成的合取範式的值爲眞。若這種賦值方法是存在的，則問題稱爲可滿足的；反之，則稱不可滿足的。僅含三個子句的隨機可滿足性問題是SAT中一個最小規模的NP完全問題。有關文獻中提出了許多把可滿足性問題轉換成整數規劃問題的方法，遺憾的是這些方法通常會產生額外的變量和約束，從而擴大了問題的規模。

本文建出了一種新的轉換方法，它以一個替代約束代替原有所有約束而不產生額外的變量。轉換後得到的單約束0-1多項式問題可用分支定界法來求解。本文首先提出了一些基本的分支定界算法，進而提出了改進的分支和定界的法則來提高運算的效率；對運算結果的分析顯示了改進的分支定界算法，較之基本的方法，無論在計算的時間上，還是在回溯的次數上，均有極大的改善。

Acknowledgements

I would like to express my deepest gratitude to my supervisor Professor Duan LI for his guidelines and suggestions. Without his advice, this thesis cannot be completed.

I would also like to thank a lot of my friends, Dr. Sha Dan, Chu Kwok Fai, Tommy Lee, Jacky Wong, Ada Ng, Cindy Lam, Gary Lee, Jason Choi, Chiu Chun Hung, Ada Luk, Jessica Hui, Paul Fung, Wong Kam Lai, Yvonne Sho, Sunny Man and Bernard Hui. They bring me much happiness in leisure time and help me a lot in study. In this two-year graduate study, we share many unforgettable memories and a fruitful university life.

Finally, I may express my deepest thanks for the God. He enlight me and set me free from the sadness and anxiety. The support and encouragement from my fellows activate the motion of my research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Satisfiability Problem

The propositional satisfiability problem (SAT) has been classified as the first NP-complete problem. It consists of testing whether the clauses in a propositional formula F in Conjunctive Normal Form (CNF) can all be satisfied by some consistent assignments of true values (0 or 1) to variables. If it is the case, F is said to be $satisfiable$. Otherwise, F is $unsatisfiable$. Moreover, if each clause exactly contains r literals, the subproblem is called r-SAT problem. 3-SAT is the smallest NP-complete subproblem of SAT with its computation time being O(2^n) while 2-SAT problem is solvable in polynomial time [1, 2, 3, 4].

1.2 Motivation of the Research

Besides Davis-Putman-Loveland procedure and $SATZ$ methods, satisfiability problems can be solved by integer programming methods or semidefinite programming methods. After transforming the SAT problem into an integer programming problem, the performance in solving the transformed problem may not be as good as that of the original.

Warren and Alain [5] suggested to convert a pseudo-Boolean function into constrained 0-1 polynomial problem. Lagrangian relaxation and roof duality techniques can solve the non-convex polynomial problem by testing the duality gap between the primal problem and its relaxation. The resulting objective value can be viewed as the upper bound of the problem if the duality gap exists.

The above consideration motivates us to figure out a procedure to convert the CNF-SAT problem into an integer programming formulation. In our research, Branch-and-Bound algorithm can be used to solve the zero-one singly-constrained polynomial problem. Thus, we need to develop our branch rule and bound rule that are suitable for our problem. The continuous relaxation model from [5] can be used to find the upper bound of the subproblem. The existence of duality gap can be checked by verifying whether the solution is an integer instead of the consistency of the 0-1 quadratic posiform suggested by [5]. As a result of studying the branching rules, a better understanding in solving SAT by an IP formulation is achieved.

1.3 Overview of the Thesis

The thesis is organized as follows. Chapter 2 gives a brief review of the satisfiability problem and its solution techniques. Methods like DPL, $SATZ$ and SDO are discussed and several SAT solvers are listed in the last section. Integer programming, in particular, zero-one programming and its continuous relaxation are important techniques adopted in this thesis. We provide some basis on them in Chapter 3. Branch-and-Bound methods, Cutting plane methods, duality methods and heuristic methods are discussed. Solvers found on the internet are listed also.

We present a two-step transformation for converting an original 3-SAT problem into a singly-constrained zero-one polynomial problem in Chapter 4. The first step of the transformation changes m SAT clauses into m integer programming constraints where $m = n * 4.25$ and n is the number of variables in the original problem, while the second step constructs the resultant zero-one singly-constrained polynomial problem from the transformed integer constraints. In Chapter 5, we describe a basic branch-and-bound method.

Chapter 6 defines a revised bound rule for the branch-and-bound method. The revised bound rule is based on the continuous relaxation of the polynomial integer constrained maximization problem shown in [5]. We use a dual simplex method to figure out the bound for the subproblem after converting the integer singly-constrained polynomial problem into a continuous constrained maximization problem. *Cplex* is a solver used to implement the revised bound rule. The example in Chapter 5 is tested for a determination of the improvement. A revised branch rule is presented in Chapter 7. We consider a similar formula as in [6] to set the weight for variables so as to find out the branching variable at each iteration. The example in both Chapter 5 and 6 is used here again to examine the revised branch-and-bound method.

Experimental results are reported in Chapter 8. We compare the performance between the basic branch-and-bound method and the revised one with different sample sizes. *SATZ* is also used to compare with the two branch-and-bound methods. Finally, we conclude the thesis in Chapter 9 by summarizing our contributions and listing some possible directions for future research.

Chapter 2

Constraint Satisfaction Problem and Satisfiability Problem

This chapter provides the background of the thesis. Satisfiability (SAT) problem was the first problem shown to be NP-complete [7]. SAT is a cornerstone of computational complexity theory, and thus is commercially important since thousands of practical combinatorial problems would benefit from a highly efficient SAT solver. Its applications include graph coloring, Boolean N-queens induction, circuit diagnosis and scheduling problem [8, 9]. In this chapter, we will define what a satisfiability problem is and how to solve the SAT problem. Also, we will list some solvers in the last section that have been released recently.

2.1 Constraint Programming

Constraint Programming is built upon constraints and constraint solving [10]. The three most important types of constraints in constraint programming are *arithmatic constraints*, *tree constraints* and *finite domain constraints*. There are three fundamental operations in solving a constraint programming problem. First, we need to determine whether a constraint is *satisfiable*. Then, we use *simplication* to rewrite a constraint in a form that makes its information more apparent. Last, we find out the "best " solution under some conditions through an *optimization* method.

Primitive constraint consists of a constraint relation symbol from its domain, D, together with the appropriate number of arguments [10]. They are constructed from the constants, functions of D and other variables. A *user-defined constraint* is a constraint in the form of $p(t_1, t_2, \ldots, t_n)$ where p is an n-ary predicate and $t_1, t_2, \ldots, t_n$ are expressions from the constraint domain [10]. A *formula* is a primitive constraint involving variables that return the value of True or False. A *literal* is either a primitive constraint or a user-defined constraint [10]. In this thesis, the literals we considered are primitive constraints.

A *Constraint Satisfaction Problem* (CSP) consists of a constraint C over variables $x_1, x_2, \ldots, x_n$ and a domain D that maps each variable x_i to a finite set of values, written $D(x_i)$, that are allowed to take. The CSP can be used to represent the constraint $C \wedge x_1 \in D(x_1) \wedge \ldots \wedge x_n \in D(x_n)$, [10]. Complete search strategies for constraint satisfaction problems are based on propagate-and-branch and they only assign integral values to the variables whose domain includes finite integers [9]. The search starts from a partial assignment of the variables. The power of CSP methods stems from strong propagation algorithms (eg. arc-consistency) that rule out all variable values that are known to be inconsistent with the current partial variable assignment and the set of constraints (local consistency). Search in CSP progresses in the space of partial variable assignments where variable values are assigned with the goal to eventually assign one value to *every* variable, such that the solution is optimal. Hence, the search in CSP maintains integrality and local consistency, but relaxes totality, i.e. require that all variables be assigned one value.

A CSP can also be defined as below. A CSP consists of a set of variables, each with a finite possible value (domain), and a set of constraints which the values assigned to the variables must satisfy, [11]. Some discrete optimization problems in operational research can be also formulated as constraint satisfaction problems. In an optimization constraint satisfaction problem, there is an objective. Each time a solution to the CSP is found, a new constraint is added to ensure that any future solution must have an improved value of the objective, and this continues until the problem becomes infeasible. The last solution found is the optimal solution of the problem.

2.2 Satisfiability Problem

Consider n propositional variables $X = \{x_1, \ldots, x_n\}$. Let $L = \{x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n\}$ denote the set of corresponding literals, where $\bar{x}_i = 1 - x_i$ denotes the *complement* of x_i. Let $x = (x_1, \ldots, x_n) \in B^n$, where $B = \{0, 1\}$, denote the vectors of binary assignments to the propositional variables. A *clause* is the disjunction of a subset of the literals, and a *Conjunctive Normal Form* (CNF) is a Boolean formula of the form

$$\Phi_{\mathcal{C}}(x) = \bigwedge_{C \in \mathcal{C}} (\bigvee_{u \in C} u),$$

where $\mathcal{C} \subseteq 2^L$ is a family of clauses, [12].

The satisfiability (SAT) problem for Boolean formulas in conjuctive normal form (CNF) was the first problem that was shown to be NP-complete, [7, 13, 14]. It consists of finding a binary assignment $x^* \in B^n$ satisfying all the clauses of a given CNF F, i.e., $\Phi_F(x^*) = 1$. It tests whether clauses in F can all be satisfied by some consistent assignment of true values (0 or 1) to variables, [15, 16]. F is said to be satisfiable if there exists a feasible assignment. Otherwise, F is said to be unsatisfiable. Moreover, if each clause exactly contains r literals, the subproblem is called r-SAT problem. It has been revealed that 2-SAT problem is solvable in polynomial time [1, 2, 3, 4] and Even, Itai and Shamir [17] outlined a linear-time algorithm for 2-SAT. Schaefer [18] also claimed a polynomial time bound for an evaluation problem with 2-SAT although he did not prove it. However, 3-SAT is the smallest NP-complete subproblem of SAT where its computation time is within $O(2^n)$. On the other hand, if the ratio of the number of clauses to the number of variables is approximately equal to 4.25 for random 3-SAT problem [19, 20], these problems are very diffcult to be solved.

2.3 Methods in Solving SAT problem

There are many different kinds of methods in solving *Conjunctive Normal Form Satisfiability* ($CNF-SAT$) problem. The best complete method to solve SAT problems is *Davis-Putnam-Loveland* (DPL) procedure.

2.3.1 Davis-Putnam-Loveland Procedure

Davis-Putnam-Loveland (DPL) procedure was defined by Martin Davis, George Logemann and Donald Loveland in 1962 [21, 15]. The DPL procedure was based on the idea in [22]. The algorithm of [22] consists of two interlocking parts. The first part is the $QFI - Generator$ that generates a growing propositional calculus formulas in conjunctive normal form which are called the "quantifier-free lines". The second part is the $Processor$ that tests the consistency of these propositional calculus formulas at regular stages in its "growth". This test is proceeded by first eliminating *one-literal* clauses and then atomic formulas all of whose occurrences are positive or negative.

Finally, the remaining atomic formulas are to be eliminated by using the *Rule for Eliminating Atomic Formulas.* This Rule can be formulated as

$$\textit{Rule III}\ (A \vee p)\ \&\ (B \vee \bar{p})\ \&\ R \quad [21, 22]$$

where A, B and R are free of p. This can be done by grouping the clauses containing p and crossing out the occurrences of p to obtain A, grouping the clauses containing $\bar{p}$ and crossing out the $\bar{p}$ to obtain B, and grouping the remaining clauses to obtain R. Then, F is consistent if and only if $(A \vee B)\&R$ is consistent.

The DPL procedure replaces the *Rule III* by *Splitting Rule* which is defined as

$$\textit{Rule III}^*\ (A \vee p)\ \&\ (B \vee \bar{p})\ \&\ R \quad [21]$$

where A, B and R are free of p. Then F is inconsistent if and only if $A\&R$ and $B\&R$ are both inconsistent. *Rule III** is more powerful because *Rule III* can easily incorporate large number and long length of the clauses in the expression. Many duplicated and redundant clauses are allowed to present. Also, *Rule III* seldom yields new $one - literal$ clauses, whereas *Rule III** often does.

2.3.2 $SATZ$ by Chu-Min Li

The Davis-Putnam-Loveland procedure has been so far the best complete method to solve SAT problems. It constructs a binary search tree for solving F and each recursive called constitutes a node of the tree. In [15], Chu-Min Li indicated that random 3-SAT problems become difficult to solve on average when the ratio of clause number to variable number is approximately equal to 4.25 [19, 20]. He explained that the shape of a search tree is highly related to the width of a search tree rather than the mean height of a search tree. The mean height of the class of Hard problems is the same as that of the class of Easy problems. However, the width of the class of

Hard problems is substantially larger than that of the class of Easy problems. It is difficult to distinguish satisfiable and unsatisfiable problems here because they give the same shape of figures. Also, the widest level of a search tree is at the mid-depth approximately equal to $n/20$ where n is the number of variables in the problem.

SATZ by Chu-Min Li is a DPL procedure that branches on the variable that would reduce the largest number of clauses on F at each iteration. Let $w(x)$ be the number of clauses reduced when x is assigned 1 and $w(\bar{x})$ be the number of clauses reduced when x is assigned 0. Then x is weighted by the equation suggested by Freeman [6],

$$H(x) = w(x) * w(\bar{x}) * 1024 + w(x) + w(\bar{x}).$$

From here, we know that the product term $w(x) * w(\bar{x})$ gives more information in selecting the branching variable in order to balance the search tree. The value of 1024 may be used for a quicker multiplication since $1024 = 2^{10}$. For example, there are 10 clauses reduced when x_1 is assigned 1 and 6 clauses reduced when x_1 is assigned 0. And there are 8 clauses reduced when x_2 is assigned 1 and 8 clauses reduced when x_2 is assigned 0. We know that the weights of x_1 and x_2 are 61456 and 65552 respectively. We would choose x_2 as the next branching variable.

On the other hand, Chu-Min Li has made some improvements to DPL procedure. In order to speed up a DPL procedure, Li proposes to reduce the width of a search tree instead of reducing its mean height. That means, his procedure could reach a dead-end as early as possible. He had 2 suggestions. First, his procedure would prepare and generate more and stronger constraints (A constraint is stronger if it suppresses more solutions). If F has n variables, it has 2^n possible solutions. Two binary clauses sharing a complementary literal, like $x_1 \vee x_2$ and $\bar{x}_1 \vee x_3$, remove 2^{n-1} solutions while two binary clauses sharing an identical literal or having no common variable, such as $x_1 \vee x_2$, $x_1 \vee x_3$ or $x_1 \vee x_2$ and $\bar{x}_3 \vee x_4$, remove $2^{n-1} - 2^{n-3}$ and $2^{n-1} - 2^{n-4}$ solutions respectively. Obviously, binary clauses sharing complementary literals remove much more solutions and have more chances to lead to a dead-end where all solutions are removed. So the DPL procedure would branch next on a variable that generates more binary clauses sharing complementary literals. Therefore, the weight of the literal x would be changed to

$$w(x) = \sum_{l \vee l' \textit{ is produced by } x=1} [f(\bar{l}) + f(\bar{l'})],$$

where $f(\bar{l})$ is the number of binary occurrences of $\bar{l}$ in F if there is a sufficient number (10 as suggested by Li) of binary clauses in F, otherwise it is the number of weighted occurrences of $\bar{l}$ in F. $w(\bar{x})$ is similarly defined. Li [15] then obtains the weight of variable x by replacing the value of both $w(x)$ and $w(\bar{x})$ in Freeman's formula [6].

Li also suggests to use *looking further forward* to search a dead-end. *Unit propagation* is a look-ahead[1] searching technique. If the satisfaction of a literal l reduces many clauses, i.e., it introduces many strong constraints by unit propagation, it probably leads to an imminent dead-end which can be reached by further unit propagations. The idea is that if $UnitPropagation(F \cup \{l\})$ reduces more than T (it is fixed to 65 for hard random 3-SAT problems [15]) clauses, for every variable y in the newly produced binary clauses occurring both positively and negatively in binary clauses, $UnitPropagation(F \cup \{l\} \cup \{y\})$ and $UnitPropagation(F \cup \{l\} \cup \{\bar{y}\})$ are executed. If both propagations reach a dead-end, $\bar{l}$ should be satisfied. These two propagations are called *unit propagations of second level.* This has a great impact on the size of a search tree. This keeps a balance in the search tree for the branching variable x. It is because if $w(x) >> w(\bar{x})$ or $w(\bar{x}) >> w(x)$, x will not be selected as a branching variable. Therefore, the formula for selecting the branching variable would be more powerful.

[1]Please refer to Appendix A for details

In addition, Li figures out that branches on a variable randomly among the *best* variables may provide a better performance if it creates simpler subproblems. He found that the randomized version of $SATZ$ could solve the benchmark problem 2670-400 in UCSC in 95 seconds while the revised version of $SATZ$ took more than 7200 seconds to solve the problem [15]. Nevertheless, a constraint hypothesis that considers unit propagations in deeper branches might reach a dead-end faster than simplified hypothesis that he has used. However, more constraints might be generated during the unit propagation process.

2.3.3 Local Search for SAT

Local search strategies have recently been used in solving propositional satisfiability problem [9]. It searches a satisfying variable assignment for a set of clauses. It moves locally to "flip" variables that are chosen according to a randomized greedy strategy. A randomized algorithm is used to select a starting point for the local search and/or to drive the search into different regions of the search space, which can reduce the dependency of the local search on its starting point and thereby make it less dependent on restarts. Hill-climbing is one of the heuristic procedures in solving SAT problems. It needs some techniques to overcome trapping in the local minima. Most SAT local search algorithms have several static and dynamic policies to select the next variable to be flipped. They are the key component to determine the performance of SAT local search algorithms.

For static policy, the probability of a variable to be flipped depends on the current variable assignment and on the scheme to compute the *score* of a variable. For example, it depends on the number of clauses satisfied after the flip and the number of clauses broken due to the flip. For dynamic policy, the decision can depend on the history of the search. For instance, a Tabu element can break ties between variables with an equal score according to how recently a variable has been flipped.

2.3.4 Integer Linear Programming Method for SAT

Linear programming (LP) was the first continuous optimization problem to be investigated. Conjunctive Normal Form Satisfaction problem (CNF-SAT) can be solved by an integer linear programming (ILP) model [23]. Monfroglio constructed the SAT problem as ILP in [23]. The resulting matrix has a regular structure and is no longer problem-specified. It does not depend on the structure of the clauses, but the number of clauses and the number of variables. The structure of the integer program allows us to solve the problem by using standard linear programming techniques. Simplex algorithm is one of the methods in solving linear programming problems. However, the computational complexity of the simplex algorithm is exponential in the worst case. The standard form of linear programming is

$$\begin{array}{ll} \text{min} & cx \\ \text{s.t.} & Ax = b, x \geq 0. \end{array}$$

An integer linear programming problem is

$$\begin{array}{ll} \text{min} & cx \\ \text{s.t.} & Ax = b, x \geq 0, \text{ x integer.} \end{array}$$

To convert the $CNF - SAT$ problem into ILP problem, Monfroglio [23, 24] suggests to add nonnegative slack or surplus variables in order to convert all inequalities to equlities, replace all unrestricted variables by differences of nonnegative variables, delete all redundant rows and take the negative value of an objective function to be maximized. However, this procedure may make the problem more difficult because the additional variables and additional constraints enlarge the size of the original $CNF - SAT$ problem. The worst case is to have $[n * 2m + 2m * n * (n-1)/2]$ for the number of columns and $[n + 2m * n * (n - 1)/2]$ for the rows in matrix A, where n is the number of literals and m is the number of clauses in the original $CNF - SAT$ problem. On the other hand, E. de klerk [25] proposes to convert the SAT problem into an integer programming problem through elippic representation and uses a semi-definite programming method to solve the problem.

2.3.5 Semidefinite Programming Method

Semidefinite programming method is a new solution concept in optimization. Interior point methods for semidefinite programming (SDP) have been studied recently due to their polynomial complexity and practical efficiency. Most of these methods are extensions of linear optimization algorithms [26]. The most common solution approach in integer programming is to relax the integrality constraints to linear constraints and subsequently solve the resulting *LP relaxation* [27, 28]. Unfortunately, the LP relaxation of (IP_{SAT}) is weak. It is easily checked that the trival all-zero solution is always feasible when no unit clause is present. By introducing some objective function, the solution can be steered away from the trival one. The incumbent solution may be found by rounding the solution to the LP relaxation.

There are several different ways of constructing primal-dual search directions in SDP. The usual scheme is to apply linearization in conjunction with symmetrization to the perturbed optimality conditions of the SDP problem. A boolean quadratic representation of a 3-clause can be used in converting a 3-SAT problem into a semidefinite programming problem. This representation can be studied as an integer programming problem. Consider the $3 - clause$ $p_1 \vee p_2 \vee p_3$. All valid quadratic representations for this clause follow from [27, 25, 29] are listed in Table 2.1.

After converting the problem, we can use Lagrangian dual techniques to solve the

$$\begin{array}{ll}
x_1x_2 + x_1x_3 - x_2 - x_3 & \leq 0 \\
x_1x_2 + x_2x_3 - x_1 - x_3 & \leq 0 \\
x_1x_3 + x_2x_3 - x_1 - x_2 & \leq 0 \\
-x_1x_2 - x_1x_3 - x_2x_3 - 1 & \leq 0 \\
-x_1x_2 + x_1 + x_2 - 1 & \leq 0 \\
-x_1x_3 + x_1 + x_3 - 1 & \leq 0 \\
-x_2x_3 + x_2 + x_3 - 1 & \leq 0 \\
x_1, x_2, x_3 & \in \{-1, 1\}.
\end{array}$$

Table 2.1: Quadratic representation for 3-CNF-SAT clause

problem. Besides, E. de Klerk proposed another transformation of the SAT problem [25]. Each clause $p_1 \vee p_2 \vee p_3$ has its *elliptic representation*:

$$(x_1 + x_2 + x_3 - 1)^2 \leq 4,\ x_1, x_2, x_3 \in \{-1, 1\}.$$

In general, $p_1 \vee p_2 \vee \ldots \vee p_n$ has elliptic representation:

$$(x_1 + \ldots + x_n - 1)^2 \leq (n-1)^2.$$

If the elliptic relaxation is infeasible, the original problem is unsatisfiable. Newton system for dual interior point methods can efficiently solve the problem. This *elliptic semidefinite feasibility problem* is *satisfiability-equivalent* to SAT for several classes of polynomially solvable formulas included 2-SAT, pigeonhole formulae and unsatisfiable formulae from graph colouring instances where clique constraints imply unsatisfiability [29]. However, there is a gap between the relaxation and its primal problem. This *gap relaxation* is always reported as feasible for 3-SAT problems if no 2-literals clauses present. These problems are reported as satisfiable even they are infeasible.

2.4 Softwares for SAT

Many SAT solvers can be found on the internet. The solution schemes in these softwares include conventional SAT techniques, complete and incomplete searching algorithms.

2.4.1 SAT01

SAT01 solver proves its efficiency empirically on *DIMACS SAT benchmarks.* It turns out that the majority of these SAT instances are fairly easy. Unfortunately, some instances were not solved because of large memory requirements. Since SAT01 solver was designed to be a general solver for NP problems, it has no specialization for SAT.

2.4.2 *SATZ* and *SATZ213*, contributed by Chu-Min Li

SATZ and *SATZ213* are two smart solvers for SAT problems. These two solvers are developed by Chu-Min Li. *SATZ* implements the algorithm proposed by Chu-Min Li that improves the traditional DPL algorithm by considering a new formula in calculating the weights for variables and using the unit propagation as the searching technique. *SATZ213* is a modification of *SATZ.*

2.4.3 Others

We list some softwares for SAT available on Internet in this section. Details can be found in "http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/solvers.html".

Stochastic Local Search Algorithms (incomplete algorithm)

- GSAT, Version 41 (contributed by Henry Kautz and Bart Selman)

- WalkSAT, Version 35 (contributed by Henry Kautz and Bart Selman)

Systematic Search Algorithms (complete algorithm)

- GRASP (version of Feb. 2000; contributed by Joao P. Marques da Silva)

- NTAB (via James Crawford's home page)

- POSIT, Version 1.0 (contributed by Jon W. Freeman)

- REL_SAT, Version 2.00 (contributed by Roberto Bayardo)

- REL_SAT-rand, Version 1.0 (contributed by Henry Kautz)

- SATO, Version 3.2.1 (contributed by Hantao Zhang)

- $SATZ$-rand, Version 4.7 (contributed by Henry Kautz)

- $SATZ$-rand, Version 2.0 (contributed by Carla Gomes, Henry Kautz, and Bart Selman)

Chapter 3

Integer Programming

In last chapter, we explained what a satisfiability problem is and how to solve such kind of problem. In this chapter, we are going to review the integer programming. There are many different ways in solving integer programming. We will introduce some solution methods in the literature.

3.1 Introduction

Many real-world problems in design, operation and management may be formulated as optimization problems in which we are seeking for some values of decision variables. The objective function takes an extreme value while satisfying all the constraints imposed on these decision variables [30]. *Integer Programming* (IP) is a branch of *mathematical programming* where its decision variables are integers [30, 31]. In general, integer programming can be a constrained one or an unconstrained one, while the constraints and the objective functions can be linear or nonlinear, [32]. The general integer programming problem can be defined as:

$$\begin{array}{lll}
\text{maximize (or minimize)} & z = f(x_1, x_2, \dots, x_n) & \\
\text{subject to} & g_i(x_1, x_2, \dots, x_n) \le b_i, & i \in M = 1, 2, \dots, m \\
 & x_j \ge 0 & j \in N = 1, 2, \dots, n \\
 & x_j \text{ an integer} & j \in I \subseteq N
\end{array}$$

If $I = N$, the problem is a *Pure Integer Programming* (PIP) problem, that means all the variables x_j are restricted to integers. Otherwise, if $I \subset N$, the decision variables consist of both integer and real variables. The problem is called a *Mixed Integer Programming* (MIP) problem. If the constraints and the objective function are both linear in an integer programming problem, the problem is called a *Linear Integer Programming* problem. If the domain of the variables is binary, i.e., the decision variables can only be assigned one of the two values, the problem is called *Binary Integer Programming* (BIP) problem.

In reality, many decision making situations can be formulated as integer programming problems. Examples include *Assignment Problem, 0-1 Knapsack Problem, Cutting-Stock Problem, Capacitated Plant Location Problem, Set Covering Problem, Traveling Salesman Problem* (TSP), *Capital Budgeting Problem, Sequencing Problem, Scheduling Problem* and *Fixed-Charge Problem* [30, 32, 33].

3.1.1 Formulation of IPs and BIPs

Before solving the IP problems and the BIP problems, formulation should be done systematically and properly. We should well define the problem instances and the decision variables used in the model [33]. The following are some guidelines.

(1) Define the necessary decision variables used in the model.

(2) Define the domain of the decision variables.

(3) Define the objective function as a function of the decision variables.

(4) Define a set of constraints imposed on the decision variables.

3.1.2 Binary Search Tree

The enumeration tree of a BIP problem can be represented as a binary tree. A binary tree consists of external (leaf) nodes and internal (non-leaf) nodes. Every internal node of a full binary tree has exactly two children. A binary search tree is a data structure of retrieving objects associated with keys. It corresponds to a full binary tree with one key stored at every internal node, and none at external nodes [34].

3.2 Methods in Solving IP problem

Integer programming solution techniques can be generally classified into two types: the search methods and the cutting methods [32]. The first type is motivated by the fact that the feasible solution points of most integer programming problems are finite (e.g., a subset of $\{0, 1\}^n$). We seek for the solution by enumerating "all" these points. This type of solution techniques includes implicit numeration techniques and branch-and-bound techniques. The implicit numeration is suitable for the zero-one integer programming problem, and may actually be considered as a speical case of the branch-and-bound methods. Clearly, the efficiency of the resulting "search" algorithm depends on the power of the techniques that are developed to discard nonpromising solution points.

Cutting methods are developed for the mixed or pure integer linear programming problem. The idea is from the fact that the solution in the simplex method must occur at an extreme point. It adds constraints, that are violated by the current noninteger solution but never by any feasible integer pointi, in the original problem. Successive applications of such a procedure should eventually result in a new convex solution space with its optimum extreme point properly satisfying the integrality condition.

It cuts off those infeasible parts of the continuous solution space. However, the cutting methods can only be applied to convex problems. Also, no feasible solution is obtained until the very end of the search procedure. In the following sub-sections, we will describe four methods used in solving integer programming which are *Branch-and-Bound Methods*, *Cutting-Plane Methods*, *Duality Methods* and *Heuristic Methods.*

3.2.1 Branch-and-Bound Method

The basic concept underlying the branch-and-bound technique is dividing and conquering. Since the original "large" problem is too difficult to be solved directly, it is divided into smaller and smaller subproblems until these subproblems can be conquered. The dividing (branching) is done by partitioning the entire set of feasible solutions into smaller and smaller subsets. The conquering (fathoming) is done partially by bounding how good the best solution in the subset can be, and then discarding the subset if its bound indicates that it cannot contain an optimal solution for the original problem.

Branching

When dealing with binary variables, the most straightforward way to partition the set of feasible solutions into subsets is to fix the value of one of the variables (say, x_1) at $x_1 = 0$ for one subset and at $x_1 = 1$ for the other subset.

Figure 3.1 portrays the division procedure (branching) into subproblems by a tree with branches (arcs) from the ALL node (corresponding to the whole problem having ALL feasible solutions) to the two nodes corresponding to the two subproblems. This tree, which will continue this "branch growing" iteration by iteration, is referred to as the solution tree (or enumeration tree) for the algorithm. The variable selected to

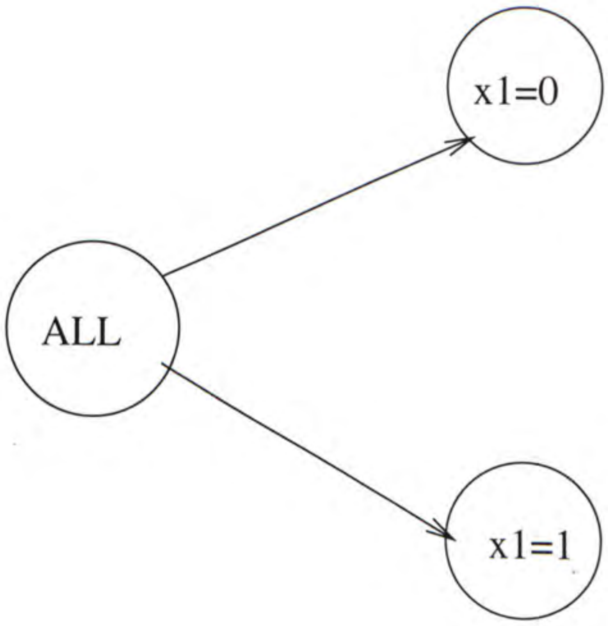

Figure 3.1: The Solution tree created by the branching for the first iteration of the BIP branch-and-bound algorithm

do a branching at any iteration by assigning its value (as with x_1 above) is called the branching variable.

One of these subproblems can be conquered (fathomed) immediately, whereas the other subproblem will need to be divided further into smaller subproblems by setting $x_2 = 0$ or $x_2 = 1$, etc.

For other IP problems where the integer variables have more than two possible values, the branching can still be done by setting the branching variable at its respective individual values, thereby creating more than two new subproblems. However, a good alternate approach is to specify a range of values (eg. $x_j \leq 2$ or $x_j > 3$) of the branching variable for each new subproblem.

Bounding

For each of these subproblems, we need to obtain a bound on how good its best feasible solution can be. The standard way of doing this is to solve a simpler relaxation of the subproblem. In most cases, a relaxation of a problem is obtained by deleting ("relaxing") one set of constraints that have made the problem difficult to solve. For IP problems, the most troublesome constraints are the ones that require the respective variables to be integer. Therefore, the most widely used relaxation is the *LP-Relaxation* that deletes integer constraints. In the next section, we would talk more about how to find out the bound by using *LP-Relaxation*.

Fathoming

A subproblem can be conquered (fathomed), and thereby dismissed from further consideration, if one of the following three tests is satisfied

Test 1. Its bound $\leq Z^*$ where $Z^* = max\ cx : x \in S_t$ and S_t is the feasible set of x.

Test 2. Its LP-Relaxation has no feasible solution.

Test 3. The optimal solution for its LP-Relaxation is integer.
(If this solution is better than the incumbent, it becomes the new incumbent, and Test 1 is reapplied to all unfathomed subproblems with the new larger Z^*.)

Unfortunately, the worst time complexity of branch-and-bound method is $O(2^n)$. A very large number of branches and nodes may be created before a feasible solution is found. The storage space may thus be large.

Techniques in Branch-and-Bound Method

There are two typical techniques in branch-and-bound-methods : *Best-first branch-and-bound* method and *Depth-first branch-and-bound* method.

Best-first Branch-and-Bound technique applies the branching process to the subproblem with the best bound. It terminates either when all nodes are reached or when the optimal solution has been found.

Depth-first Branch-and-Bound technique applies the branching process to a subproblem that has been generated by the last branching step. The procedure backtracks to the most recent alternatives when a solution has been found or when infeasibility has been encountered. It terminates when the optimal solution has been found or when all the search space has been reached. We tend to use this technique in solving CSP.

3.2.2 Cutting-Plane Method

Consider the integer programming problem of the following form

$$(P) \quad v(P) = \max \{cx | Ax = b, x \geq 0, x \in Z^n\}$$

where Z^n is the set of all integer points in R^n.

If an optimal solution to the linear programming relaxation, i.e., the solution to the problem $(\bar{P})$

$$(\bar{P}) \quad v(\bar{P}) = v(\bar{P}_0) = \max \{cx | Ax = b, x \geq 0\},$$

is not integer, then *cutting-plane methods* can be applied. The idea of cutting plane method consists in adding a new functional constraint (cut) $\alpha x \leq \beta$ which cuts off the optimal solution to $\bar{P}_0$ from $v(P)$ [30]. This reduces the feasible region for the LP relaxation without eliminating any feasible solutions for the IP problem. Then, we can obtain a new linear programming problem $(\bar{P}_1)$ from the original problem (P). If its optimal solution is an integer, it would be the optimal solution to (P) also. Otherwise, we can add a new cut to $\bar{P}_1$ to obtain $\bar{P}_2$ and so on. Assuming the sequence P_0, P_1, ... is finite, there must exist an index r such that $\bar{x} \in v^*(\bar{P}_r)$ is an integer.

There are many methods for finding the cutting planes. Hillier and Lieberman [35] proposed a method for generating cutting planes for pure BIP problems. It first considers any constraint in "$\leq$" form with only non-negative coefficients. Then, it finds a group of variables which is called *minimum cover* of the constraint such that the constraint is violated if every variable in the group equals to 1 and all other variables equal to 0, but the constraint becomes satisfiable if the value of any one of these variables is changed from 1 to 0. The resulting cutting plane would be

$$\boxed{\text{sum of the variables in the minimum cover} \leq N-1}$$

where N is the number of variables in the *minimum cover*. For example, we have a constraint $3x_1 + 4x_2 + 2x_3 + 5x_4 \leq 10$. The minimum covers of the constraint are (x_1, x_2, x_4), (x_2, x_3, x_4) and (x_1, x_2, x_3, x_4). Therefore, the cutting planes of the contraint would be $x_1 + x_2 + x_4 \leq 2$, $x_2 + x_3 + x_4 \leq 2$ and $x_1 + x_2 + x_3 + x_4 \leq 3$.

Wolsey, Walukiewicz, and Garfinkel and Nemhauser [30, 31, 33] proposed another cutting plane method for integer programming problems. The idea is to generate a *Chvátal-Gomory inequaliy* on the constraint associated with a chosen basic variable after finding an optimum from the linear programming relaxation. This approach was the first cutting plane method which was proposed by Gomory [36]. The problem (P) can be rewritten as

$$\begin{array}{ll} max & \bar{a}_{00} + \sum_{j \in NB} \bar{a}_{0j} x_j \\ \text{s.t.} & xB_u + \sum_{j \in NB} \bar{a}_{uj} x_j = \bar{a}_{u0} \text{ for } u = 1, \ldots, m \\ & x \geq 0 \text{ and integer} \end{array}$$

with $\bar{a}_{oj} \leq 0$ for $j \in NB$, $\bar{a}_{u0} \geq 0$ for $u = 1, \ldots, m$ and $xB_u = 1$ for xB_u are basic variables not in NB, where NB is the set of nonbasic variables. If the basic optimal solution x^* is not an integer, there exists some row u with $\bar{a}_{u0} \notin Z^1$. By choosing this row, the *Chvátal – Gomory* cut for row u is

$$xB_u + \sum_{j \in NB} \lfloor \bar{a}_{uj} \rfloor x_j \leq \lfloor \bar{a}_{u0} \rfloor.$$

We can rewrite this inequality by elminating xB_u, giving

$$\sum_{j \in NB} (\bar{a}_{uj} - \lfloor \bar{a}_{uj} \rfloor) x_j \geq \bar{a}_{u0} - \lfloor \bar{a}_{u0} \rfloor.$$

Since $0 \leq f_{uj} < 1$ and $0 < f_{u0} < 1$, this inequality cuts off x^* as $x_j^* = 0$ for all nonbasic variables $j \in NB$ in the optimal LP solution. As a result, the slack variable s is a nonnegative integer variable where

$$s = -(\bar{a}_{u0} - \lfloor \bar{a}_{u0} \rfloor) + \sum_{j \in NB} (\bar{aj} - \lfloor \bar{a}_{uj} \rfloor x_j).$$

Consider Example 8.9 in [33],

$$\begin{array}{lll}
z = \max & 4x_1 - x_2 & \\
 & 7x_1 - 2x_2 & \leq 14 \\
 & x_2 & \leq 3 \\
 & 2x_1 - 2x_2 & \leq 3 \\
 & x_1, x_2 & \geq 0 \text{ and integer}
\end{array}$$

By adding integer slack variables x_3, x_4, x_5, we can obtain the solution of the above problem by solving this LP problem,

$$\begin{array}{lll}
z = \max 59/7 & -4/7x_3 - 1/7x_4 & \\
 & x_1 + 1/7x_3 + 2/7x_4 & = 20/7 \\
 & x_2 + x_4 & = 3 \\
 & -2/7x_3 + 10/7x_4 + x_5 & = 23/7 \\
 & x_1, x_2, x_3, x_4, x_5 & \geq 0 \text{ and integer.}
\end{array}$$

Since the optimal linear programming solution is $x = (20/7, 3, 0, 0, 23/7) \notin Z^5_+$, we get the cut

$1/7x_3 + 2/7x_4 \geq 6/7$ or $s = -6/7 + 1/7x_3 + 2/7x_4$

with s, x_3, $x_4 \geq 0$ and integer. Then, we add this constraint to the original constraint set and get the solution $x = (2, 1/2, 1, 5/2, 0)$. Then, we generate the cut again for x_2, i.e. $1/2x_5 \geq 1/2$ or $t = -1/2 + 1/2x_5$. As a result, we find an integer solution $x = (2, 1)$.

3.2.3 Duality in Integer Programming

indent Consider the linear programming problem,

$$\begin{array}{ll} \max & z = cx \\ \text{s.t.} & Ax \leq b \\ & x \geq 0 \end{array}$$

which is called the *primal problem*. There is a related LP,

$$\begin{array}{ll} \min & v_0 = vb \\ & vA \geq c \\ & v \geq 0 \end{array}$$

called the *dual* problem where v is an m-dimensional row vector [31, 32]. The dual problem is derived from its primal by using the following conditions:

1. The primal objective is maximization and the dual objective is minimization.

2. The number of variables in the dual is equal to the number of constraints in the primal.

3. The number of constraints in the dual is equal to the number of variables in the primal.

4. The coefficients in the objective function of the primal form the right-hand side of the dual.

5. The right-hand side of the primal forms the coefficient of the objective of the dual.

6. All variables are nonnegative in both problems.

Obviously, the dual of the dual is the primal itself, so that either problem may be called the primal or the dual. Note that $x_0^c = v_0^c$ where x_0^c is the optimal objective value in the primal problem and v_0^c is the optimal objective value in the dual problem. Exactly one of the following four relationships between the primal and dual problems must hold:

1. Both the primal and dual problems have optimal solutions and $x_0^c = v_0^c$.
2. The primal is unbounded and the dual is infeasible.
3. The dual is unbounded and the primal is infeasible.
4. The primal and dual are both infeasible.

Here, we know that by solving the primal (dual), we can solve the dual (primal). For LP, we can apply the simplex method to either one. Sometimes, it is possible to start with a dual feasible solution and solve the dual using the same tableau that we use in the primal simplex method. This algorithm is called the *dual simplex* method.

The Lagrangian Duality

Lagrangian Duality is one of the Dual techniques in optimization. In [37], we know that the Lagrangian Dual of a linear integer programming problem is usually formulated as a linear programming problem. The objective value in each LP subproblem is an upper bound of its original IP subproblem. Thus, Lagrangian Relaxation is usually used as the bounding relaxation for the Branch-and-Bound method. If the solution from the Lagragian relaxation is an integer, it may be a feasible solution of the original integer programming problem. We can stop, check its feasibility and compare its objective value with the incumbent to see whether it is an optimal solution for the problem. If the solution is not an integer, we can compare its objective value with the bound in the upper level. If it is better, we can branch further. Otherwise, we prune this branch and then backtrack to the preceding node.

An alternative approach to implement Lagrangian Duality without using linear programming problem is a *subgradient algorithm*. A *subgradient* at u of a convex function $f : R^m \rightarrow R^1$ is a vector $\gamma(u) \in R^m$ such that $f(v) \geq f(u) + \gamma(u)^T(v-u)$ for all $v \in R^m$ [33]. For a smooth convex function f, $\gamma(u) = \nabla f(u) = (\partial f/\partial u_1, \ldots, \partial f/\partial u_m)$ is the gradient of f at u. The subgradient algorithm for the Lagrangian dual can be described as follows.

1. Initialization. $u = u^0$.

2. Iteration k. Set $u = u^k$. Solve the Lagrangian problem $IP(u^k)$ with optimal solution $x(u^k)$.

2.1 Set $u^{k+1} = max\{u^k - \mu_k(d - D_x(u^k)), 0\}$ and $k \leftarrow k+1$.

3. Step size. Set $\mu_k = \epsilon_k(\bar{w} - z(u^k))/\Sigma_{i\in V}(2 - \Sigma_{e\in\delta(i)}x_e(u^k))^2$ where $\bar{w}$ is the upper bound in the Lagrangian Duality and $\mu_k rightarrow 0$ with $\sum_{i=1}^{+\infty} \mu_k = +\infty$. Go to step 2.

The difficulty of this algorithm is in choosing the step size $\{\mu_k\}_{k=1}^{\infty}$. The convergence of the series $\{\mu_k\}$ may be too slow. This may affect the solution process. Also, a duality gap may exist in Lagrangian relaxation because the subgradient algorithm is often terminated before the optimal value is obtained.

3.2.4 Heuristic Algorithm

Many combinatorial problems are very difficult to be formulated as an IP or MIP when the instance is large. Even if it can be formulated as an IP or MIP, it is still difficult or impossible to find feasible solutions by using a branch-and-bound method due to its large size. Sometimes, it is easy to find feasible solutions by inspection or knowing some problem structures. A general-purpose MIP approach without using specific problem structures is usually ineffective. Hence, heuristic or approximation algorithms may be preferable because it can find a "good" feasible solution quickly. Examples of applications of heuristic methods include time-tabling and sports scheduling problems, set covering, assignment problems and capacitated production planning problems [9].

3.3 Zero-one Optimization and Continuous Relaxation

Consider a problem of maximizing a general pseudo-Boolean function. We can formulate it as a constrained 0-1 polynomial problem. An upper bound on the maximum objective function value can be computed by constructing a Lagrangian dual. This Lagrangian dual yields an objective value equal to the roof dual value for the original 0-1 polynomial problems. Since this Lagrangian dual can be transformed to a linear programming problem, we can easily find out the roof or even the solution by checking whether a roof duality gap exists. This algorithm constitutes a basis for the bounding scheme in the branch-and-bound algorithm in this thesis.

3.3.1 Introduction

Consider the following unconstrained 0-1 polynomial programming problem

$$\text{(P1)} \qquad Z_{P1} = \max_{x \in \{0,1\}^n} h(x) = \sum_{i=1}^n l_i x_i + \sum_{k=1}^n q_k \prod_{x \in S(k)} x_k.$$

Here, k is the number of nonlinear terms and for each $k = 1, \cdots, K$, $S(k) \subseteq I = \{1, 2, \cdots, n\}$ is the index set of binary variables whose product with a nonzero constant q_k defines the kth term.

From the paper, *Roof duality for polynomial 0-1 optimization* [5, 38], we can introduce complemented variables $\bar{x}$ into $h(x)$ where $\bar{x}_i = 1 - x_i$, $\forall i = 1, 2, \cdots, n$.

By replacing a variable by its complement in the negative term of degree 2 or more, (P1) can be reformulated such that all terms of degree 2 or more have nonnegative coefficients, each term contains at most one complemented variable, and no term contains the same variable in both the complemented and uncomplemented forms. Without loss of generality, (P1) can be rewritten as

$$\text{(P2)} \qquad Z_{P2} = \max_{x \in 0,1^n \bar{x} \in 0,1^{|I_c|}} f(x, \bar{x}) = \sum_{i=1}^n l_i x_i + \sum_{k \in P} d_k \prod_{i \in Q(k)} x_i + \sum_{k \in N} c_k \bar{x}_{T(k)} \prod_{i \in R(k)} x_i$$

where P represents the index set of all nonlinear terms which do not contain a complemented variable, N represents the index set of all nonlinear terms which contain a complemented variable, $Q(k) \subseteq I \; \forall k \in P$, and $R(k) \subseteq I$ with $T(k) \notin R(k) \; \forall k \in N$. For each term $k \in N$, $T(k)$ is the index of the associated complemented variable. Also, we define the set I_C as the index set of all variables x whose complement appears in (P2) and I_N as the index set of all variables x whose complement does not appear in (P2). Notice that $I_c \bigcup I_N = I$.

3.3.2 The Roof Dual expressed in terms of Lagrangian Relaxation

From [5], we can find a linear function $p(x)$ as a roof for problem (P2) iff

$$p(x) = \sum_{i=1}^n l_i x_i + \sum_{k \in N} \{v_k(1 - x_{T(k)}) + \sum_{i \in R(k)} u_{ik} x_i\} + \sum_{k \in P} \{\sum_{i \in Q(k)} \lambda_k x_i\}$$

where $v_k \in N$, $u_{ik} \; \forall (i,k) \; \exists i \in R(k)$ and $k \in N$, and $\lambda_{ik} \; \forall (i,k) \; \exists i \in Q(k)$ and $k \in P$ are scalars satisfying $(v, u, l) \geq 0$ with $v_k + \sum_{i \in R(k)} u_{ik} = c_k \; \forall k \in N$ and $\sum_{i \in Q(k)} \lambda_{ik} = d_k \; \forall k \in P$.

Clearly, any roof $p(x)$ is an upper bounding linear function of $f(x, \bar{x})$ since for all x binary,

$$\sum_{i\in Q(k)} \lambda_{ik} x_i \geq \prod_{i\in Q(k)} x_i,\ \forall k \in P$$

$$v_k\{1 - x_{T(k)} + \sum_{i\in R(k)} u_{ik} x_i\} \geq c_k \bar{x}_{T(k)} \sum_{i\in R(k)} x_i\ \forall k \in N$$

Let R represent the set of all roofs, the roof dual is defined as

$$W(R) = \min_{p(x)\in R} \max_{x\in(0,1)^n} p(x),$$

and this value can be computed by solving the linear programming problem

$$\begin{array}{lll}
\text{(LP)} & Z_{LP} \max \sum_{i=1}^{n} l_i xi + \sum_{k\in P} d_k t_k + \sum_{k\in N} c_k w_k, & \\
\text{s.t.} & t_k \leq xi & \forall (i,k)\ \exists i \in Q(k) \text{ and } k \in P, \\
 & w_k \leq 1 - x_{T(k)} & \forall k \in N, \\
 & 0 \leq x_i \leq 1 & \forall i \in I, \\
 & t_k \geq 0 & \forall k \in P, \\
 & w_k \geq 0 & \forall k \in N.
\end{array}$$

where $t_k = \prod_{i\in Q(k)} x_i\ \forall k \in P$ and $w_k = \bar{x}_{T(k)} \prod_{i\in R(k)} x_i\ \forall k \in N$.

The set of roofs is complete in the sense that $f(x) = \min_{p(x)\in R} p(x)$ for all x binary. The value $W(R)$ is an upper bound on Z_{P1} and the *roof duality gap* is defined as

$$W(R) - Z_{P1} = \min_{p(x)\in R}\{\max_{x\in(0,1)^n p(x)}\} - \max_{x\in(0,1)^n}\{\min_{p(x)\in R} p(x)\}.$$

3.3.3 Determining the Existence of a Duality Gap

In order to determine the existence of a duality gap between (P1) and its Lagrangian relaxation, we can convert (P1) into the following form by adding some redundant constraints,

$$\text{(P4)}\ Z_{P4} = \max_{x\in(0,1)^n \bar{x}\in(0,1)^{|I_c|}} f(x,\bar{x}) = \begin{array}{l} \sum_{i\in I} l_i x_i + \sum_{k\in P} d_k \prod_{i\in Q(k)} x_i + \\ \sum_{k\in N} c_k \bar{x}_{T(k)} \prod_{i\in R(k)} x_i \end{array}$$

$$\begin{array}{lll}
\text{s.t.} & x_i + \bar{x}_i = 1 & \forall i \in I_c, \\
& \prod_{i\in Q(k)} x_i \leq x_i & \forall (i,k)\ \exists i \in Q(k) \text{ and } k \in P, \\
& \bar{x}_{T(k)} \prod_{i\in R(k)} x_i \leq \bar{x}_{T(k)} & \forall k \in N, \\
& \bar{x}_{T(k)} \prod_{i\in Q(k)} x_i \leq x_i & \forall (i,k)\ \exists i \in R(k) \text{ and } k \in N, \\
& x_i \geq 0 & \forall i \in I, \\
& \bar{x}_i \leq 1 & \forall i \in I_N, \\
& \bar{x}_i \geq 0 & \forall i \in I_C.
\end{array}$$

The continuous relaxation of (P4), (CP4), is obtained by treating all product terms as continuous variables and eliminating binary restrictions for the x and $\bar{x}$ by $t_k = \prod_{i\in Q(k)} x_i\ \forall k \in P$ and $w_k = \bar{x}_{T(k)} \prod_{i\in R(k)} x_i\ \forall k \in N$,

$$\begin{array}{llll}
\text{(CP4)} & Z_{CP4} = \max \sum_{i\in I} l_i x_i + \sum_{k\in P} d_k t_k + \sum_{k\in N} c_k w_k & & \\
\text{s.t.} & x_i + \bar{x}_i = 1 & \forall i \in I_C & (24) \\
& t_k \leq x_i & \forall (i,k) \exists i \in Q(k) \text{ and } k \in P & (25) \\
& w_k \leq \bar{x}_{T(k)} & \forall k \in N & (26) \\
& w_k \leq x_i & \forall (i,k)\ \exists i \in R(k) \text{ and } k \in N & (27) \\
& x_i \geq 0 & \forall i \in I & (28) \\
& x_i \leq 1 & \forall i \in I_N & (29) \\
& \bar{x}_i \geq 0 & \forall i \in I_C & (30)
\end{array}$$

From [5], we know that if there is no duality gap exists between (P4) and (CP4), (P4) and (CP4) would be consistent by considering a 0-1 quadratic posiform $p(x, \bar{x})$ where

$$\begin{array}{rl}
p(x,\bar{x}) = & \sum_{i\in I\beta_i\neq 0} x_i + \sum_{i\in I_N\gamma_i\neq 0} \bar{x}_i + \sum_{i\in I_C\gamma_i\neq 0} \bar{x}_i + \\
& \sum_{k\in P} \sum_{i\in Q(k)\lambda_{ik}\neq 0} \sum_{j\in Q(k) j\neq i} x_i\bar{x}_j + \sum_{k\in N v_k\neq 0} \sum_{i\in R(k)} \bar{x}_{T(k)}\bar{x}_i + \\
& \sum_{k\in N} \sum_{i\in R(k) u_{ik}\neq 0} (x_i x_{T(k)} + \sum_{j\in R(k) j\neq i} x_i \bar{x}_j).
\end{array}$$

However, the value of this $p(x, \bar{x})$ is very difficult to be solved. Luckily, we found out that $p(x, \bar{x})$ is consistent ONLY if all the variables are integer, i.e. $x_i = 0$ or 1. So, if the solution from CP4 is an integer solution, P4 and CP4 would be consistent. We can prove by considering the example in [5].

Consider the unconstrained 0-1 cubic programming with 4 variables

$\max_{x\ binary} h(x) = -x_3 + x_1x_3 - x_1x_4 + x_2x_3 + x_3x_4 - x_1x_2x_3 + x_1x_2x_4 - x_2x_3x_4$

by substituting $x_i = 1 - \bar{x}_i$ for the smallest indexed variable in each of the negative nonlinear term x_1x_4, $x_1x_2x_3$ and $x_2x_3x_4$, this problem can be rewritten as

$\max_{x,\bar{x}\ binary} f(x,\bar{x}) = -x_3 - x_4 + x_1x_3 + \bar{x}_1x_4 + \bar{x}_1x_2x_3 + x_1x_2x_4 + \bar{x}_2x_3x_4$

By using *CPLEX*[1], the solution is $(x_1 = x_2 = x_3 = x_4 = 0.5)$. The 0-1 posiform is

$p(x,\bar{x}) = x_1\bar{x}_2+x_1\bar{x}_3+x_1x_4+x_1\bar{x}_4+\bar{x}_1x_4+\bar{x}_1\bar{x}_2+\bar{x}_1\bar{x}_2+\bar{x}_1\bar{x}_3+\bar{x}_1\bar{x}_4+x_2x_3+\bar{x}_2x_4+x_3\bar{x}_4$

Obviously, $p(x,\bar{x})$ is inconsistent and so a duality gap exists.

If we change the coefficients of 1 and -1 on the terms x_3x_4 and $x_2x_3x_4$ to 4 and -4 in the original function $h(x)$, the problem becomes

$\max_{x\ binary} h(x) = -x_3 + x_1x_3 - x_1x_4 + x_2x_3 + 4x_3x_4 - x_1x_2x_3 + x_1x_2x_4 - 4x_2x_3x_4$ and

$\max_{x,\bar{x}\ binary} f(x,\bar{x}) = -x_3 - x_4 + x_1x_3 + \bar{x}_1x_4 + \bar{x}_1x_2x_3 + x_1x_2x_4 + 4\bar{x}_2x_3x_4$

From *CPLEX*, we can find out the solution for the problem which is $(x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 1)$ and $(x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1)$. The 0-1 posiform

$p(x,\bar{x}) = x_1x_2+\bar{x}_1x_2+x_1\bar{x}_3+\bar{x}_1\bar{x}_4+x_2x_3+x_2\bar{x}_3+x_2x_4+x_2\bar{x}_4+\bar{x}_2\bar{x}_3+\bar{x}_2\bar{x}_4+x_3\bar{x}_4+\bar{x}_3x_4$

is consistent with the above solutions. From these examples, we find that only integer solution would give $p(x,\bar{x})$ equal to 1. Thus, we can prove that the 0-1 posiform $p(x,\bar{x})$ can only be consistent if the solution is an integer solution.

3.4 Software for solving Integer Programs

There are many softwares for integer programming problem. MINTO is a software system that solves mixed-integer linear programs by a branch-and-bound algorithm with linear programming relaxations. It also provides automatic constraint classification, preprocessing, primal heuristics and constraint generation. Users can enrich the basic algorithm by providing a variety of specialized application routines that can customize MINTO to achieve maximum efficiency for a problem class.

[1]*CPLEX* is a commercial LP solver that can be written in C language. Please refer to section 6.1.1 for more details.

The heart of MINTO is a linear programming based branch-and-bound algorithm. It can be implemented on top of any LP-solver, like *CPLEX* that provides capabilities to solve and modify linear programs and interpret their solutions. Details can be found in http://akula.isye.gatech.edu/ mwps/projects/minto.html.

To be as effective and efficient as possible when used as a general purpose mixed-integer optimizer, MINTO attempts to:

- improve the formulation by preprocessing and probing;
- construct feasible solutions;
- generate strong and valid inequalities;
- perform variable fixing based on reduced prices;
- control the size of the linear programs by managing active constraints.

Chapter 4

Integer Programming Formulation for SAT Problem

To solve the 3-CNF SAT problem, we can convert a satisfiability problem into an integer constrained maximization problem formulation. We can then use a branch-and-bound algorithm to find out if a feasible solution exist for the original problem. By considering the Venn Diagram for 3 independent parties, we can set up some axioms for converting the CNF clauses into IP constraints. In this chapter, we would show how to set up these axioms and convert the problem into a singly-constrained 0-1 polynomial problem.

4.1 From 3-CNF SAT Clauses to Zero-One IP Constraints

There are many methods to convert the 3-SAT clauses into IP constraints. Etienne de Klerk [25] proposed to use *Semidefinite Programming* (SDP) approach to solve the

$(x_1 \vee x_2 \vee x_3) = x_1 + x_2 + x_3 - x_1x_2 - x_1x_3 - x_2x_3 + x_1x_2x_3$
$(x_1 \vee x_2 \vee \bar{x}_3) = 1 - x_3 + x_1x_3 + x_2x_3 - x_1x_2x_3$
$(x_1 \vee \bar{x}_2 \vee \bar{x}_3) = 1 - x_2x_3 + x_1x_2x_3$
$(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) = 1 - x_1x_2x_3$

Table 4.1: Axioms for transforming the 3-SAT clause into Integer Programming format

SAT problem. As we stated in chapter 2, there are 7 forms of valid *elliptic representations* for 3-literal clauses [25]. Any clause can be transformed to a combination of these *elliptic representations.* The domain of the variables would be -1 or 1 instead of 0 or 1. If these *elliptic representations* are infeasible, the primal 3-CNF formula will be unsatisfiable. However, the *elliptic* semidefinite feasiability problem is "satisfiabilty-equivalent" to 2-SAT. It is always feasible for "pure 3-SAT" problem. It has to make some adjustment so that it can detect the unsatisfiability of 3-SAT problem.

Actually, we have an easier way to convert the CNF clauses into IP constraints. Consider the Venn diagram with 3 independent parties in Figure 4.1. The shared area of the union of three parties is the sum of these 3 parties, minus the intersaction of 2 out of these 3 parties, and plus the intersaction of these 3 parties. Thus, we can have

$$\begin{aligned}(x_1 \vee x_2 \vee x_3) &= x_1 + x_2 + x_3 - (x_1 \wedge x_2) - (x_1 \wedge x_3) - (x_2 \wedge x_3) + (x_1 \wedge x_2 \wedge x_3) \\ &= x_1 + x_2 + x_3 - x_1x_2 - x_1x_3 - x_2x_3 + x_1x_2x_3\end{aligned}$$

In addition, we can rewrite $\bar{x}_i$ as $(1 - x_i)$ and form the axioms listed in Table 4.1.

Since each literal can only take the true value 0 or 1, each clause should return the value 1 if it is consistent. Therefore, we can transform the CNF clause into an IP constraint as in Table 4.2.

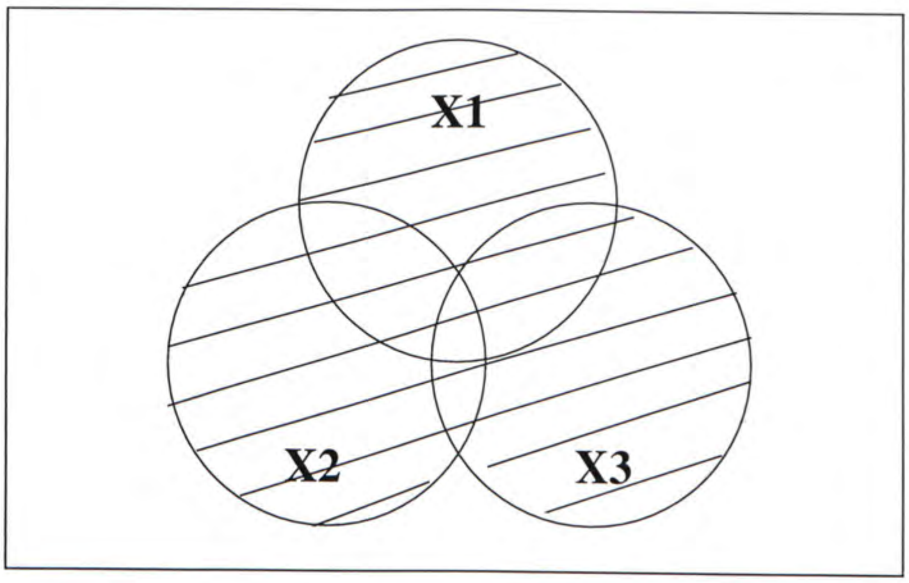

Figure 4.1: Venn Diagram for 3 independent parties x_1, x_2 and x_3

$(x_1 \vee x_2 \vee x_3) \Rightarrow x_1 + x_2 + x_3 - x_1x_2 - x_1x_3 - x_2x_3 + x_1x_2x_3 \geq 1$
$(x_1 \vee x_2 \vee \bar{x}_3) \Rightarrow 1 - x_3 + x_1x_3 + x_2x_3 - x_1x_2x_3 \geq 1$
$(x_1 \vee \bar{x}_2 \vee \bar{x}_3) \Rightarrow 1 - x_2x_3 + x_1x_2x_3 \geq 1$
$(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \Rightarrow 1 - x_1x_2x_3 \geq 1$

Table 4.2: IP constraint for converting 3-SAT clause

For example,

$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_4) \Rightarrow \begin{array}{l} x_1 + x_2 + x_3 - x_1x_2 - x_1x_3 - x_2x_3 + x_1x_2x_3 \geq 1 \wedge \\ 1 - x_2x_4 + x_1x_2x_4 \geq 1 \end{array}$$

Now, we can convert m $(= 4.25 * n)$ clauses into m IP constraints, where n is the number of variables in the original CNF problem.

4.2 From m-Constrained IP Problem to Singly-Constrained IP Problem

At this stage, we have total m IP constraints where each constraint is a polynomial represenation with its value larger than or equal to 1. That is,

$$g_1 \geq 1,\ g_2 \geq 1,\ \ldots,\ g_m \geq 1$$

where $m = 4.25 * n$ and n is the number of variables in the original problem.

In this thesis, the domain of variables is binary (0 or 1). There is no doubt that each IP constraint can only return the value of 0 or 1, i.e., $g_i \in \{0, 1\}$. If the problem is feasible, such that all the constraints are satisfied, the sum of these constraints should be equal to m, $\sum_{i=1}^{m} g_i(x) = m$. If the sum of these constraints is less than m ($\sum_{i=1}^{m} g_i(x) < m$), the original problem must be infeasible. Thus, we can combine these m IP constraints together into an unconstrained zero-one polynomial maximization problem, that is

$$max \sum_{i=1}^{m} g_i(x).$$

If the original problem is feasible, the optimal objective value should return a value of m. If the value of the optimal objective value is smaller than m, that means the original problem is infeasible. Therefore, the original problem and the singly-constrained zero-one polynomial problem are equivalent.

4.2.1 Example

In order to implement the transformation, we generate a 5-variables problem randomly. There are 22 (4.25*5) clauses where each clause contains exactly 3 literals. The problem is listed as below:

$x_4 \vee \bar{x}_2 \vee x_5$

$\bar{x}_3 \vee x_5 \vee \bar{x}_2$

$\bar{x}_4 \vee \bar{x}_1 \vee \bar{x}_5$

$x_2 \vee \bar{x}_3 \vee \bar{x}_4$

$\bar{x}_5 \vee x_4 \vee \bar{x}_1$

$x_2 \vee \bar{x}_3 \vee x_1$

$\bar{x}_5 \vee \bar{x}_2 \vee \bar{x}_3$

$\bar{x}_4 \vee x_1 \vee x_3$

$x_5 \vee \bar{x}_1 \vee \bar{x}_4$

$\bar{x}_3 \vee \bar{x}_2 \vee \bar{x}_4$

$\bar{x}_1 \vee x_3 \vee x_5$

$x_5 \vee \bar{x}_3 \vee \bar{x}_4$

$x_1 \vee \bar{x}_2 \vee x_4$

$\bar{x}_5 \vee x_4 \vee \bar{x}_2$

$x_2 \vee x_4 \vee \bar{x}_5$

$\bar{x}_5 \vee x_1 \vee \bar{x}_3$

$x_3 \vee x_2 \vee \bar{x}_1$

$\bar{x}_1 \vee x_2 \vee \bar{x}_4$

$x_5 \vee x_1 \vee x_2$

$\bar{x}_3 \vee \bar{x}_2 \vee x_5$

$\bar{x}_5 \vee \bar{x}_3 \vee \bar{x}_1$

$x_4 \vee \bar{x}_2 \vee x_3$

Then, we convert these 3-CNF-SAT clauses into IP constraints. The problem can be formulated as

$x_4 \vee \bar{x}_2 \vee x_5 \Rightarrow 1 - x_2 + x_2x_4 + x_2x_5 - x_2x_4x_5 \geq 1$

$\bar{x}_3 \vee x_5 \vee \bar{x}_2 \Rightarrow 1 - x_2x_3 + x_2x_3x_5 \geq 1$

$\bar{x}_4 \vee \bar{x}_1 \vee \bar{x}_5 \Rightarrow 1 - x_1x_4x_5 \geq 1$

$x_2 \vee \bar{x}_3 \vee \bar{x}_4 \Rightarrow 1 - x_3x_4 + x_2x_3x_4 \geq 1$

$\bar{x}_5 \vee x_4 \vee \bar{x}_1 \Rightarrow 1 - x_1x_5 + x_1x_4x_5 \geq 1$

$x_2 \vee \bar{x}_3 \vee x_1 \Rightarrow 1 - x_3 + x_1x_3 + x_2x_3 - x_1x_2x_3 \geq 1$

$\bar{x}_5 \vee \bar{x}_2 \vee \bar{x}_3 \Rightarrow 1 - x_2x_3x_5 \geq 1$

$\bar{x}_4 \vee x_1 \vee x_3 \Rightarrow 1 - x_4 + x_1x_4 + x_3x_4 - x_1x_3x_4 \geq 1$

$x_5 \vee \bar{x}_1 \vee \bar{x}_4 \Rightarrow 1 - x_1x_4 + x_1x_4x_5 \geq 1$

$\bar{x}_3 \vee \bar{x}_2 \vee \bar{x}_4 \Rightarrow 1 - x_2x_3x_4 \geq 1$

$\bar{x}_1 \vee x_3 \vee x_5 \Rightarrow 1 - x_1 + x_1x_3 + x_1x_5 - x_1x_3x_5 \geq 1$

$x_5 \vee \bar{x}_3 \vee \bar{x}_4 \Rightarrow 1 - x_3x_4 + x_3x_4x_5 \geq 1$

$x_1 \vee \bar{x}_2 \vee x_4 \Rightarrow 1 - x_2 + x_1x_2 + x_2x_4 - x_1x_2x_4 \geq 1$

$\bar{x}_5 \vee x_4 \vee \bar{x}_2 \Rightarrow 1 - x_2x_5 + x_2x_4x_5 \geq 1$

$x_2 \vee x_4 \vee \bar{x}_5 \Rightarrow 1 - x_5 + x_2x_5 + x_4x_5 - x_2x_4x_5 \geq 1$

$\bar{x}_5 \vee x_1 \vee \bar{x}_3 \Rightarrow 1 - x_3x_5 + x_1x_3x_5 \geq 1$

$x_3 \vee x_2 \vee \bar{x}_1 \Rightarrow 1 - x_1 + x_1x_2 + x_1x_3 - x_1x_2x_3 \geq 1$

$\bar{x}_1 \vee x_2 \vee \bar{x}_4 \Rightarrow 1 - x_1x_4 + x_1x_2x_4 \geq 1$

$x_5 \vee x_1 \vee x_2 \Rightarrow x_1 + x_2 + x_5 - x_1x_2 - x_1x_5 - x_2x_5 + x_1x_2x_5 \geq 1$

$\bar{x}_3 \vee \bar{x}_2 \vee x_5 \Rightarrow 1 - x_2x_3 + x_2x_3x_5 \geq 1$

$\bar{x}_5 \vee \bar{x}_3 \vee \bar{x}_1 \Rightarrow 1 - x_1x_3x_5 \geq 1$

$x_4 \vee \bar{x}_2 \vee x_3 \Rightarrow 1 - x_2 + x_2x_3 + x_2x_4 - x_2x_3x_4 \geq 1$

Finally, we combine all these constraints together to form a singly-constrained polynomial problem as

$-x_1 - 2x_2 - x_3 - x_4 + x_1x_2 + 3x_1x_3 - x_1x_4 - x_1x_5 + 3x_2x_4 - x_3x_4 - x_3x_5 + x_4x_5 - 2x_1x_2x_3 - x_1x_3x_4 - x_2x_3x_4 + x_1x_2x_5 - x_1x_3x_5 + x_1x_4x_5 + x_2x_3x_5 - x_2x_4x_5 + x_3x_4x_5 \geq 1$

Now, we can use branch-and-bound algorithm to solve the resulting singly-constrained zero-one polynomial problem, then check the satisfiability of the original problem. Branch-and-bound algorithm is a simple method in solving the IP problem. In the next chapter, we will show how to choose the branch rule and the bound rule for our IP problem. Also, an example is used to show the details of the algorithm.

Chapter 5

A Basic Branch-and-Bound Algorithm for the Zero-One Polynomial Maximization Problem

After converting the 3-CNF-SAT problem into an IP problem, we would use a branch-and-bound method to find out whether the problem is feasible. To solve an integer programming problem, there are many methods in the literature, such as Branch-and-Bound methods, Cutting Planes, etc. In this chapter, we will express the reason for choosing Branch-and-Bound method as our algorithm rather than others. Also, we will discuss our branch and bound rules in the coming sections.

5.1 Reason for choosing Branch-and-Bound Method

There are many methods in solving an integer programming problem. Branch-and-bound and cutting planes are two typical solution schemes in the literature. Branch-and-bound method is quite popular. However, the time complexity is $O(2^n)$ in the worst case where n is the number of variables. For a large problem, cutting planes may do better. In each iteration, it reduces the feasible region by cutting down the region that does not contain feasible solutions. However, we can only apply cutting planes method to those constraints which are convex. In this thesis, the resulting zero-one polynomial maximization problem does not possess a convexity. Branch-and-bound method seems to be more suitable as being the skeleton of our searching algorithm. For a branch-and-bound algorithm, we have to set up a branch rule and a bound rule for the search procedure. For the branch rule, we have to identify a branching variable at each node by considering the weight of each variable in the constraint. In each iteration, we have to define the upper bound and lower bound for the subproblem. In the following sections, we will discuss how to choose our branch rule and bound rule.

5.2 Searching Algorithm

The searching algorithm in this section would begin by checking whether the problem is solvable. We would first check the feasibility of the problem. If the right hand side of the constraint, b, is non-positive, we could set all the variables at zero and return the solution. If the sum of all the coefficients of the constraint is larger than or equal to the right hand side (*RHS*), then we set all the variables at one and return the solution. If it is neither the above two cases, we have to continue an iteration process. Thus, we have to execute the following procedure.

First, we calculate the number of *occurrence* of each variable according to its *positive terms* and its *negative terms* in the objective. If the number of occurrence of a variable with positive terms is 0, the existence of that variable would lower the *LHS* value (sum of the coefficients of all the variable terms) of the objective. Thus, we set that variable to be 0. On the other hand, if the number of occurrence of a variable with negative terms is 0, then we set that variable to be 1 because the existence of that variable would raise up the *LHS* value.

If it is not the above cases, we need to find out the branching variable by a branch rule. We substitute this branching variable by a suitable value (0 or 1) to the constraint and simplify it. We will discuss more about this in the next section. Next, after updating the constraint, we have to check the *LHS* value and the *RHS* value of the subproblem. If the right hand side of the constraint, b, is non-positive, we would set the unassigned variables at zero, stop and return the solution. If the sum of the coefficients of all the variable terms of the constraint is larger than or equal to the right hand side, we set all the unassigned variables at one, stop and return the solution. Then, we check the satisfiablity of the original problem by substituting the solution into the objective function. Here, we can find out the upper and lower bound of our subproblem. The values of the upper and lower bound are the sum of coefficients of the positive and negative terms, respectively. The upper and lower bound at each iteration should be within the boundaries of its predecessor. In section 5.2.2, we will explain the details of how the bound works. At this stage, if a dead end is reached, we backtrack to the latest node. If a feasible solution is found, we will stop and report the feasibility of the problem. Otherwise, we will branch further by repeating the above procedures until a dead end is reached or all nodes are checked.

5.2.1 Branch Rule

For constrained satisfiability problem, the order of labeling and the domain size of the variables are essential in labeling the variables. Domain of a variable is the range that represents the values of the variable. Labeling is a kind of elimination method which uses a bakctracking search to find a solution to the constraints. The order of labeling is the order of variables being labelled in the binary search tree. A good variable ordering can shorten the time of searching.

In "Programming with Constraints: An Introduction" [10], Marriott and Stuckey proposed to separate the variable list into 2 halves and then form in an order. Besides, we can choose the branching variables according to the domain size of the variables. However, this may not be applicable in this thesis because the domain of the variables is binary. It may be meaningless if we sort the variables depending on their domains. According to this, we may need to think of a suitable branching rule for our thesis.

Let us look at the solution tree shown in Figure 5.1. In each iteration, we have to find a branching variable to explore. If no variable has a zero positive occurrence or a zero negative occurrence in the reduced objective function of a subproblem, we need to find out the branching variable that has the largest occurrence of positive terms. If more than one variable have the same number of the largest occurrence, we choose one involved in a term with the largest absolute coefficient. We set the branching variable at 1.

For example, we have a reduced objective $2x_1 - 4x_3 - x_1x_2 + 2x_2x_3$. The numbers of both positive and negative occurences of these variables are 1. However, the coefficient of one term involving x_3 has the largest absolute value, 4. So, we set x_3 as our branching variable and set $x_3 = 1$.

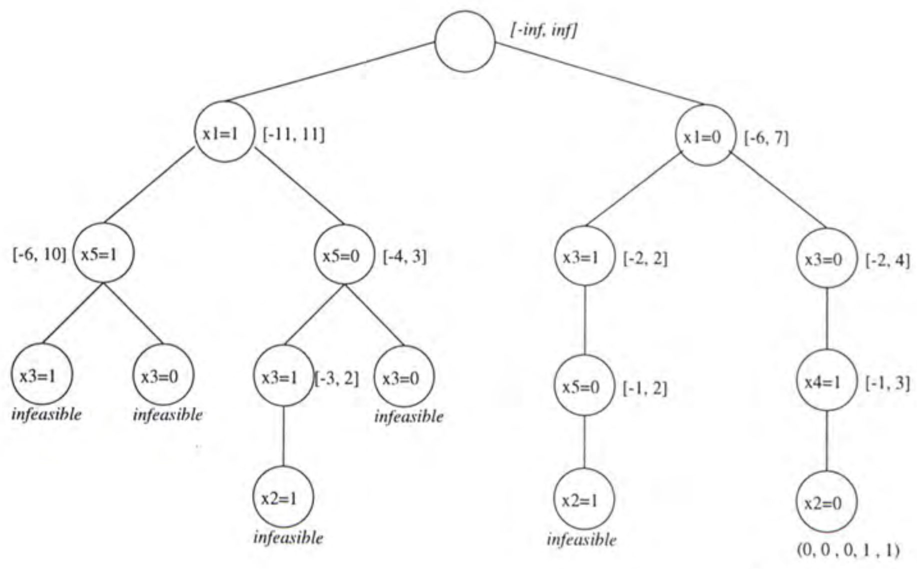

Figure 5.1: The solution tree for checking the satisfiability of a 5-variables problem.

5.2.2 Bounding Rule

For any IP problem, we can find an upper bound and a lower bound of the subproblem. The easiest way is to set the sum of all its positive coefficients as the upper bound and the sum of the negative coefficients as its lower bound. During the iterations, the region of these bounds should become tighter and tighter.

5.2.3 Fathoming Test

The fathoming tests in our algorithm are similar to those in the conventional branch-and-bound method and are listed as follows

Test 1. The upper bound is less than the RHS.

Test 2. An integer optimal solution is found.

In the following section, we will use a simple example with 5 variables in Chapter

4 to test the performance of our algorithm. This example will also be used in the following two chapters to check the preformances of the revised branch and bound rules.

5.2.4 Example

After converting the CNF problem into IP problem as in Section 4.2.1, the constraint becomes

$$-x_1 - 2x_2 - x_3 - x_4 + x_1x_2 + 3x_1x_3 - x_1x_4 - x_1x_5 + 3x_2x_4 - x_3x_4 - x_3x_5 + x_4x_5 - 2x_1x_2x_3 - x_1x_3x_4 - x_2x_3x_4 + x_1x_2x_5 - x_1x_3x_5 + x_1x_4x_5 + x_2x_3x_5 - x_2x_4x_5 + x_3x_4x_5 \geq 1$$

Since the *RHS* of the constraint is positive and the *LHS* value (adding the coefficients of all variable terms) equals to -3, we set the upper and lower bound of the problem as 12 and -15 which are the sum of all positive and negative coefficients, respectively.
Iteration 1:

(1) We set up an occurrence table as below:

	x_1	x_2	x_3	x_4	x_5
+ve	4	4	3	4	5
-ve	6	4	7	6	4

(2) Since there is no variable that contains only positive or negative occurrence, we choose x_5 to be the branching variable and set it at 1.

(3) The constraint can be reduced as

(I): $-2x_1 - 2x_2 - 2x_3 + 2x_1x_2 + 2x_1x_3 + x_2x_3 + 2x_2x_4 - 2x_1x_2x_3 - x_1x_3x_4 - x_2x_3x_4 \geq 1$

(4) The *RHS* of the constraint is positive and the *LHS* value equals to -3. Since the upper bound and the lower bound of the subproblem (I) are 7 and -10 respectively, we continue branching.

```
Searching Algorithm for CSP
Procedure:
(1)     (i)   If the RHS of the constraint is non-positive, set the unassigned variable
              to be 0. Stop and go to (10).
        (ii)  If the sum of all the coefficient of the constraint is larger than the RHS,
              set the unassigned variable to be 1. Stop and go to (10).
(2)           Set the upper bound and lower bound of the primal problem as the sum of
              positive and negative coefficients of all the terms, respectively.
(3)           Calculate the number of occurrence of each variable in the constraint
              according to
        (i)   positive coefficient
        (ii)  negative coefficient
(4)     (i)   If the number of occurrence of variable in (3)(i) is 0, then we set that
              variable to 0.
        (ii)  If the number of occurrence of variable in (3)(ii) is 0, then we set that
              variable to 1.
        (iii) Go to (6).
(5)           Branch Rule:
        (i)   Choose the variable with the largest occurrence in positive term.
        (ii)  If more than one variable have the same weight, choose a variable
              with the largest absolute coefficient of its term.
        (iii) Set the branching variable at 1.
(6)           Update the constraint.
(7)     (i)   If the RHS of the constraint is non-positive, set the unassigned
              variable to be 0. Stop and go to (10).
        (ii)  If the sum of all the coefficient in LHS of the constraint is larger
              than the RHS, set the unassigned variable to be 1. Stop and go to (10).
(8)     (i)   Bound Rule
              Set the upper and lower bound of the subproblem by adding up the
              positive and negative coefficient of subproblem respectively.
        (ii)  If solution is found or all the nodes have been reached,
              stop and go to (10).
        (iii) If dead end is reached, backtrack to the latest node.
(9)           Go back to (3).
(10)          Return the feasibility of the problem and the feasible solution
              if it is satisfiable.
```

Table 5.1: The branch-and-bound algorithm for the singly-constrained polynomial CSP.

Iteration 2:

(1) We set up an occurrence table as below:

	x_1	x_2	x_3	x_4
+ve	2	3	2	1
-ve	3	3	4	2

(2) Since there is no variable that contains only positive or negative occurrence, we choose x_2 to be the branching variable and set it at 1.

(3) The constraint can be reduced as

(II): $-x_3 + 2x_4 - x_3x_4 - x_1x_3x_4 \geq 3$

(4) The *RHS* of the constraint is positive and the *LHS* value equals to -1. Since the upper bound and the lower bound of the subproblem (II) are 2 and -3 respectively, we continue branching.

Iteration 3:

(1) We set up an occurrence table as below:

	x_1	x_3	x_4
+ve	0	0	1
-ve	1	3	2

(2) Since x_1 and x_3 have negative occurrence only, we set both x_1 and x_3 at 0.

(3) The constraint can be reduced as

(III): $2x_4 \geq 3$

(4) The *RHS* of the constraint is positive and the *LHS* value equals to 2. Obviously, the problem is infeasible and so we backtrack at $x_2 = 0$.

Backtrack at $x_2 = 0$:

(1) The constraint can be formed as

(IV): $-2x_1 - 2x_3 + 2x_1x_3 - x_1x_3x_4 \geq 1$

(2) The *RHS* of the constraint is positive and the *LHS* value equals to -3. From here, we find that the upper bound and the lower bound of the subproblem (IV) are 2 and -5. We continue branching.

Iteration 4:

(1) We set up an occurrence table as below:

	x_1	x_3	x_4
+ve	1	1	0
-ve	2	2	1

(2) Since x_4 has the negative occurrence only, we set x_4 at 0.

(3) The constraint can be reduced as

(V): $-2x_1 - 2x_3 + 2x_1x_3 \geq 1$

(4) The *RHS* of the constraint is positive and the *LHS* value equals to -2. From here, we find that the upper bound and the lower bound of the subproblem (V) are 2 and -4. We continue branching.

Iteration 5:

(1) We set up an occurrence table as below:

	x_1	x_3
+ve	1	1
-ve	1	1

(2) Since there is no variable that contains only positive or negative occurrence, we choose x_1 to be the branching variable and set it at 1.

(3) The constraint can be reduced as

(VI): $0 \geq 3$

(4) Obviously, the problem is infeasible and so we backtrack at $x_1 = 0$.

Backtrack at $x_1 = 0$:

(1) The constraint can be formed as

(VII): $-2x_3 \geq 1$

(2) The *RHS* of the constraint is positive and the *LHS* value equals to -2. Obviously, the problem is infeasible and so we backtrack at $x_5 = 0$.

Backtrack at $x_5 = 0$:

(1) The constraint can be formed as

(VIII): $-x_1 - 2x_2 - x_3 - x_4 + x_1x_2 + 3x_1x_3 - x_1x_4 + 3x_2x_4 - x_3x_4 - 2x_1x_2x_3 - x_1x_3x_4 - x_2x_3x_4 \geq 1$

(2) Since the *RHS* of the constraint is positive and the *LHS* value equals to -4. From here, we find that the upper bound and the lower bound of the subproblem (VIII) are 7 and -11, respectively. We continue branching.

Iteration 6:

(1) We set up an occurrence table as below:

	x_1	x_2	x_3	x_4
+ve	2	2	1	1
-ve	4	3	5	5

(2) Since there is no variable that contains only positive or negative occurrence, we choose x_1 to be the branching variable and set it at 1.

(3) The constraint can be reduced as

(IX): $-x_2 + 2x_3 - 2x_4 - 2x_2x_3 + 3x_2x_4 - 2x_3x_4 - x_2x_3x_4 \geq 2$

(4) The *RHS* of the constraint is positive and the *LHS* value equals to -3. From here, we find that the upper bound and lower bound of the subproblem (IX) are 5 and -8,

respectively. We continue branching.

Iteration 7:

(1) We set up an occurrence table as below:

	x_2	x_3	x_4
+ve	1	1	1
-ve	3	3	3

(2) Since there is no variable that contains only positive or negative occurrence, we choose x_2 to be the branching variable and set it at 1.

(3) The constraint can be reduced as

(X): $x_4 - 3x_3x_4 \geq 3$

(4) The *RHS* of the constraint is positive and the *LHS* value equals to -2. From here, we find that the upper bound and lower bound of the subproblem (X) are 1 and -3, respectively. We continue branching.

Iteration 8:

(1) We set up an occurrence table as below:

	x_3	x_4
+ve	0	1
-ve	1	1

(2) Since x_3 has negative occurrence only, we set x_3 at 0.

(3) The constraint can be reduced as

(XI): $x_4 \geq 3$

(4) The *RHS* of the constraint is positive and the *LHS* value equals to 1. Obviously, the problem is infeasible and so we backtrack at $x_2 = 0$.

Backtrack at $x_2 = 0$:

(1) The constraint can be formed as

(XII): $2x_3 - 2x_4 - 2x_3x_4 \geq 2$

(2) The *RHS* of the constraint is positive and the *LHS* value equals to -2. From here, we find that the upper bound and the lower bound of the subproblem (XII) are 2 and -4. We continue branching.

Iteration 9:

(1) We set up an occurrence table as below:

	x_3	x_4
+ve	1	0
-ve	1	2

(2) Since x_4 has the negative occurrence only, we set x_4 at 0.

(3) The constraint can be reduced as

(XIII): $2x_3 \geq 2$

(4) From here, we find that the *LHS* value of the subproblem (XIII) equals to the *RHS*. We set $x_3 = 1$.

(5) Since a feasible solution is found (i.e. $x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0, x_5 = 0$), we stop and report that the problem is satisfiable.

Conclusion

From the above example, we can see that our branch-and-bound algoithm is better than the conventional backtracking method. Although the number of iterations is reduced, we still have to branch to a deeper level for checking the satisfiability, especially for those infeasible branches. In the following chapters, we will discuss revised branch and bound rules so as to improve the performance of our branch-and-bound method.

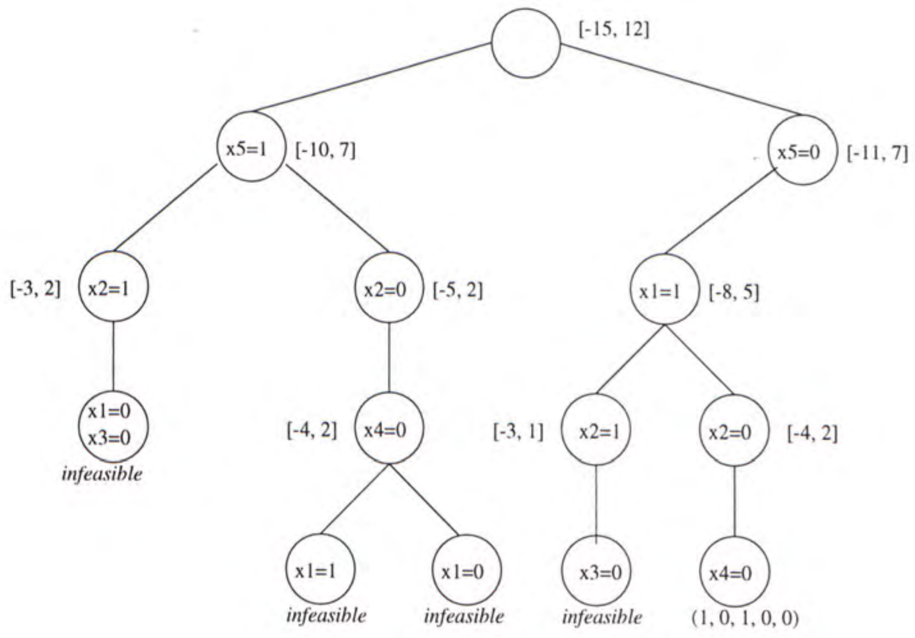

Figure 5.2: The solution tree of the problem $-x_1 - 2x_2 - x_3 - x_4 + x_1x_2 + 3x_1x_3 - x_1x_4 - x_1x_5 + 3x_2x_4 - x_3x_4 - x_3x_5 + x_4x_5 - 2x_1x_2x_3 - x_1x_3x_4 - x_2x_3x_4 + x_1x_2x_5 - x_1x_3x_5 + x_1x_4x_5 + x_2x_3x_5 - x_2x_4x_5 + x_3x_4x_5 \geq 1$ under our basic searching algorithm

Chapter 6

Revised Bound Rule for Branch-and-Bound Algorithm

In last chapter, we have discussed a basic branch-and-bound algorithm to solve our zero-one polynomial maximization problem. From the example in section 5.2.4, we find that the performance of the algorithm is not so good, that is, we have to search in a deeper level for infeasibility. In this chapter, we propose another bound rule that can produce a tighter bound for the subproblem. A linear programming problem is formed to figure out the upper bound of each subproblem and a powerful solver, *CPLEX* is used to solve the reduced linear programmming problem. We will end with an example in section 5.2.4 to see the performance of using this tighter-bound rule.

6.1 Revised Bound Rule

The procedure of the revised bound rule is described as below. In Chapter 3, we describe a method in solving zero-one optimization problems through its relaxation.

Bounding Rule		
(1)		Transform the subproblem into P4 and CP4 and solve the CP4 by using *CPLEX*.
(2)		If x is an integer solution, i.e. P4 and CP4 are consistent,
	(i)	if $Z_{CP4} \geq RHS_{subproblem}$, return x as the feasible solution of the original problem.
	(ii)	if $Z_{CP4} < RHS_{subproblem}$, the subproblem is infeasible and backtrack to the previous node.
(3)		If x is not an integer solution, i.e. P4 and CP4 are inconsistent,
	(i)	Z_{CP4} is the upper bound of the subproblem.
	(ii)	if $Z_{CP4} < RHS_{subproblem}$, then stop and return the subproblem as infeasible and backtrack to the previous node.
	(iii)	if $Z_{CP4} \geq RHS_{subproblem}$, we continue branching.

Table 6.1: Revised Bound Rule for the CSP

Now, we will apply a similar way to find out the upper bound or even the feasible integer solution of zero-one optimization problem. We first transform the subproblem into the form of P4 and CP4 as discussed in Chapter 3. Then, we can find out its solution by using a dual method. If x is an integer solution of the dual problem, CP4 and P4 are consistent. We can then check the satisfiability by checking whether the objective value, Z_{CP4}, is larger than or equal to the *RHS* of the subproblem. If so, we can stop and return the solution x^*, where x^* is the optimal solution of the subproblem. Thus, x^* is a feasible solution to the original problem. Otherwise, this subproblem is infeasible and we need to backtrack at the previous nodes.

If x is not an integer solution, CP4 and P4 are inconsistent. We can still find the upper bound (Z_{CP4}) of the subproblem. If Z_{CP4} is smaller than the *RHS* of the subproblem, the subproblem is infeasible and we backtrack to the latest node. Otherwise, we can continue branching. The revised bound rule is listed in Table 6.1.

6.1.1 *CPLEX*

In [5], it is suggested to use a Lagrangian dual method to solve the zero-one polynomial problem. Lagrangian dual method is a powerful solution scheme in integer programming problem. It reduces the primal zero-one polynomial problem into a linear programming problem which is easier to solve.

In this thesis, we use *CPLEX* as a solver for solving the linear relaxation of the subproblem. *CPLEX* is designed to solve linear programming problems using *Simplex method.* It can solve a linear programming problem by using the primal-simplex optimizer, the dual-simplex optimizer or the primal-dual barrier optimizer. If a linear program contains a substantial network, a speical network optimizer can be used. If the problem includes integer variables, a branch-and-bound method must be used. If the problem is a convex quadratic programming problem, the primal-dual barrier optimizer must be used. Many pratical problems can be solved faster by its *dual-simplex.*

On the other hand, *CPLEX* is callable in C or C++ language and so it can be embedded in any programme in C or C++. In this thesis, we use C language to implement the whole search procedure, including both branch and bound rules by calling *CPLEX* as a solver for CP4 formulation of the primal subproblem. In the following section, we will use the example in last chapter to test the performance of our revised bound rule.

6.2 Example

Refer to the example in section 5.2.4, the constraint is

$$-x_1 - 2x_2 - x_3 - x_4 + x_1x_2 + 3x_1x_3 - x_1x_4 - x_1x_5 + 3x_2x_4 - x_3x_4 - x_3x_5 + x_4x_5 -$$

$$2x_1x_2x_3 - x_1x_3x_4 - x_2x_3x_4 + x_1x_2x_5 - x_1x_3x_5 + x_1x_4x_5 + x_2x_3x_5 - x_2x_4x_5 + x_3x_4x_5 \geq 1$$

We do a similar procedure except that we would use the revised bound rule to find a solution and a tighter upper bound for the subproblem. Since the *RHS* of the constraint is positive and the *LHS* equals to 1, we set the upper and lower bound of the problem as 16 and -15, respectively. Before processing the search algorithm, we set up P4 and CP4 for the primal problem for checking the satisfiability of primal problem. P4: Max $-x_1 - 2x_2 - 3x_3 - 5x_4 - 3x_5 + x_1x_2 + 3x_1x_3 + \bar{x}_1x_4 + \bar{x}_1x_5 + 2\bar{x}_2x_3 + 3x_2x_4 + 3\bar{x}_3x_4 + 2\bar{x}_3x_5 + 2\bar{x}_1x_2x_3 + \bar{x}_1x_3x_4 + \bar{x}_2x_3x_4 + x_1x_2x_5 + \bar{x}_1x_3x_5 + x_1x_4x_5 + x_2x_3x_5 + \bar{x}_2x_4x_5 + x_3x_4x_5$

s.t.

$$\begin{array}{lll}
x_1 + \bar{x}_1 = 1 \quad x_2 + \bar{x}_2 = 1 & x_3 + \bar{x}_3 = 1 & \\
x_1x_2 \leq x_1 & x_1x_2 \leq x_2 & \\
x_1x_3 \leq x_1 & x_1x_3 \leq x_3 & \\
\bar{x}_1x_4 \leq \bar{x}_1 & \bar{x}_1x_4 \leq x_4 & \\
\bar{x}_1x_5 \leq \bar{x}_1 & \bar{x}_1x_5 \leq x_5 & \\
\bar{x}_2x_3 \leq \bar{x}_2 & \bar{x}_2x_3 \leq x_3 & \\
x_2x_4 \leq x_2 & x_2x_4 \leq x_4 & \\
\bar{x}_3x_4 \leq \bar{x}_3 & \bar{x}_3x_4 \leq x_4 & \\
\bar{x}_3x_5 \leq \bar{x}_3 & \bar{x}_3x_5 \leq x_5 & \\
\bar{x}_1x_2x_3 \leq \bar{x}_1 & \bar{x}_1x_2x_3 \leq x_2 & \bar{x}_1x_2x_3 \leq x_3 \\
\bar{x}_1x_3x_4 \leq \bar{x}_1 & \bar{x}_1x_3x_4 \leq x_3 & \bar{x}_1x_3x_4 \leq x_4 \\
\bar{x}_2x_3x_4 \leq \bar{x}_2 & \bar{x}_2x_3x_4 \leq x_3 & \bar{x}_2x_3x_4 \leq x_4 \\
x_1x_2x_5 \leq x_1 & x_1x_2x_5 \leq x_2 & x_1x_2x_5 \leq x_5 \\
\bar{x}_1x_3x_5 \leq \bar{x}_1 & \bar{x}_1x_3x_5 \leq x_3 & \bar{x}_1x_3x_5 \leq x_5 \\
x_1x_4x_5 \leq x_1 & x_1x_4x_5 \leq x_4 & x_1x_4x_5 \leq x_5 \\
x_2x_3x_5 \leq x_2 & x_2x_3x_5 \leq x_3 & x_2x_3x_5 \leq x_5 \\
\bar{x}_2x_4x_5 \leq \bar{x}_2 & \bar{x}_2x_4x_5 \leq x_4 & \bar{x}_2x_4x_5 \leq x_5 \\
x_3x_4x_5 \leq x_3 & x_3x_4x_5 \leq x_4 & x_3x_4x_5 \leq x_5 \\
x_1 \geq 0 \; x_2 \geq 0 \; x_3 \geq 0 \; x_4 \geq 0 \; x_5 \geq 0 & x_4 \leq 1 \; x_5 \leq 1 & \bar{x}_1 \geq 0 \; \bar{x}_2 \geq 0 \; \bar{x}_3 \geq 0 \\
x_1, x_2, x_3, x_4, x_5, \bar{x}_1, \bar{x}_2, \bar{x}_3 \text{ binary} & &
\end{array}$$

Let $t_1 = x_1x_2, t_2 = x_1x_3, w_3 = \bar{x}_1x_4, w_4 = \bar{x}_1x_5, w_5 = \bar{x}_2x_3, t_6 = x_2x_4, w_7 = \bar{x}_3x_4, w_8 = \bar{x}_3x_5, w_9 = \bar{x}_1x_2x_3, w_{10} = \bar{x}_1x_3x_4, w_{11} = \bar{x}_2x_3x_4, t_{12} = x_1x_2x_5, w_{13} = \bar{x}_1x_3x_5, t_{14} = x_1x_4x_5, t_{15} = x_2x_3x_5, w_{16} = \bar{x}_2x_4x_5, t_{17} = x_3x_4x_5$

CP4: Max $-x_1 - 2x_2 - 3x_3 - 5x_4 - 3x_5 + t_1 + 3t_2 + w_3 + w_4 + 2w_5 + 3t_6 + 3w_7 + 2w_8 + 2w_9 + w_{10} + w_{11} + t_{12} + w_{13} + t_{14} + t_{15} + w_{16} + t_{17}$

s.t.

$$\begin{array}{lll}
x_1 + \bar{x}_1 = 1 \quad x_2 + \bar{x}_2 = 1 & x_3 + \bar{x}_3 = 1 & \\
t_1 \le x_1 & t_1 \le x_2 & \\
t_2 \le x_1 & t_2 \le x_3 & \\
w_3 \le \bar{x}_1 & w_3 \le x_4 & \\
w_4 \le \bar{x}_1 & w_4 \le x_5 & \\
w_5 \le \bar{x}_2 & w_5 \le x_3 & \\
t_6 \le x_2 & t_6 \le x_4 & \\
w_7 \le \bar{x}_3 & w_7 \le x_4 & \\
w_8 \le \bar{x}_3 & w_8 \le x_5 & \\
w_9 \le \bar{x}_1 & w_9 \le x_2 & w_9 \le x_3 \\
w_{10} \le \bar{x}_1 & w_{10} \le x_3 & w_{10} \le x_4 \\
w_{11} \le \bar{x}_2 & w_{11} \le x_3 & w_{11} \le x_4 \\
t_{12} \le x_1 & t_{12} \le x_2 & t_{12} \le x_5 \\
w_{13} \le \bar{x}_1 & w_{13} \le x_3 & w_{13} \le x_5 \\
t_{14} \le x_1 & t_{14} \le x_2 & t_{14} \le x_5 \\
t_{15} \le x_2 & t_{15} \le x_3 & t_{15} \le x_5 \\
w_{16} \le \bar{x}_2 & w_{16} \le x_4 & w_{16} \le x_5 \\
t_{17} \le x_3 & t_{17} \le x_4 & t_{17} \le x_5 \\
x_1 \ge 0 \; x_2 \ge 0 \; x_3 \ge 0 \; x_4 \ge 0 \; x_5 \ge 0 & x_4 \le 1 \; x_5 \le 1 & \bar{x}_1 \ge 0 \; \bar{x}_2 \ge 0 \; \bar{x}_3 \ge 0
\end{array}$$

By using *CPLEX*, we find the solution of the relaxed problem is $x_1 = 0, x_2 = x_3 = x_4 = x_5 = 0.5$ with the objective value $Z_{CP4} = 3.5$. The upper bound of the primal problem becomes 3.5.

Iteration 1:

(1) We set up an occurrence table as below:

	x_1	x_2	x_3	x_4	x_5
+ve	4	4	3	4	5
-ve	6	4	7	6	4

(2) Since there is no variable that contains only positive or negative occurrence, we choose x_5 to be the branching variable and set it at 1.

(3) The constraint can be reduced as

(I): $-2x_1-2x_2-2x_3+2x_1x_2+2x_1x_3+x_2x_3+2x_2x_4-2x_1x_2x_3-x_1x_3x_4-x_2x_3x_4 \geq 1$

(4) Since the *RHS* of the constraint is positive and the *LHS* equals to -3, we try the revised bound rule to find out the solution of the subproblem and find out its upper bound.

(5) We set up the P4 and CP4 of the subproblem as:

P4: Max $-2x_1-2x_2-3x_3-2x_4+2x_1x_2+2x_1x_3+\bar{x}_2x_3+2x_2x_4+2\bar{x}_3x_4+2\bar{x}_1x_2x_3+\bar{x}_1x_3x_4+\bar{x}_2x_3x_4$

s.t.

$$\begin{array}{lll}
x_1+\bar{x}_1=1 \quad x_2+\bar{x}_2=1 & x_3+\bar{x}_3=1 & \\
x_1x_2 \leq x_1 & x_1x_2 \leq x_2 & \\
x_1x_3 \leq x_1 & x_1x_3 \leq x_3 & \\
\bar{x}_2x_3 \leq \bar{x}_2 & \bar{x}_2x_3 \leq x_3 & \\
x_2x_4 \leq x_2 & x_2x_4 \leq x_4 & \\
\bar{x}_3x_4 \leq \bar{x}_3 & \bar{x}_3x_4 \leq x_4 & \\
\bar{x}_1x_2x_3 \leq \bar{x}_1 & \bar{x}_1x_2x_3 \leq x_2 & \bar{x}_1x_2x_3 \leq x_3 \\
\bar{x}_1x_3x_4 \leq \bar{x}_1 & \bar{x}_1x_3x_4 \leq x_3 & \bar{x}_1x_3x_4 \leq x_4 \\
\bar{x}_2x_3x_4 \leq \bar{x}_2 & \bar{x}_2x_3x_4 \leq x_3 & \bar{x}_2x_3x_4 \leq x_4 \\
x_1 \geq 0 \ x_2 \geq 0 \ x_3 \geq 0 \ x_4 \geq 0 & x_4 \leq 1 & \bar{x}_1 \geq 0 \ \bar{x}_2 \geq 0 \ \bar{x}_3 \geq 0 \\
x_1, x_2, x_3, x_4, \bar{x}_1, \bar{x}_2, \bar{x}_3 \text{ binary} & &
\end{array}$$

Let $t_1 = x_1x_2, t_2 = x_1x_3, w_3 = \bar{x}_2x_3, t_4 = x_2x_4, w_5 = \bar{x}_3x_4, w_6 = \bar{x}_1x_2x_3, w_7 = \bar{x}_1x_3x_4, w_8 = \bar{x}_2x_3x_4$

CP4: Max $-2x_1-2x_2-3x_3-2x_4+2t_1+2t_2+w_3+2t_4+2w_5+2w_6+w_7+w_8$

s.t.

$$\begin{array}{lll}
x_1+\bar{x}_1=1 \quad x_2+\bar{x}_2=1 & x_3+\bar{x}_3=1 & \\
t_1 \leq x_1 & t_1 \leq x_2 & \\
t_2 \leq x_1 & t_2 \leq x_3 & \\
w_3 \leq \bar{x}_2 & w_3 \leq x_3 & \\
t_4 \leq x_2 & t_4 \leq x_4 & \\
w_5 \leq \bar{x}_3 & w_5 \leq x_4 & \\
w_6 \leq \bar{x}_1 & w_6 \leq x_2 & w_6 \leq x_3 \\
w_7 \leq \bar{x}_1 & w_7 \leq x_3 & w_7 \leq x_4 \\
w_8 \leq \bar{x}_2 & w_8 \leq x_3 & w_8 \leq x_4 \\
x_1 \geq 0 \ x_2 \geq 0 \ x_3 \geq 0 \ x_4 \geq 0 & x_4 \leq 1 & \bar{x}_1 \geq 0 \ \bar{x}_2 \geq 0 \ \bar{x}_3 \geq 0
\end{array}$$

(6) By using *CPLEX*, we find the solution of subproblem (I) is $x_1 = x_2 = x_3 = x_4 = 0.5$ with the objective value $Z_{CP4} = 2$

(7) Since $Z_{CP4} > RHS_{subproblem}$, we continue branching and set the upper bound as 2.

Iteration 2:

(1) We set up an occurrence table as below:

	x_1	x_2	x_3	x_4
+ve	2	3	2	1
-ve	3	3	4	2

(2) Since there is no variable that contains only positive occurrence or negative occurrence, we choose x_2 to be the branching variable and set x_2 at 1.

(3) The constraint can be reduced as

(II): $-x_3 + 2x_4 - x_3x_4 - x_1x_3x_4 \geq 3$

(4) Since the *RHS* of the constraint is positive and the *LHS* equals to -1, we try the revised bound rules to find out the solution of the subproblem and its upper bound.

(5) We set up the P4 and CP4 of the subproblem as:

P4: Max $-x_3 + 2\bar{x}_3x_4 + \bar{x}_1x_3x_4$

s.t.

$$\begin{array}{lll} x_1 + \bar{x}_1 = 1 & x_3 + \bar{x}_3 = 1 & \\ \bar{x}_3x_4 \leq \bar{x}_3 & \bar{x}_3x_4 \leq x_4 & \\ \bar{x}_1x_3x_4 \leq \bar{x}_1 & \bar{x}_1x_3x_4 \leq x_3 & \bar{x}_1x_3x_4 \leq x_4 \\ x_1 \geq 0\ x_3 \geq 0\ x_4 \geq 0 & x_4 \leq 1 & \bar{x}_1 \geq 0\ \bar{x}_3 \geq 0 \\ x_1, x_3, x_4, \bar{x}_1, \bar{x}_3 \text{ binary} & & \end{array}$$

Let $w_1 = \bar{x}_3x_4, w_2 = \bar{x}_1x_3x_4$

CP4: Max $-x_3 + 2w_1 + w_2$

s.t.

$$\begin{array}{lll} x_1 + \bar{x}_1 = 1 & x_3 + \bar{x}_3 = 1 & \\ w_1 \leq \bar{x}_3 & w_1 \leq x_4 & \\ w_2 \leq \bar{x}_1 & w_2 \leq x_3 & w_2 \leq x_4 \\ x_1 \geq 0\ x_3 \geq 0\ x_4 \geq 0 & x_4 \leq 1 & \bar{x}_1 \geq 0\ \bar{x}_3 \geq 0 \end{array}$$

(6) By using *CPLEX*, we find the solution of subproblem (II) is $x_1 = x_3 = 0, x_4 = 1$ with the objective value $Z_{CP4} = 2$. Since $Z_{CP4} < RHS_{subproblem}$, the problem is infeasible and so we backtrack at $x_2 = 0$.

Backtrack at $x_2 = 0$:

(1) The constraint can be formed as

(III): $-2x_1 - 2x_3 + 2x_1x_3 - x_1x_3x_4 \geq 1$

(2) Since the *RHS* of the constraint is positive and the *LHS* equals to -3, we check with our bound rule.

(3) We set up the P4 and CP4 of the subproblem as:

P4: Max $-2x_1 - 2x_3 - x_4 + 2x_1x_3 + \bar{x}_3x_4 + \bar{x}_1x_3x_4$

s.t.

$$\begin{array}{lll} x_1 + \bar{x}_1 = 1 & x_3 + \bar{x}_3 = 1 & \\ x_1x_3 \leq x_1 & x_1x_3 \leq x_3 & \\ \bar{x}_3x_4 \leq \bar{x}_3 & \bar{x}_3x_4 \leq x_4 & \\ \bar{x}_1x_3x_4 \leq \bar{x}_1 & \bar{x}_1x_3x_4 \leq x_3 & \bar{x}_1x_3x_4 \leq x_4 \\ x_1 \geq 0 \ x_3 \geq 0 \ x_4 \geq 0 & x_4 \leq 1 & \bar{x}_1 \geq 0 \ \bar{x}_3 \geq 0 \\ x_1, x_3, x_4, \bar{x}_1, \bar{x}_3 \text{ binary} & & \end{array}$$

Let $t_1 = x_1x_3, w_2 = \bar{x}_3x_4, w_3 = \bar{x}_1x_3x_4$

CP4: Max $-2x_1 - 2x_3 - x_4 + 2t_1 + w_2 + w_3$

s.t.

$$\begin{array}{lll} x_1 + \bar{x}_1 = 1 & x_3 + \bar{x}_3 = 1 & \\ t_1 \leq x_1 & t_1 \leq x_3 & \\ w_2 \leq \bar{x}_3 & w_2 \leq x_4 & \\ w_3 \leq \bar{x}_1 & w_3 \leq x_3 & w_3 \leq x_4 \\ x_1 \geq 0 \ x_3 \geq 0 \ x_4 \geq 0 & x_4 \leq 1 & \bar{x}_1 \geq 0 \ \bar{x}_3 \geq 0 \end{array}$$

(4) By using *CPLEX*, we find the solution of subproblem (III) is $x_1 = x_3 = 0, x_4 = 1$ with the objective value $Z_{CP4} = 0$

(5) Since $Z_{CP4} < RHS_{subproblem}$, we backtrack at $x_5 = 0$.

Backtrack at $x_5 = 0$:

(1) The constraint can be formed as

(IV): $-x_1 - 2x_2 - x_3 - x_4 + x_1x_2 + 3x_1x_3 - x_1x_4 + 3x_2x_4 - x_3x_4 - 2x_1x_2x_3 - x_1x_3x_4 - x_2x_3x_4 \geq 1$

(2) The *RHS* of the constraint is positive and the *LHS* value equals to -4.

(3) We set up the P4 and CP4 of the subproblem as:

P4: Max $-x_1 - 2x_2 - 3x_3 - 5x_4 + x_1x_2 + 3x_1x_3 + \bar{x}_1x_4 + 2\bar{x}_2x_3 + 3x_2x_4 + 3\bar{x}_3x_4 + 2\bar{x}_1x_2x_3 + \bar{x}_1x_3x_4 + \bar{x}_2x_3x_4$

s.t.

$$\begin{array}{lll}
x_1 + \bar{x}_1 = 1 \quad x_2 + \bar{x}_2 = 1 & x_3 + \bar{x}_3 = 1 & \\
x_1x_2 \leq x_1 & x_1x_2 \leq x_2 & \\
x_1x_3 \leq x_1 & x_1x_3 \leq x_3 & \\
\bar{x}_1x_4 \leq \bar{x}_1 & \bar{x}_1x_4 \leq x_4 & \\
\bar{x}_2x_3 \leq \bar{x}_2 & \bar{x}_2x_3 \leq x_3 & \\
x_2x_4 \leq x_2 & x_2x_4 \leq x_4 & \\
\bar{x}_3x_4 \leq \bar{x}_3 & \bar{x}_3x_4 \leq x_4 & \\
\bar{x}_1x_2x_3 \leq \bar{x}_1 & \bar{x}_1x_2x_3 \leq x_2 & \bar{x}_1x_2x_3 \leq x_3 \\
\bar{x}_1x_3x_4 \leq \bar{x}_1 & \bar{x}_1x_3x_4 \leq x_3 & \bar{x}_1x_3x_4 \leq x_4 \\
\bar{x}_2x_3x_4 \leq \bar{x}_2 & \bar{x}_2x_3x_4 \leq x_3 & \bar{x}_2x_3x_4 \leq x_4 \\
x_1 \geq 0 \; x_2 \geq 0 \; x_3 \geq 0 \; x_4 \geq 0 & x_4 \leq 1 & \bar{x}_1 \geq 0 \; \bar{x}_2 \geq 0 \; \bar{x}_3 \geq 0 \\
x_1, x_2, x_3, x_4, \bar{x}_1, \bar{x}_2, \bar{x}_3 \text{ binary} & &
\end{array}$$

Let $t_1 = x_1x_2, t_2 = x_1x_3, w_3 = \bar{x}_1x_4, w_4 = \bar{x}_2x_3, t_5 = x_2x_4, w_6 = \bar{x}_3x_4, w_7 = \bar{x}_1x_2x_3, w_8 = \bar{x}_1x_3x_4, w_9 = \bar{x}_2x_3x_4$

CP4: Max $-x_1 - 2x_2 - 3x_3 - 5x_4 + t_1 + 3t_2 + w_3 + 2w_4 + 3t_5 + 3w_6 + 2w_7 + w_8 + w_9$

s.t.

$$\begin{array}{lll}
x_1 + \bar{x}_1 = 1 \quad x_2 + \bar{x}_2 = 1 & x_3 + \bar{x}_3 = 1 & \\
t_1 \le x_1 & t_1 \le x_2 & \\
t_2 \le x_1 & t_2 \le x_3 & \\
w_3 \le \bar{x}_1 & w_3 \le x_4 & \\
w_4 \le \bar{x}_2 & w_4 \le x_3 & \\
t_5 \le x_2 & t_5 \le x_4 & \\
w_6 \le \bar{x}_3 & w_6 \le x_4 & \\
w_7 \le \bar{x}_1 & w_7 \le x_2 & w_7 \le x_3 \\
w_8 \le \bar{x}_1 & w_8 \le x_3 & w_8 \le x_4 \\
w_9 \le \bar{x}_2 & w_9 \le x_3 & w_9 \le x_4 \\
x_1 \ge 0 \; x_2 \ge 0 \; x_3 \ge 0 \; x_4 \ge 0 & x_4 \le 1 & \bar{x}_1 \ge 0 \; \bar{x}_2 \ge 0 \; \bar{x}_3 \ge 0
\end{array}$$

(4) By using *CPLEX*, we find the solution of subproblem (IV) is $x_1 = x_2 = x_3 = x_4 = 0.5$ with the objective value $Z_{CP4} = 3$.

(5) Since $Z_{CP4} > RHS_{subproblem}$, we continue branching and set the upper bound as 3.

Iteration 3:

(1) We set up an occurrence table as below:

	x_1	x_2	x_3	x_4
+ve	2	2	1	1
-ve	4	3	5	5

(2) Since there is no variable that contains only positive or negative occurrence, we choose x_1 to be the branching variable and set it at 1.

(3) The constraint can be reduced as

(V): $-x_2 + 2x_3 - 2x_4 - 2x_2x_3 + 3x_2x_4 - 2x_3x_4 - x_2x_3x_4 \ge 2$

(4) Since the *RHS* of the constraint is positive and the *LHS* equals to -3, we check the bound rule.

(5) We set up the P4 and CP4 of the subproblem as:

P4: Max $-x_2 - 5x_4 + 2\bar{x}_2x_3 + 3x_2x_4 + 3\bar{x}_3x_4 + \bar{x}_2x_3x_4$

s.t.

$$\begin{array}{lll}
x_2 + \bar{x}_2 = 1 & x_3 + \bar{x}_3 = 1 & \\
\bar{x}_2 x_3 \leq \bar{x}_2 & \bar{x}_2 x_3 \leq x_3 & \\
x_2 x_4 \leq x_2 & x_2 x_4 \leq x_4 & \\
\bar{x}_3 x_4 \leq \bar{x}_3 & \bar{x}_3 x_4 \leq x_4 & \\
\bar{x}_2 x_3 x_4 \leq \bar{x}_2 & \bar{x}_2 x_3 x_4 \leq x_3 & \bar{x}_2 x_3 x_4 \leq x_4 \\
x_2 \geq 0 \; x_3 \geq 0 \; x_4 \geq 0 & x_4 \leq 1 & \bar{x}_2 \geq 0 \; \bar{x}_3 \geq 0 \\
x_2, x_3, x_4, \bar{x}_2, \bar{x}_3 \text{ binary} & &
\end{array}$$

Let $w_1 = \bar{x}_2 x_3, t_2 = x_2 x_4, w_3 = \bar{x}_3 x_4, w_4 = \bar{x}_2 x_3 x_4$

CP4: Max $-x_2 - 5x_4 + 2w_1 + 3t_2 + 3w_3 + w_4$

s.t.

$$\begin{array}{lll}
x_2 + \bar{x}_2 = 1 & x_3 + \bar{x}_3 = 1 & \\
w_1 \leq \bar{x}_2 & w_1 \leq x_3 & \\
t_2 \leq x_2 & t_2 \leq x_4 & \\
w_3 \leq \bar{x}_3 & w_3 \leq x_4 & \\
w_4 \leq \bar{x}_2 & w_4 \leq x_3 & w_4 \leq x_4 \\
x_2 \geq 0 \; x_3 \geq 0 \; x_4 \geq 0 & x_4 \leq 1 & \bar{x}_2 \geq 0 \; \bar{x}_3 \geq 0
\end{array}$$

(6) By using *CPLEX*, we find the solution of subproblem (V) is $x_2 = 0, x_3 = 0, x_5 = 1$ with the objective value $Z_{CP4} = 2$

(7) Since $Z_{CP4} = RHS_{subproblem}$, we stop the iterations and report a feasible solution of the original problem x_1=1, x_2=0, x_3=1, x_4=0, x_5=0.

6.3 Conclusion

From the above example, we know that the revised bound rule reduced many branches that may contain infeasible solutions. Also, we can prove the satisfiability problem in a shorter time. However, is there any further improvement in our branch-and-bound method? As we know, the shape of the searching tree is highly related to the order of the branching variable list. Thus, we will propose a revised branch rule in the next chapter to see whether we can have any further improvement.

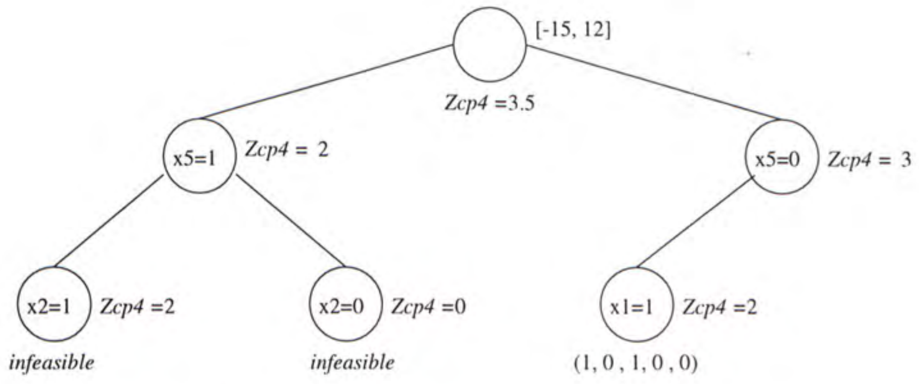

Figure 6.1: The solution tree for the revised bound rule of the problem $-x_1 - 2x_2 - x_3 - x_4 + x_1x_2 + 3x_1x_3 - x_1x_4 - x_1x_5 + 3x_2x_4 - x_3x_4 - x_3x_5 + x_4x_5 - 2x_1x_2x_3 - x_1x_3x_4 - x_2x_3x_4 + x_1x_2x_5 - x_1x_3x_5 + x_1x_4x_5 + x_2x_3x_5 - x_2x_4x_5 + x_3x_4x_5 \geq 1$ under our searching algorithm

Chapter 7

Revised Branch Rule for Branch-and-Bound Algorithm

In Chapters 5 and 6, we have developed a basic branch-and-bound algorithm and a revised bound rule. Although these two methods improve the traditional branch-and-bound method, their running time is still slow compared with other SAT searching methods. In this chapter, we will explain our revised branch rule and compare it with those we discussed in last two chapters. Moreover, we will test the performance of the revised branch rule combined with the revised bound rule in last chapter.

7.1 Revised Branch Rule

In Chapter 5, we define a branch rule by considering the positive occurrence of the variables. This method may be improper because the elimination of a variable would affect other variables in both the positive and negative terms. It can be improved by considering both the positive and negative occurrences.

From [15], Chu-Min Li suggested to use Freeman's method [6] to set the branching variable by considering the weight of the variables. The weight can be evaluated by the following formula :

$$\boxed{H(x) = w(x) * w(\bar{x}) * 1024 + w(x) + w(\bar{x})}$$

where $w(x)$ is the number of clauses reduced if we set x at 1 and $w(\bar{x})$ is the number of caluses reduced if we set x at 0. Obviously, the contribution is to give more importance to the product $w(x) * w(\bar{x})$ to balance the search tree. The value of 1024 may be used for a quick multiplication because $1024 = 2^{10}$.

In this thesis, we would modify this formula to:

$$\boxed{W(x) = [w(x) * w(\bar{x}) * n + w(x) + w(\bar{x})] * priority(x)}$$

where $w(x)$ is the number of positive occurrence of the variable, x, $w(\bar{x})$ is the number of negative occurrence of x and n is the number of variables in the problem. Initially, the $priority(x)$ is set to 1. After processing the CP4, we can find out whether x is integral. From the literature, the value branching variable should be non-integral from the relaxed problem. Here, we also prefer to branch on a non-integer variable. If x is an integer, we set $priority(x) = 0$. Otherwise, we set $priority(x) = 1$. If there are more than one variable that contain the same weight, we would randomly choose one of them. This randomization approach, instead of branching on the *best* variable, enables our algorithm to find a feasible solution quicker.

In addition, we have to decide whether to set the branching variable as 0 or 1. Let us consider the linear term of the branching variable. If it is positive, we set the branching variable at 1. Otherwise, we set it at 0. Fang and Loetamonphong [39] proved that the cost vectors c_i are effective in a constrained problem. In a minimization problem, if c_i is larger than zero, we should set x_i^* at its lower bound. Otherwise, we should set x_i^* as its upper bound. Therefore, the assignment of the branching

Revised Branch Rule		
(1)		Evaluate the weight of each variable x_i, $i = 1, \ldots, n$ by using the formula $W(x_i) = [w(x_i) * w(\bar{x}_i) * n + w(x_i) + w(\bar{x}_i)] * priority(x_i)$ where $priority(x_i) = 0$ if x_i is an integer in CP4 or $priority(x_i) = 1$ if x_i is not an integer in CP4.
(2)	(i)	Choose the variable, x_i, with the highest weight as the branching variable.
	(ii)	If more than one variable have the highest weight, choose the branching variable x_i randomly among these variables.
(3)	(i)	If the linear term of the branching variable is positive, eliminate it by setting $x_i = 1$.
	(ii)	Otherwise, set $x_i = 0$.

Table 7.1: Revised Branch Rule for the CSP

variables should be considered in the sign of the linear term in the constraint. The revised branch rule and the revised branch-and-bound algorithm is shown in Table 7.1 and Table 7.2.

7.2 Comparison between Branch Rule and Revised Branch Rule

In [15], Chu-Min Li proved that the width of a search tree is more important than its mean height. The width and the shape of the search tree are highly related to the order in eliminating variables. When n is small, the searching tree is shallow. That is, we can find the solution in a short running time no matter how the ordering is. However, if n becomes larger, the ordering would be more significant. The weight of the product $w(x) * w(\bar{x})$ would be heavier. For example, the positive occurrence and negative occurrence of x_1 are 3 and 5, respectively, and those of x_2 are 4 and 4, respectively. If n equals to 5, the weights of x_1 and x_2 are 83 and 88, respectively.

```
Revised Searching Algorithm for CSP
Procedure:
(1)   (i)   If the RHS of the constraint is non-positive, set the unassigned variable
            to be 0. Stop and go to (10).
      (ii)  If the sum of all the coefficient of the constraint is larger than the RHS,
            set the unassigned variable to be 1. Stop and go to (10).
(2)         Use the Revised Bound Rule to check whether the original problem is
            feasible and find its upper bound.
(3)         Calculate the number of occurrence of each variable in the constraint
            according to
      (i)   positive coefficient
      (ii)  negative coefficient
(4)   (i)   If the number of occurrence of variable in (3)(i) is 0, then we set that
            variable to 0.
      (ii)  If the number of occurrence of variable in (3)(ii) is 0, then we set that
            variable to 1.
      (iii) Go to (6).
(5)         Use the Revised Branch Rule to find the branching variable and eliminate
            this branching variable.
(6)         Update the constraint.
(7)   (i)   If the RHS of the constraint is non-positive, set the unassigned variable to
            be 0. Stop and go to (10).
      (ii)  If the sum of all the coefficient in LHS of the constraint is larger than
            the RHS, set the unassigned variable to be 1. Stop and go to (10).
(8)   (i)   Use the Revised Bound Rule to figure out the upper bound of the
            subproblem.
      (ii)  If a solution is found or all the nodes have been reached, stop and
            go to (10).
      (iii) If dead end is reached, backtrack to the latest node.
(9)         Go back to (3).
(10)        Return the feasibility of the problem and the feasible solution
            if it is satisfiable.
Revised Branch Rule
(1)         Evaluate the weight of each variable $x_i$, $i = 1, \ldots, n$ by using the formula
            $W(x_i) = [w(x_i) * w(\bar{x_i}) * n + w(x_i) + w(\bar{x_i})] * priority(x_i)$
            where $priority(x_i) = 0$ if $x_i$ is an integer in CP4 and
            $priority(x_i) = 1$ if $x_i$ is a non-integer.
(2)   (i)   Choose the variable, $x_i$, with the highest weight as the branching variable.
      (ii)  If more than one variables have the same highest weight, choose the
            branching variable $x_i$ randomly among these variables.
(3)   (i)   If the linear term of the branching variable is positive, eliminate it by
            setting $x_i = 1$.
      (ii)  Otherwise, set $x_i = 0$.
```

Revised Bound Rule		
(1)		Transform the subproblem into P4 and CP4 and solve the CP4 by using *CPLEX*.
(2)		If x is an integer solution, i.e. P4 and CP4 are consistent,
	(i)	if $Z_{CP4} \geq RHS_{subproblem}$, return x as the feasible solution of the original problem.
	(ii)	if $Z_{CP4} < RHS_{subproblem}$, the sub-problem is infeasible and backtrack to the previous node.
(3)		If x is not an integer solution, i.e. P4 and CP4 are inconsistent,
	(i)	Z_{CP4} is the upper bound of the subproblem.
	(ii)	if $Z_{CP4} < RHS_{subproblem}$, then stop and return the subproblem as infeasible and backtrack to the previous node.
	(iii)	if $Z_{CP4} \geq RHS_{subproblem}$, we continue branching.

Table 7.2: Revised Branch-and-Bound Algorithm for the CSP

However, if n equals 100, the difference between these two variables is larger (i.e. 1508 and 1608 respectively). Besides, when n becomes larger, the number of clauses generated would be larger. Thus, the number of occurrences (both positive and negative) would deviate from variables sharply. That means, we can find a better branching variable easier. On the contrary, if we only consider the weight of the positive occurrence of the variable, it would be ambiguous if more than one variable contain the same number of positive occurrence. A better branching variable can be found under our revised branch rule.

On the other hand, Chu-Min Li also proposed that branching on a variable randomly selected among the best variables may solve the problem faster. Our revised branch rule chooses one variable randomly among those variables with the same weight. This method is better than comparing the absolute coefficient among those variables. The elimination of the branching variable may affect other unassigned variables deeply in the last method. This changes the width of the search tree in the next level and the shape of the whole searching tree.

Neverthless, the revised branch rule considers whether to set the branching variable to 0 or 1 by considering the sign of its linear term. The shape of the search tree would be different and the width of the serach tree can be reduced. Thus, we can find out the feasible solution of the original problem in a shorter running time. According to the above reasons, the revised branch rule should have a better performance. We can test it by using the example discussed in last two chapters.

7.3 Example

Refer to the example in last two chapters, the constraint is

$$-x_1 - 2x_2 - x_3 - x_4 + x_1x_2 + 3x_1x_3 - x_1x_4 - x_1x_5 + 3x_2x_4 - x_3x_4 - x_3x_5 + x_4x_5 - 2x_1x_2x_3 - x_1x_3x_4 - x_2x_3x_4 + x_1x_2x_5 - x_1x_3x_5 + x_1x_4x_5 + x_2x_3x_5 - x_2x_4x_5 + x_3x_4x_5 \geq 1$$

In the following, we will use the revised branch and bound rule to show the improvement of these two methods. Since the *RHS* of the constraint is positive and the *LHS* equals to 1, we set the upper and lower bound as 16 and -15, respectively. Before processing the search procedure, we check the satisfiability of the problem by using *CPLEX* and set up the P4 and CP4 format of the problem as

P4: Max $-x_1 - 2x_2 - 3x_3 - 5x_4 - 3x_5 + x_1x_2 + 3x_1x_3 + \bar{x}_1x_4 + \bar{x}_1x_5 + 2\bar{x}_2x_3 + 3x_2x_4 + 3\bar{x}_3x_4 + 2\bar{x}_3x_5 + 2\bar{x}_1x_2x_3 + \bar{x}_1x_3x_4 + \bar{x}_2x_3x_4 + x_1x_2x_5 + \bar{x}_1x_3x_5 + x_1x_4x_5 + x_2x_3x_5 + \bar{x}_2x_4x_5 + x_3x_4x_5$

s.t.

$$\begin{array}{lll}
x_1+\bar{x}_1=1 \quad x_2+\bar{x}_2=1 & x_3+\bar{x}_3=1 & \\
x_1x_2 \leq x_1 & x_1x_2 \leq x_2 & \\
x_1x_3 \leq x_1 & x_1x_3 \leq x_3 & \\
\bar{x}_1x_4 \leq \bar{x}_1 & \bar{x}_1x_4 \leq x_4 & \\
\bar{x}_1x_5 \leq \bar{x}_1 & \bar{x}_1x_5 \leq x_5 & \\
\bar{x}_2x_3 \leq \bar{x}_2 & \bar{x}_2x_3 \leq x_3 & \\
x_2x_4 \leq x_2 & x_2x_4 \leq x_4 & \\
\bar{x}_3x_4 \leq \bar{x}_3 & \bar{x}_3x_4 \leq x_4 & \\
\bar{x}_3x_5 \leq \bar{x}_3 & \bar{x}_3x_5 \leq x_5 & \\
\bar{x}_1x_2x_3 \leq \bar{x}_1 & \bar{x}_1x_2x_3 \leq x_2 & \bar{x}_1x_2x_3 \leq x_3 \\
\bar{x}_1x_3x_4 \leq \bar{x}_1 & \bar{x}_1x_3x_4 \leq x_3 & \bar{x}_1x_3x_4 \leq x_4 \\
\bar{x}_2x_3x_4 \leq \bar{x}_2 & \bar{x}_2x_3x_4 \leq x_3 & \bar{x}_2x_3x_4 \leq x_4 \\
x_1x_2x_5 \leq x_1 & x_1x_2x_5 \leq x_2 & x_1x_2x_5 \leq x_5 \\
\bar{x}_1x_3x_5 \leq \bar{x}_1 & \bar{x}_1x_3x_5 \leq x_3 & \bar{x}_1x_3x_5 \leq x_5 \\
x_1x_4x_5 \leq x_1 & x_1x_4x_5 \leq x_4 & x_1x_4x_5 \leq x_5 \\
x_2x_3x_5 \leq x_2 & x_2x_3x_5 \leq x_3 & x_2x_3x_5 \leq x_5 \\
\bar{x}_2x_4x_5 \leq \bar{x}_2 & \bar{x}_2x_4x_5 \leq x_4 & \bar{x}_2x_4x_5 \leq x_5 \\
x_3x_4x_5 \leq x_3 & x_3x_4x_5 \leq x_4 & x_3x_4x_5 \leq x_5 \\
x_1 \geq 0 \; x_2 \geq 0 \; x_3 \geq 0 \; x_4 \geq 0 \; x_5 \geq 0 & x_4 \leq 1 \; x_5 \leq 1 & \bar{x}_1 \geq 0 \; \bar{x}_2 \geq 0 \; \bar{x}_3 \geq 0 \\
x_1, x_2, x_3, x_4, x_5, \bar{x}_1, \bar{x}_2, \bar{x}_3 \text{ binary} & &
\end{array}$$

Let $t_1 = x_1x_2, t_2 = x_1x_3, w_3 = \bar{x}_1x_4, w_4 = \bar{x}_1x_5, w_5 = \bar{x}_2x_3, t_6 = x_2x_4, w_7 = \bar{x}_3x_4, w_8 = \bar{x}_3x_5, w_9 = \bar{x}_1x_2x_3, w_{10} = \bar{x}_1x_3x_4, w_{11} = \bar{x}_2x_3x_4, t_{12} = x_1x_2x_5, w_{13} = \bar{x}_1x_3x_5, t_{14} = x_1x_4x_5, t_{15} = x_2x_3x_5, w_{16} = \bar{x}_2x_4x_5, t_{17} = x_3x_4x_5$

CP4: Max $-x_1 - 2x_2 - 3x_3 - 5x_4 - 3x_5 + t_1 + 3t_2 + w_3 + w_4 + 2w_5 + 3t_6 + 3w_7 + 2w_8 + 2w_9 + w_{10} + w_{11} + t_{12} + w_{13} + t_{14} + t_{15} + w_{16} + t_{17}$

s.t.

$$\begin{array}{lll}
x_1+\bar{x}_1=1 \quad x_2+\bar{x}_2=1 & x_3+\bar{x}_3=1 & \\
t_1 \le x_1 & t_1 \le x_2 & \\
t_2 \le x_1 & t_2 \le x_3 & \\
w_3 \le \bar{x}_1 & w_3 \le x_4 & \\
w_4 \le \bar{x}_1 & w_4 \le x_5 & \\
w_5 \le \bar{x}_2 & w_5 \le x_3 & \\
t_6 \le x_2 & t_6 \le x_4 & \\
w_7 \le \bar{x}_3 & w_7 \le x_4 & \\
w_8 \le \bar{x}_3 & w_8 \le x_5 & \\
w_9 \le \bar{x}_1 & w_9 \le x_2 & w_9 \le x_3 \\
w_{10} \le \bar{x}_1 & w_{10} \le x_3 & w_{10} \le x_4 \\
w_{11} \le \bar{x}_2 & w_{11} \le x_3 & w_{11} \le x_4 \\
t_{12} \le x_1 & t_{12} \le x_2 & t_{12} \le x_5 \\
w_{13} \le \bar{x}_1 & w_{13} \le x_3 & w_{13} \le x_5 \\
t_{14} \le x_1 & t_{14} \le x_2 & t_{14} \le x_5 \\
t_{15} \le x_2 & t_{15} \le x_3 & t_{15} \le x_5 \\
w_{16} \le \bar{x}_2 & w_{16} \le x_4 & w_{16} \le x_5 \\
t_{17} \le x_3 & t_{17} \le x_4 & t_{17} \le x_5 \\
x_1 \ge 0 \; x_2 \ge 0 \; x_3 \ge 0 \; x_4 \ge 0 \; x_5 \ge 0 & x_4 \le 1 \; x_5 \le 1 & \bar{x}_1 \ge 0 \; \bar{x}_2 \ge 0 \; \bar{x}_3 \ge 0
\end{array}$$

By using *CPLEX*, we find the solution of relaxed problem is $x_1 = 0, x_2 = x_3 = x_4 = x_5 = 0.5$ with the objective value $Z_{CP4} = 3.5$. The priorities of the variables are

	x_1	x_2	x_3	x_4	x_5
priority	0	1	1	1	1

Iteration 1:

(1) We set up an occurrence table as below:

	x_1	x_2	x_3	x_4	x_5
+ve	4	4	3	4	5
-ve	6	4	7	6	4
$W(x)$	0	88	115	130	109

(2) Since there is no variable that contains only positive occurrence or negative occurrence, we choose x_4 to be the branching variable. Also, we should set x_4 at 0 because the sign of its linear term is negative.

(3) The constraint is reduced to

(I): $-x_1-2x_2-x_3+x_1x_2+3x_1x_3-x_1x_5-x_3x_5-2x_1x_2x_3+x_1x_2x_5-x_1x_3x_5+x_2x_3x_5 \ge 1$

(4) Since the *RHS* of the constraint is positive and the *LHS* equals to -3, we try the revised bound rules to find out the solution of the subproblem and its upper bound.

(5) We set up the P4 and CP4 of the subproblem as:

P4: Max $-x_1 - 2x_2 - 3x_3 - 3x_5 + x_1x_2 + 3x_1x_3 + \bar{x}_1x_5 + 2\bar{x}_2x_3 + 2\bar{x}_3x_5 + 2\bar{x}_1x_2x_3 + x_1x_2x_5 + \bar{x}_1x_3x_5 + x_2x_3x_5$

s.t.

$$\begin{array}{lll}
x_1 + \bar{x}_1 = 1 & x_2 + \bar{x}_2 = 1 & x_3 + \bar{x}_3 = 1 \\
x_1x_2 \le x_1 & x_1x_2 \le x_2 & \\
x_1x_3 \le x_1 & x_1x_3 \le x_3 & \\
\bar{x}_1x_5 \le \bar{x}_1 & \bar{x}_1x_5 \le x_5 & \\
\bar{x}_2x_3 \le \bar{x}_2 & \bar{x}_2x_3 \le x_3 & \\
\bar{x}_3x_5 \le \bar{x}_3 & \bar{x}_3x_5 \le x_5 & \\
\bar{x}_1x_2x_3 \le \bar{x}_1 & \bar{x}_1x_2x_3 \le x_2 & \bar{x}_1x_2x_3 \le x_3 \\
x_1x_2x_5 \le x_1 & x_1x_2x_5 \le x_2 & x_1x_2x_5 \le x_5 \\
\bar{x}_1x_3x_5 \le \bar{x}_1 & \bar{x}_1x_3x_5 \le x_3 & \bar{x}_1x_3x_5 \le x_5 \\
x_2x_3x_5 \le x_2 & x_2x_3x_5 \le x_2 & x_2x_3x_5 \le x_5 \\
x_1 \ge 0\ x_2 \ge 0\ x_3 \ge 0\ x_5 \ge 0 & x_5 \le 1 & \bar{x}_1 \ge 0\ \bar{x}_2 \ge 0\ \bar{x}_3 \ge 0 \\
x_1, x_2, x_3, x_5, \bar{x}_1, \bar{x}_2, \bar{x}_3 \text{ binary} & &
\end{array}$$

Let $t_1 = x_1x_2, t_2 = x_1x_3, w_3 = \bar{x}_1x_5, w_4 = \bar{x}_2x_3, w_5 = \bar{x}_3x_5, w_6 = \bar{x}_1x_2x_3, t_7 = x_1x_2x_5, w_8 = \bar{x}_1x_3x_5, t_9 = x_2x_3x_5$

CP4: Max $-x_1 - 2x_2 - 3x_3 - 3x_5 + t_1 + 3t_2 + w_3 + 2w_4 + 2w_5 + 2w_6 + t_7 + w_8 + t_9$

s.t.

$$\begin{array}{lll}
x_1 + \bar{x}_1 = 1 & x_2 + \bar{x}_2 = 1 & x_3 + \bar{x}_3 = 1 \\
t_1 \le x_1 & t_1 \le x_2 & \\
t_2 \le x_1 & t_2 \le x_3 & \\
w_3 \le \bar{x}_1 & w_2 \le \bar{x}_5 & \\
w_4 \le \bar{x}_2 & w_4 \le x_3 & \\
w_5 \le \bar{x}_3 & w_5 \le x_5 & \\
w_6 \le \bar{x}_1 & w_6 \le x_2 & w_6 \le x_3 \\
t_7 \le x_1 & t_7 \le x_2 & t_7 \le x_5 \\
w_8 \le \bar{x}_1 & w_8 \le x_3 & w_8 \le x_5 \\
t_9 \le x_2 & t_9 \le x_3 & t_9 \le x_5 \\
x_1 \ge 0\ x_2 \ge 0\ x_3 \ge 0\ x_5 \ge 0 & x_5 \le 1 & \bar{x}_1 \ge 0\ \bar{x}_2 \ge 0\ \bar{x}_3 \ge 0
\end{array}$$

(6) By using *CPLEX*, we find the solution of subproblem (I) is $x_1 = x_2 = x_3 =$

$x_5 = 0.5$ with the objective value $Z_{CP4} = 2.5$. The priorities of the variables are

	x_1	x_2	x_3	x_5
priority	1	1	1	1

(7) Since $Z_{CP4} > RHS_{subproblem}$, we continue branching and set the upper bound as 2.5.

Iteration 2:

(1) We set up an occurrence table as below:

	x_1	x_2	x_3	x_5
+ve	3	3	2	2
-ve	4	2	4	3
$W(x)$	67	35	46	35

(2) Since there is no variable that contains only positive occurrence or negative occurrence, we choose x_1 to be the branching variable (x_1 has the largest weight). Also, we should set x_1 at 0 because the sign of its linear term is non-positive.

(3) The constraint can be reduced as

(II): $-2x_2 - x_3 - x_3x_5 + x_2x_3x_5 \geq 1$

(4) Since the *RHS* of the constraint is positive and the *LHS* equals to -3, we try the revised bound rules to find out the solution of the subproblem and its upper bound.

(5) We set up the P4 and CP4 of the subproblem as:

P4: Max $-2x_2 - x_3 - x_5 + \bar{x}_3x_5 + x_2x_3x_5$

s.t.

$$\begin{array}{lll} x_3 + \bar{x}_3 = 1 & & \\ \bar{x}_3x_5 \leq \bar{x}_3 & \bar{x}_3x_5 \leq x_5 & \\ x_2x_3x_5 \leq x_2 & x_2x_3x_5 \leq x_3 & x_2x_3x_5 \leq x_5 \\ x_2 \geq 0 \ x_3 \geq 0 \ x_5 \geq 0 & x_2 \leq 1 \ x_5 \leq 1 & \bar{x}_3 \geq 0 \\ x_2, x_3, x_5, \bar{x}_3 \text{ binary} & & \end{array}$$

Let $w_1 = \bar{x}_3x_5, t_2 = x_2x_3x_5$

CP4: Max $-2x_2 - x_3 - x_5 + w_1 + t_2$

s.t.

$$\begin{array}{lll} x_3 + \bar{x}_3 = 1 & & \\ w_1 \leq \bar{x}_3 & w_1 \leq x_5 & \\ t_2 \leq x_2 & t_2 \leq x_3 & t_2 \leq x_5 \\ x_2 \geq 0 \; x_3 \geq 0 \; x_5 \geq 0 & x_2 \leq 1 \; x_5 \leq 1 & \bar{x}_3 \geq 0 \end{array}$$

(6) By using *CPLEX*, we find the solution of subproblem (II) is $x_2 = x_3 = 0, x_5 = 1$ with the objective value $Z_{CP4} = 0$. Since $Z_{CP4} < RHS_{subproblem}$, the problem is infeasible and so we backtrack at $x_1 = 1$.

Backtrack at $x_1 = 1$:

(1) The constraint can be formed as

(III): $-x_2 + 2x_3 - x_5 - 2x_2x_3 + x_2x_5 - 2x_3x_5 + x_2x_3x_5 \geq 2$

(2) The *RHS* of the constraint is positive and the *LHS* value equals to -1 when all variables are set at 1.

(3) We set up the P4 and CP4 of the subproblem as:

P4: Max $-x_2 - 3x_5 + 2\bar{x}_2x_3 + x_2x_5 + 2\bar{x}_3x_5 + \bar{x}_2x_3x_5$

s.t.

$$\begin{array}{lll} x_2 + \bar{x}_2 = 1 & x_3 + \bar{x}_3 = 1 & \\ \bar{x}_2x_3 \leq \bar{x}_2 & \bar{x}_2x_3 \leq x_3 & \\ x_2x_5 \leq x_2 & x_2x_5 \leq x_5 & \\ \bar{x}_3x_5 \leq \bar{x}_3 & \bar{x}_3x_5 \leq x_5 & \\ \bar{x}_2x_3x_5 \leq \bar{x}_2 & \bar{x}_2x_3x_5 \leq x_3 & \bar{x}_2x_3x_5 \leq x_5 \\ x_2 \geq 0 \; x_3 \geq 0 \; x_5 \geq 0 & x_5 \leq 1 & \bar{x}_2 \geq 0 \; \bar{x}_3 \geq 0 \\ x_2, x_3, x_5, \bar{x}_2, \bar{x}_3 \text{ binary} & & \end{array}$$

Let $w_1 = \bar{x}_2x_3, t_2 = x_2x_5, w_3 = \bar{x}_3x_5, w_4 = \bar{x}_2x_3x_5$

CP4: Max $-x_2 - 3x_5 + 2w_1 + t_2 + 2w_3 + w_4$

s.t.

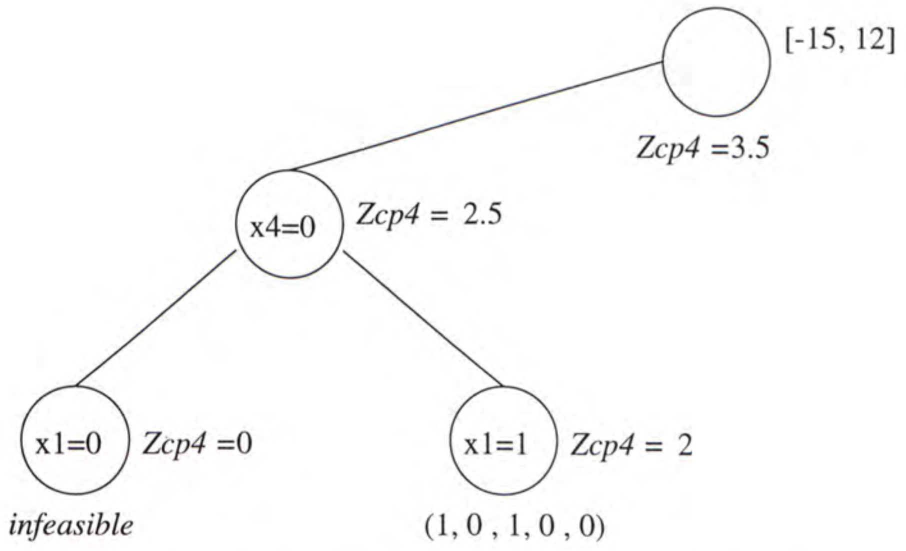

Figure 7.1: The solution tree for the revised branch-and-bound algorithm of the problem $-x_1-2x_2-x_3-x_4+x_1x_2+3x_1x_3-x_1x_4-x_1x_5+3x_2x_4-x_3x_4-x_3x_5+x_4x_5-2x_1x_2x_3-x_1x_3x_4-x_2x_3x_4+x_1x_2x_5-x_1x_3x_5+x_1x_4x_5+x_2x_3x_5-x_2x_4x_5+x_3x_4x_5 \geq 1$ under our searching algorithm

$$\begin{array}{lll}
x_2+\bar{x}_2=1 & x_3+\bar{x}_3=1 & \\
w_1 \leq \bar{x}_2 & w_1 \leq x_3 & \\
t_2 \leq x_2 & t_2 \leq x_5 & \\
w_3 \leq \bar{x}_3 & w_3 \leq x_5 & \\
w_4 \leq \bar{x}_2 & w_4 \leq x_3 & w_4 \leq x_5 \\
x_2 \geq 0 \; x_3 \geq 0 \; x_5 \geq 0 & x_5 \leq 1 & \bar{x}_2 \geq 0 \; \bar{x}_3 \geq 0
\end{array}$$

(4) By using *CPLEX*, we find the solution of subproblem (III) is $x_3 = 1, x_2 = x_5 = 0$ with the objective value $Z_{CP4} = 2$. Since $Z_{CP4} = RHS_{subproblem}$, we stop here and report the problem is satisfiable with the solution $(x_1 = x_3 = 1, x_2 = x_4 = x_5 = 0)$.

7.4 Conclusion

From the above example, we observe that the revised branch rule gets a great improvement from the original branch-and-bound-algorithm. It is significant that nine steps are saved compared with the basic branch-and-bound algorithm. In the next chapter, we will present experimental results and analysis from different sizes of problems. Moreover, we will compare our algorithm with other methods, like *SATZ*.

Chapter 8

Experimental Results and Analysis

In this chapter, experimental results for the three methods, the basic branch-and-bound method, the revised bound rule method and the revised branch-and-bound method, are presented in the first section. The statistics include the satisfiability of the problem, the mean and standard deviation of the computational time, the maximum number of backtracking and the maximum number of constraints generated in CP4. We discuss the performance of the three methods and compare them with *SATZ* under the same computing environment. We only present part of the results here and the complete results can be found in Appendix B. In the second section, we study the significance on the difference between our revised branch-and-bound method and *SATZ* and find out some reasons of outperformance of *SATZ*.

8.1 Experimental Results

Table 8.1, Table 8.2 and Table 8.3 list the mean and standard deviation of the computational time, the maximum number of backtracking and the maximum number of constraints generated in CP4 for n=5, n=10 and n=30 under 100 samples respectively. All methods are written in C language and the samples are run under Sun Workstation, Ultra-60. In Table 8.1, 76 samples are satisfiable while 24 samples are unsatisfiable. In Table 8.2, 74 samples are satisfiable while 26 samples are unsatisfiable. In Table 8.3, 73 samples are satisfiable while 27 samples are unsatisfiable. From the results, we can see that the performance of the three methods are in the same level when n is small. When n=5 and 10, the basic branch-and-bound algorithm runs in the shortest time although its maximum number of backtracking is much larger than that of the revised bound rule and the revised branch-and-bound algorithm. These two methods may take longer time in running the *CPLEX* for bounding parts.

However, when n is larger (i.e. n=30), the revised branch-and-bound algorithm does the best in both the mean computational time and the maximum number of backtracking. Obviously, the completion time becomes less for a smaller number of backtracking. This finding agrees with what we state in Chapter 5 that the efficiency of a searching procedure is highly related to the order of labeling. It means that the decision for choosing a branching variable is significant in branch-and-bound algorithm.

Table 8.4 gives the computational result of *SATZ* by using the same data we used for testing the basic branch-and-bound algorithm, the revised bound rule and the revised branch-and-bound algorithm. From the result, we find that the mean computational time of *SATZ* is less than one second. In Appendeix B, we can see that the running time of *SATZ* is within 2 seconds, which is 251 times faster than our revised branch-and-bound algorithm. In the next section, we will analyze the computational result of the four methods.

Method	Mean Completion Time (sec.)	Standard Deviation of Computational Time (sec.)	Maximum number of Backtracking	Maximum number of Constraints in CP4
Basic B&B	0.0001	0.001000	20	0
Revised Bound Rule	0.0137	0.011777	6	52
Revised B&B	0.0109	0.011110	7	52

Table 8.1: Results for the three methods when n=5

Method	Mean Completion Time (sec.)	Standard Deviation of Computational Time (sec.)	Maximum number of Backtracking	Maximum number of Constraints in CP4
Basic B&B	0.0143	0.012248	382	0
Revised Bound Rule	0.1046	0.062729	48	196
Revised B&B	0.0918	0.052134	31	196

Table 8.2: Results for the three methods when n=10

Method	Mean Completion Time (sec.)	Standard Deviation of Computational Time (sec.)	Maximum number of Backtracking	Maximum number of Constraints in CP4
Basic B&B	26315.6411	18579.064637	11772316	0
Revised Bound Rule	214.0988	148.253970	15006	729
Revised B&B	130.9823	111.793230	7372	729

Table 8.3: Results for the three methods when n=30

n	Mean Computational Time (sec.)	S.D. of Computaional Time (sec.)
5	0.0120	0.006963
10	0.0210	0.008933
30	0.5400	0.261437

Table 8.4: Results for *SATZ* in n=5, 10, 30

8.2 Statistical Analysis

In last section, we know that our revised branch-and-bound algorithm has a great improvement when compared to the basic branch-and-bound algorithm and the revised bound rule method. In section 8.2.1, we will discuss the performance of the above methods. Also from Table 8.4 and Appendix B, we know that *SATZ* implemented by Chu-Min Li runs much faster than our methods. We will talk over how it works in section 8.2.2.

8.2.1 Analysis of Search Techniques

From Appendix B, we can see that our basic branch-and-bound algorithm has the shortest running time when n is small (i.e. n=5, 10). The basic algorithm takes less than 0.01 seconds to complete all the samples. However, its number of backtracking is more than the revised bound rule method and the revised branch-and-bound algorithm. The maximum number of backtracking is 20 and 382 for n=5 and 10, respectively while they are 6 and 48 in the revised bound rule for n=5 and 10, respectively and they are 7 an 31 in the revised algorithm for n=5 and 10, respectively. Obviously, the latter two methods take time to find out the upper bound of the subproblems by using *CPLEX*. The computational time of these two methods may be larger than that of the basic algorithm although the number of backtracking of these two methods is smaller than that of the basic algorithm. However, when n is large (i.e. n=30), the order of variable labeling would be significant. As we state in Chapters 3 and 7, the choice of next branching variable may affect the efficiency for finding the feasible solution of the problem. A better branch rule is necessary for a large and hard problem.

Let us look at the results for the revised bound rule method and the revised branch-and-bound algorithm. Our revised branch-and-bound algorithm has an improvement especially when n becomes large. We can see that both two methods have the same degree of performance for n=5 from Appendix B. The revised bound rule method runs faster in 27 samples out of 100 while the revised branch-and-bound algorithm does 40. They have the same computational time in 33 samples. In that 27 samples, where the revised bound rule method does better, there are 3 samples that the revised branch-and-bound algorithm has smaller number of backtracking. The revised branch-and-bound algorithm may consume more time in computing the weight formula as we state in Chapter 7.

For n=10, the revised branch-and-bound algorithm does better than the revised bound rule method. There are 57 samples that our revised branch-and-bound algorithm has a shorter computational time while 31 samples that the revised bound rule method does better. In these 31 samples, where the revised bound rule method has a better performance, the number of backtracking of our revised branch-and-bound algorithm is smaller than that of the revised bound rule method. Evidently, our revised branch-and-bound algorithm produces a shorter enumeration tree during execution. On the other hand, we can see that the revised branch-and-bound algorithm performs much better for n=30. There are 83 samples that our revised algorithm has both a shorter computational time and a smaller number of backtracking, and there are 15 samples that the revised algorithm runs less than 10 seconds. In sample 50, 69, and 86, our revised algorithm runs at around 2 seconds when the revised bound rule takes more than 100 seconds. Clearly, the revised branch rule is powerful in general.

8.2.2 Discussion of the Performance of *SATZ*

The *SATZ* implemented by Chu-Min Li is a powerful solution scheme in SAT, especially when n is large. From Appendix B, we can see that *SATZ* takes at most 1.6 seconds for computing a sample with 30 variables. Why is *SATZ* so efficient especially when n is large? For any 3-SAT problem, every clause contains only three literals. That means, we can easily figure out whether the problem is feasible after several eliminations. Let's consider an example with 5 variables and 5 clauses.

$$\begin{array}{ll} x_1 \vee x_2 \vee x_5 & (1) \\ x_2 \vee x_3 \vee \bar{x}_4 & (2) \\ x_1 \vee \bar{x}_2 \vee \bar{x}_3 & (3) \\ \bar{x}_2 \vee x_4 \vee x_5 & (4) \\ \bar{x}_3 \vee \bar{x}_4 \vee \bar{x}_5 & (5) \end{array}$$

First, we evaluate the weight of each variable by using the formula

$$H(x) = w(x) * w(\bar{x}) * 1024 + w(x) + w(\bar{x})$$

and the weights are

variable	x_1	x_2	x_3	x_4	x_5
weight	2	4100	2051	2051	2051

Since x_1 has the only positive occurrence, we better set $x_1 = 1$ and the problem reduces as

$$\begin{array}{ll} \mathrm{T} & (1) \\ x_2 \vee x_3 \vee \bar{x}_4 & (2) \\ \mathrm{T} & (3) \\ \bar{x}_2 \vee x_4 \vee x_5 & (4) \\ \bar{x}_3 \vee \bar{x}_4 \vee \bar{x}_5 & (5) \end{array}$$

and the weights for the remaining variables are

variable	x_2	x_3	x_4	x_5
weight	1026	1026	2051	1026

We now set x_4=0 as the number of negative occurrence is larger and the problem becomes

$$\begin{array}{ll} \text{T} & (1) \\ \text{T} & (2) \\ \text{T} & (3) \\ \bar{x}_2 \vee F \vee x_5 & (4) \\ \text{T} & (5) \end{array}$$

Obviously, we choose x_5 or x_2 as our next branching variable so that (4) is satisfiable. We can set x_5=1 and so the problem becomes satisfiable.

$$\begin{array}{ll} \text{T} & (1) \\ \text{T} & (2) \\ \text{T} & (3) \\ \text{T} & (4) \\ \text{T} & (5) \end{array}$$

We can easily see that *SATZ* is very powerful for 3-SAT problem. If the number of variables, n, is large, i.e. the number of clauses is large, we can find out whether the subproblem is satisfiable after doing three to four times of eliminations. We backtrack earlier even the problem is hard.

Chapter 9

Concluding Remarks

We conclude the thesis by stating our contributions and possible directions for future research.

9.1 Conclusion

The contributions of our work can be summarized as follows. We derive a two-step transformation for converting any 3-CNF-SAT problem into a singly-constrained zero-one polynomial problem. With the help of this transformation, techniques in solving integer programming problem, such as branch-and-bound method, can be applied directly for finding the satisfiability of the original CNF-SAT problem. Based on the transformed singly-constrained zero-one polynomial problem, we propose our branch-and-bound algorithm in solving the SAT problem. Revised branch-and-bound rules are suggested to increase the efficiency.

In the literature, there are several transformations for converting CNF-SAT problem into an integer programming problem. These transformations usually produce extra variables and constraints that enlarge the storage size. Luckily, no additional variables and constraints are generated during our transformation and only a single surrogate constraint is formed at the end of the transformation. The storage space is smaller compared with other IP transformations. However, information may be lost in the proposed singly-constrained problem. For the 3-CNF-SAT problem, there are only 3 literals in each clause. Information gap exists between the two kinds of problems although they are equivalent.

Branch-and-bound method is suggested in solving the singly-constrained zero-one polynomial problem. Both revised bound and branch rules are proposed in Chapter 6 and 7, respectively. The basic branch-and-bound algorithm can solve a small-size problem in a short running time. It is ineffective for a large problem because of the loose bound and the weak branch rule. The solution tree becomes deeper when n is large and so the algorithm performs poorly. The revised bound rule provides a tighter upper bound for the subproblems so that backtracking can be carried out earlier. Furthermore, *CPLEX* is a strong solver in solving linear programming problems. It is used to find out the upper bound of the subproblems after relaxing. So, the computational time is shortened after using *CPLEX*. In addition, the weight formula in the revised branch rule can be used to obtain a better branching variable which balances the enumeration tree. Thus, the feasible solution can be figured out quickly. As a result, the revised branch-and-bound algorithm is more efficient.

9.2 Suggestions for Future Research

Our work represents a major step toward the understanding of satisfiability problem and integer programming problem. The two-step transformation provides a new way for tackling the satisfiablity problem. However, the resulting singly-constrained problem after the transformation may lose the structural property of the 3-CNF-SAT problem. The 3-CNF-SAT problem has a special structure: each clause contains exactly three literals. It is easy to figure out the satisfiability of the problem after several eliminations in general. The solution tree is short such that backtracking may occur earlier in every subproblem. On the other hand, the singly-constrained problem includes n variables in a single constraint and it may become inefficient when n is large. The enumeration tree is relatively large and backtracking may happen in a deeper level. Thus, a new approach is needed. Remaining m constraints after converting the 3-CNF clauses into integer programming format may be appliable since each new constraint contains at most three variables. This reformulated problem may be easier to solve.

Nevertheless, we can determine the next branching variable by using the idea in [5]. Considering the singly-constrained polynomial problem in Chapter 4, 5, 6 and 7[1], we can first evaluate the sum of all coefficient of the terms involving each variable and form the table

	x_1	x_2	x_3	x_4	x_5
Sum	-1	-1	-3	0	1

Then we choose the variable with the largest absolute sum, x_3, as the next branching variable. Next, we differentiate $f(x)$ with respect to x_3 and get

$$\frac{\partial f}{\partial x_3} = -1 + 3x_1 - x_4 - x_5 - 2x_1x_2 - x_1x_4 - x_2x_4 - x_1x_5 + x_2x_5 + x_4x_5$$

After converting it into CP4 format, we can use *CPLEX* to find out its upper and lower bounds which are 2 and -8, respectively. Since the absolute value of the lower bound is larger than the lower bound, we set $x_3 = 0$. The subproblem becomes

[1]The singly-constrained polynomial problem is $f(x) = -x_1 - 2x_2 - x_3 - x_4 + x_1x_2 + 3x_1x_3 - x_1x_4 - x_1x_5 + 3x_2x_4 - x_3x_4 - x_3x_5 + x_4x_5 - 2x_1x_2x_3 - x_1x_3x_4 - x_2x_3x_4 + x_1x_2x_5 - x_1x_3x_5 + x_1x_4x_5 + x_2x_3x_5 - x_2x_4x_5 + x_3x_4x_5 \geq 1$

$$f_1(x) = -x_1 - 2x_2 - x_4 + x_1x_2 - x_1x_4 - x_1x_5 + 3x_2x_4 + x_4x_5 + x_1x_2x_5 + x_1x_4x_5 - x_2x_4x_5 \geq 1$$

By using *CPLEX*, we find that the solution of the subproblem is $x_1 = x_2 = x_4 = x_5 = 0.5$ with $Z_{CP4} = 1.5$. Since it is feasible, we continue branching by setting up the table

	x_1	x_2	x_4	x_5
Sum	0	2	2	1

Now, we choose x_2 as the branching variable because x_2 has a term containing the largest absolute coefficient in $f_1(x)$. We differentiate $f_1(x)$ w.r.t. x_2 and get

$$\frac{\partial f_1(x)}{\partial x_2} = -2 + x_1 + 3x_4 + x_1x_5 - x_4x_5.$$

Since the upper and lower bounds are 2 and -3, respectively, we set $x_2 = 0$. We simplify the subproblem as

$$f_2(x) = -x_1 - x_4 - x_1x_4 + x_4x_5 + x_1x_4x_5 \geq 1$$

and find that Z_{CP4} of the objective function is 0. The subproblem is infeasible and we backtrack at $x_2 = 1$. The subproblem becomes

$$f_3(x) = 2x_4 - x_1x_4 + x_1x_4x_5 \geq 3$$

which is also infeasible because its $Z_{CP4} = 2$. So, we backtrack at $x_3 = 1$. By using *CPLEX*, we find that the problem is feasible with the solution (1,0,1,0,0). This approach helps us to get more information of how the branching variable effects in the objective function. Backtracking occurs earlier under this consideration.

Appendix A

Searching Procedures for Solving Constraint Satisfaction Problem (CSP)

The present appendix is devoted to the presentation of some tree search procedures in solving CSPs. These procedures include *generate and test*, *standard backtracking*, *forward checking* and *looking ahead*. For convenience, we restricted our scope in the area of binary CSPs.

A.1 Notation

Let $x_1, x_2, \ldots, x_n$ be the variables occurring in the binary CSP with their domains $D_1, D_2, \ldots, D_n$ on the binary values. Let $C_{ij}(x_i, x_j)$ be the constraint between the variables x_i and x_j where $i < j$. We say that values v_i and v_j for variables x_i and x_j are consistent if and only if $C_{ij}(v_i, v_j)$ is true for $i \neq j$. Also, a value v_i for x_i is said to be consistent with the values $v_1, \ldots, v_k$ for the variables $x_1, \ldots, x_k$ iff values v_i and

GENERATE AND TEST $P_k(v_1, \ldots, v_k)$ is true for all $k < n$ iff
1. $v_i \in D_i$ for $(1 \leq i \leq k)$.

Table A.1: Search Procedure for Generate and Test technique

v_j for variables x_i and x_j are consistent for $1 \leq j \leq k$.

A.2 Procedures for Solving CSP

In this section, we review 4 methods to solve the CSPs. In general, the searching procedure consists of testing whether $(v_1, \ldots, v_n)$ satisfies some property $P_n(v_1, \ldots, v_n)$, where $v_1, \ldots, v_n$ are the values of $x_1, \ldots, x_n$ and P_n holds if all the constraints are satisfied for this assignment [40].

A.2.1 Generate and Test

"Generate and Test" is the reverse of the constraint and generate methodology [10]. It is the simpliest way to find out the optimal solution of CSP. It first generates all the possible solutions and then tests them to see whether they satisfy all the constraints. The procedure is listed in Table A.1

However, "generate and test" is not an efficient way in solving CSP. All its constraints are only used to test whether the overall assignment is a solution or not. Actually, no pruning occurs in the search. Thus, it explores all the search space. On the other hand, the size of the enumeration tree is roughly proportional to the number of nodes in the tree. Therefore, "generate and test" is an inefficient search procedure with time complexity of 2^n where n is the number of variable in the problem.

STANDARD BACKTRACKING $P_k(v_1, \ldots, v_k)$ is true for all $k < n$ iff
1. $v_i \in D_i$ for $(1 \leq i \leq k)$.
2. for all i, j $(1 \leq i < j \leq k)$, $C_{ij}(v_i, v_j)$ is true.

Table A.2: Search Procedure for Standard Backtracking technique

A.2.2 Standard Backtracking

One of the simplest techniques for determining the satisfiablity of a CSP is backtracking. The idea is to choose a variable, and for each value of its domain, replacing the variable with that value in the constraints and determining the satisfiability of the constraints. This process repeats until a solution is found or all the variables are reached and then returns *true* or *false* indicating whether the constraints are satisfiable. Moreover, if a dead end is reached, it will return to the previous stage and check on the other side. The procedure is listed in Table A.2.

In standard backtracking, constraints are used backward to achieve a *posteriori* pruning. Given a sequence $< v_1, \ldots, v_k >$, the problem is to extend it by finding a value v_{k+1} for x_{k+1} in such a way that all the constraints involving x_{k+1} and a variable from $x_1, \ldots, x_k$ are satisfied. If there exists no such value, the sequence $< v_1, \ldots, v_k >$ cannot be extended and there is no need to search further in this part of the tree. This *posterior* pruning enables a drastic improvement in efficiency over "generate and test", which has to test all the assignments beginning by $< v_1, \ldots, v_k >$. However, if standard backtracking rediscovers the same fact continually, detects the failures and useless generation lately or backtracks to the first choice, it will suffer from a pathological behavior called *thrashing*.

FORWARD CHECKING $P_k(v_1, \ldots, v_k)$ is true for all $k < n$ iff
1. $v_i \in D_i$ for $(1 \leq i \leq k)$.
2. for all i, j $(1 \leq i < j \leq k)$, $C_{ij}(v_i, v_j)$ is true.
3. for all l $(k < l \leq n)$, there exists a value v_l in D_l such that $C_{1l}(v_1, v_l), \ldots, C_{kl}(v_k, v_l)$ are true.

Table A.3: Search Procedure for Forward Checking technique

A.2.3 Forward Checking

Forward checking is the easiest way to prevent future conflicts. It checks only the constraints between the current variable and the future variables. When a value is assigned to the current variable, any value in the domain of an unassigned variable, which conflicts with this assignment, is removed from the domain. The advantage of this is that if the domain of an unassigned variable becomes empty, it is known immediately that the current partial solution is inconsistent. Forward checking therefore allows branches of the search tree that will lead to failure to be pruned earlier than with simple backtracking. Note that whenever a new variable is considered, all its remaining values are guaranteed to be consistent with the past variables. So checking an assignment against the past assignments is no longer necessary.

The procedure of forward checking is listed in Table A.3. It overcomes many drawbacks in "generate and test" and backtracking. It spends more time in each node of the serach tree to reduce the number of nodes considered, the set of possible values for the not-yet-assigned variables and the number of constraints checks so as to achieve an overall improvement in performance.

LOOKING AHEAD $P_k(v_1, \ldots, v_k)$ is true for all $k < n$ iff
1. $v_i \in D_i$ for $(1 \leq i \leq k)$.
2. for all i, j $(1 \leq i < j \leq k)$ $C_{ij}(v_i, v_j)$ is true.
3. for all l $(k < l \leq n)$, there exists a value v_l in D_l such that $C_{1l}(v_1, v_l), \ldots, C_{kl}(v_k, v_l)$ are true.
4. for all l $(k < l \leq n)$, there exists a value v_l in D_l such that it is possible to find values $v_{k+1}, \ldots, v_{l-1}, v_{l+1}, \ldots, v_n$ in $D_{k+1}, \ldots, D_{l-1}, D_{l+1}, \ldots, D_n$ which satisfy $C_{k+1l}(v_{k+1}, v_l), \ldots, C_{l-1l}(v_{l-1}, v_l), C_{ll+1}(v_l, v_{l+1}), \ldots, C_{ln}(v_l, v_n)$.

Table A.4: Search Procedure for Looking Ahead technique

A.2.4 Looking Ahead

The techniques used in looking ahead is similiar to that of forward checking. In looking ahead, constraints are used even when more than one variable are left uninstantiated. It reduces the set of possible values that can be assigned to these variables. It prunes the variables earlier than that in forward checking. Therefore, the search size is reduced due to the smaller search space.

Looking ahead avoids much redundant work and also makes failures appear earlier in the search tree. It prevents bad backtracking points because of the directly propagated consequences of the choices to the unassigned variables. However, it is less incremental than forward checking. Indeed, looking ahead cannot remember and save most of the test results when checking the unassigned variables against the unassigned variables. Therefore, omitting these tests often results in a better efficiency.

Appendix B

Complete Results for Experiments

B.1 Complete Result for *SATZ*

B.1.1 $n = 5$

Sample no.	Completion time (sec.)		Sample no.	Completion time (sec.)
1	0.0100		16	0.0100
2	0.0200		17	0.0200
3	0.0100		18	0.0100
4	0.0000		19	0.0000
5	0.0200		20	0.0100
6	0.0000		21	0.0200
7	0.0100		22	0.0000
8	0.0100		23	0.0100
9	0.0200		24	0.0100
10	0.0100		25	0.0100
11	0.0200		26	0.0300
12	0.0100		27	0.0200
13	0.0200		28	0.0200
14	0.0000		29	0.0100
15	0.0200		30	0.0100

continued on next page

Sample no.	Completion time (sec.)		Sample no.	Completion time (sec.)
31	0.0200		66	0.0100
32	0.0200		67	0.0200
33	0.0100		68	0.0100
34	0.0200		69	0.0100
35	0.0200		70	0.0200
36	0.0200		71	0.0200
37	0.0100		72	0.0000
38	0.0100		73	0.0200
39	0.0200		74	0.0100
40	0.0200		75	0.0100
41	0.0100		76	0.0000
42	0.0000		77	0.0100
43	0.0100		78	0.0200
44	0.0000		79	0.0000
45	0.0000		80	0.0100
46	0.0100		81	0.0200
47	0.0000		82	0.0100
48	0.0100		83	0.0200
49	0.0200		84	0.0100
50	0.0200		85	0.0100
51	0.0000		86	0.0100
52	0.0100		87	0.0100
53	0.0100		88	0.0100
54	0.0100		89	0.0100
55	0.0000		90	0.0200
56	0.0100		91	0.0200
57	0.0200		92	0.0100
58	0.0100		93	0.0200
59	0.0000		94	0.0100
60	0.0100		95	0.0200
61	0.0100		96	0.0100
62	0.0200		97	0.0100
63	0.0100		98	0.0100
64	0.0100		99	0.0100
65	0.0200		100	0.0100

B.1.2 $n = 10$

Sample no.	Completion time (sec.)		Sample no.	Completion time (sec.)
1	0.0200		41	0.0100
2	0.0300		42	0.0300
3	0.0100		43	0.0300
4	0.0300		44	0.0300
5	0.0200		45	0.0200
6	0.0400		46	0.0200
7	0.0100		47	0.0200
8	0.0200		48	0.0200
9	0.0300		49	0.0200
10	0.0300		50	0.0200
11	0.0200		51	0.0300
12	0.0300		52	0.0100
13	0.0200		53	0.0300
14	0.0300		54	0.0300
15	0.0200		55	0.0100
16	0.0100		56	0.0100
17	0.0100		57	0.0300
18	0.0200		58	0.0300
19	0.0200		59	0.0200
20	0.0300		60	0.0300
21	0.0300		61	0.0100
22	0.0300		62	0.0200
23	0.0100		63	0.0100
24	0.0100		64	0.0300
25	0.0200		65	0.0300
26	0.0100		66	0.0100
27	0.0200		67	0.0100
28	0.0100		68	0.0300
29	0.0200		69	0.0100
30	0.0100		70	0.0200
31	0.0300		71	0.0300
32	0.0300		72	0.0200
33	0.0200		73	0.0200
34	0.0200		74	0.0300
35	0.0300		75	0.0300
36	0.0300		76	0.0300
37	0.0200		77	0.0200
38	0.0200		78	0.0200
39	0.0200		79	0.0200
40	0.0200		80	0.0200

continued on next page

Sample no.	Completion time (sec.)		Sample no.	Completion time (sec.)
81	0.0100		91	0.0200
82	0.0400		92	0.0200
83	0.0100		93	0.0000
84	0.0200		94	0.0300
85	0.0100		95	0.0000
86	0.0300		96	0.0000
87	0.0200		97	0.0100
88	0.0200		98	0.0200
89	0.0100		99	0.0400
90	0.0300		100	0.0300

B.1.3 $n = 30$

Sample no.	Completion time (sec.)		Sample no.	Completion time (sec.)
1	0.8900		26	0.7000
2	0.1000		27	0.8600
3	0.5800		28	0.6100
4	0.1500		29	0.7500
5	0.1100		30	0.5000
6	0.6500		31	0.4900
7	0.5000		32	1.0200
8	0.5700		33	0.7100
9	0.0700		34	0.7200
10	1.0300		35	0.5700
11	0.6000		36	0.6000
12	0.6700		37	0.4500
13	0.3000		38	0.3800
14	0.6600		39	0.2800
15	0.3400		40	0.4100
16	0.7900		41	0.3800
17	0.2700		42	0.7900
18	0.7000		43	0.5700
19	0.6000		44	0.6500
20	0.2900		45	0.3000
21	0.4400		46	0.7000
22	1.1200		47	0.4900
23	0.6900		48	0.7500
24	0.6400		49	0.4200
25	1.5800		50	0.7400

continued on next page

Sample no.	Completion time (sec.)
51	0.2500
52	0.7700
53	0.1300
54	0.4700
55	0.5600
56	0.8100
57	0.4400
58	0.1400
59	0.8500
60	0.2600
61	0.3100
62	0.3200
63	0.9100
64	0.6000
65	0.6500
66	0.4700
67	0.6300
68	0.0400
69	0.3400
70	0.4900
71	0.1400
72	0.2700
73	0.6100
74	0.3000
75	0.3700
76	0.3400
77	0.5500
78	0.4200
79	0.8000
80	0.3100
81	0.7900
82	0.5500
83	1.1100
84	0.4700
85	0.3600
86	0.7600
87	0.4200
88	0.2500
89	1.0100
90	0.3500
91	0.4800
92	0.5800
93	0.7500
94	0.4700
95	0.2500
96	0.5900
97	0.4000
98	0.8700
99	0.3200
100	0.4900

B.2 Complete Result for Basic Branch-and-Bound Algorithm

B.2.1 $n = 5$

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration
1	0.0000	12	9
2	0.0000	0	1
3	0.0000	3	6
4	0.0000	15	11
5	0.0000	9	10
6	0.0000	3	7
7	0.0000	2	0
8	0.0000	0	5
9	0.0000	12	10
10	0.0000	0	4
11	0.0000	13	10
12	0.0000	0	4
13	0.0000	17	13
14	0.0000	0	5
15	0.0000	3	6
16	0.0000	0	3
17	0.0000	9	10
18	0.0000	2	6
19	0.0000	4	5
20	0.0000	5	5
21	0.0000	11	12
22	0.0000	0	5
23	0.0000	7	8
24	0.0000	0	5
25	0.0000	18	14
26	0.0000	7	9
27	0.0000	0	4
28	0.0000	0	2
29	0.0000	0	5
30	0.0000	0	1

continued on next page

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration
31	0.0100	2	5
32	0.0000	7	9
33	0.0000	5	7
34	0.0000	2	6
35	0.0000	9	11
36	0.0000	7	9
37	0.0000	1	4
38	0.0000	12	10
39	0.0000	0	1
40	0.0000	7	7
41	0.0000	11	10
42	0.0000	2	5
43	0.0000	7	10
44	0.0000	17	13
45	0.0000	14	14
46	0.0000	15	12
47	0.0000	0	1
48	0.0000	5	6
49	0.0000	11	11
50	0.0000	20	15
51	0.0000	0	4
52	0.0000	7	8
53	0.0000	0	4
54	0.0000	14	12
55	0.0000	5	0
56	0.0000	9	10
57	0.0000	0	5
58	0.0000	0	4
59	0.0000	0	5
60	0.0000	7	8
61	0.0000	18	14
62	0.0000	8	10
63	0.0000	15	12
64	0.0000	0	1
65	0.0000	0	4
66	0.0000	13	10
67	0.0000	10	8
68	0.0000	9	9
69	0.0000	0	4
70	0.0000	14	11

continued on next page

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration
71	0.0000	5	7
72	0.0000	3	6
73	0.0000	0	2
74	0.0000	0	4
75	0.0000	16	13
76	0.0000	5	0
77	0.0000	17	13
78	0.0000	16	13
79	0.0000	0	4
80	0.0000	0	1
81	0.0000	0	1
82	0.0000	5	7
83	0.0000	0	3
84	0.0000	18	14
85	0.0000	0	5
86	0.0000	13	11
87	0.0000	9	10
88	0.0000	0	1
90	0.0000	0	2
91	0.0000	5	8
92	0.0000	14	11
93	0.0000	0	4
94	0.0000	13	10
95	0.0000	14	11
96	0.0000	0	1
97	0.0000	9	9
98	0.0000	3	4
99	0.0000	1	5
100	0.0000	0	5

B.2.2 $n = 10$

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration
1	0.0300	159	119
2	0.0300	261	193
3	0.0300	262	193
4	0.0000	12	19
5	0.0200	192	144
6	0.0400	382	269
7	0.0200	95	77
8	0.0000	78	62
9	0.0100	70	57
10	0.0100	96	77
11	0.0300	241	178
12	0.0000	0	1
13	0.0000	31	32
14	0.0400	278	201
15	0.0000	17	20
16	0.0000	0	9
17	0.0100	0	9
18	0.0200	148	112
19	0.0000	131	102
20	0.0000	13	17
21	0.0100	64	52
22	0.0100	87	67
23	0.0200	131	101
24	0.0100	78	63
25	0.0100	6	12
26	0.0000	20	23
27	0.0000	6	13
28	0.0000	64	55
29	0.0300	197	149
30	0.0200	139	110
31	0.0300	286	204
32	0.0300	279	202
33	0.0200	165	129
34	0.0200	177	130
35	0.0100	104	85

continued on next page

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration
36	0.0200	324	233
37	0.0300	190	138
38	0.0100	0	9
39	0.0100	75	59
40	0.0000	22	24
41	0.0000	0	6
42	0.0200	143	114
43	0.0000	67	56
44	0.0300	286	211
45	0.0100	337	244
46	0.0300	242	184
47	0.0000	2	11
48	0.0100	106	86
49	0.0100	23	24
50	0.0000	12	17
51	0.0300	242	177
52	0.0000	0	7
53	0.0100	80	69
54	0.0100	64	53
55	0.0300	249	182
56	0.0100	44	40
57	0.0000	39	37
58	0.0200	174	135
59	0.0200	129	102
60	0.0300	234	173
61	0.0000	11	14
62	0.0200	145	115
63	0.0300	229	167
64	0.0000	6	12
65	0.0000	58	48
66	0.0300	256	193
67	0.0100	204	149
68	0.0100	15	19
69	0.0100	6	12
70	0.0200	81	70

continued on next page

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration
71	0.0100	11	15
72	0.0300	237	174
73	0.0000	70	61
74	0.0100	82	69
75	0.0200	159	122
76	0.0100	93	74
77	0.0100	65	56
78	0.0100	189	141
79	0.0400	327	237
80	0.0000	0	7
81	0.0200	168	126
82	0.0100	33	31
83	0.0200	153	115
84	0.0100	68	56
85	0.0400	309	223
86	0.0300	271	194
87	0.0100	9	15
88	0.0000	86	72
89	0.0300	296	213
90	0.0400	293	210
91	0.0300	229	170
92	0.0100	77	65
93	0.0100	48	43
94	0.0100	13	18
95	0.0000	13	15
96	0.0000	25	27
97	0.0000	2	10
98	0.0000	169	128
99	0.0200	236	173
100	0.0200	132	101

B.2.3 $n = 30$

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration
1	10727.7600	1575732	1187777
2	187.7100	29071	22095
3	35452.8800	5529659	4170098
4	41224.5200	6083949	4602421
5	8862.9700	1304851	992397
6	45583.2600	6809587	5128830
7	35174.5200	5248507	3948288
8	41064.2800	4558643	3434278
9	21446.7100	3343211	2528336
10	34701.6200	5423585	4102313
11	2516.6900	391402	295066
12	14781.3300	2310952	1739584
13	75195.1100	11772316	8842614
14	8474.8100	1306260	992032
15	27262.4700	4206767	3184373
16	14899.1200	2319646	1745341
17	39267.3500	6096143	4600421
18	35008.0900	5399612	4101859
19	9025.4400	1405705	1060508
20	4156.0800	643662	485374
21	17920.8300	2785663	2102286
22	3050.7900	475756	358963
23	32000.9700	4982253	3754772
24	55037.6300	8567071	6413401
25	14405.6900	2243578	1687770
26	40577.5300	6321933	4744907
27	50001.5100	7509983	5629201
28	15279.7800	2364208	1792056
29	5692.4000	882469	665490
30	15178.2300	2355677	1785754

continued on next page

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration
31	46526.9000	6944312	5198023
32	67410.8300	9908148	7498961
33	7377.6600	1145283	861818
34	7775.9200	1202975	911410
35	35130.8000	5200526	3936922
36	394.9100	61430	46294
37	4413.1100	679639	515112
38	39731.7400	5890582	4445993
39	20492.2900	3192373	2410813
40	30370.6100	4728537	3564827
41	3782.6700	584415	442536
42	1316.1400	204228	153985
43	39621.9400	6209183	4653914
44	741.5300	116015	86858
45	31521.3400	4901398	3710782
46	5376.3600	797927	601290
47	52873.6900	7857952	5922740
48	35363.4200	5225284	3955174
49	32277.9100	4749665	3600649
50	31782.4100	4711417	3553183
51	12746.8500	1893829	1425197
52	31111.7000	4652442	3484544
53	22902.3500	3422493	2570778
54	11533.3800	1701968	1292010
55	50392.9400	7534113	5663159
56	72138.2400	10755194	8079082
57	10743.3800	1673983	1258206
58	39655.2000	5880949	4432966
59	46881.4300	6974524	5259028
60	11956.5500	1862328	1402868
61	34186.5700	5077899	3813711
62	62983.4900	9375250	7056047
63	19552.5700	2920613	2201361
64	50643.9700	7504727	5657906
65	42091.7400	6239241	4706913
66	55900.3400	8253388	6239150
67	32500.6000	4817626	3633354
68	18550.6300	2788543	2106355
69	29470.1600	4360551	3293909
70	2163.7300	330682	252449

continued on next page

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration
71	20794.9600	3212761	2429045
72	15114.7400	2335160	1771537
73	24082.7600	2635406	1993972
74	17830.5300	2635406	1993972
75	33901.3500	5008215	3799110
76	12649.1400	1934404	1469105
77	9.4400	1416	1097
78	42614.4200	6339793	4779648
79	23356.9000	3509844	2659156
80	25737.9800	3954266	2988154
81	140.8700	21474	16455
82	12319.5100	1811001	1370018
83	22642.7700	3360847	2534561
84	40221.2200	5976391	4514993
85	53090.2200	7988510	5972516
86	4976.2400	765347	575670
87	25635.3800	3797407	2866227
88	38169.6200	5662305	4267173
89	4654.2800	686394	515315
90	24856.5100	3666525	2763633
91	4476.8800	686394	515315
92	24885.4000	3668878	3668878
93	62419.2700	9326257	7020550
94	20146.6000	2984110	2246253
95	12878.4300	1910416	1441161
96	50457.4700	7500996	5644830
97	11909.3100	1773485	1335448
98	51638.7100	7743731	5810020
99	42899.1500	6360133	4801420
100	507.9700	78514	59072

B.3 Complete Result for Revised Bound Rule

B.3.1 $n = 5$

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration	Maximum number of Constraints generated in CP4
1	0.0100	4	4	19
2	0.0000	0	1	0
3	0.0000	1	3	22
4	0.0300	2	3	21
5	0.0200	3	6	25
6	0.0000	1	3	18
7	0.0000	0	2	0
8	0.0100	0	3	16
9	0.0100	4	4	17
10	0.0000	0	3	22
11	0.0100	1	2	14
12	0.0100	0	3	18
13	0.0100	4	4	24
14	0.0000	0	3	17
15	0.0100	1	3	22
16	0.0100	0	3	23
17	0.0100	2	4	21
18	0.0200	0	3	14
19	0.0000	1	2	12
20	0.0000	2	3	21
21	0.0100	2	4	24
22	0.0300	0	3	20
23	0.0100	1	3	16
24	0.0300	0	3	27
25	0.0000	4	4	21
26	0.0200	1	3	24
27	0.0000	0	3	19
28	0.0000	0	2	0
29	0.0000	0	3	24
30	0.0000	0	1	0

continued on next page

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration	Maximum number of Constraints generated in CP4
31	0.0000	1	3	16
32	0.0000	2	3	19
33	0.0200	0	2	18
34	0.0200	0	3	22
35	0.0400	2	4	24
36	0.0000	1	3	15
37	0.0000	0	3	21
38	0.0000	5	5	24
39	0.0000	0	1	0
40	0.0100	2	4	17
41	0.0200	3	4	22
42	0.0100	1	3	21
43	0.0100	2	4	23
44	0.0500	4	4	21
45	0.0300	3	4	22
46	0.0300	4	4	18
47	0.0000	0	1	0
48	0.0200	1	3	20
49	0.0100	2	3	21
50	0.0200	4	4	27
51	0.0000	0	3	24
52	0.0200	2	3	13
53	0.0000	0	2	14
54	0.0100	4	4	27
55	0.0200	0	4	19
56	0.0100	6	7	23
57	0.0100	0	3	19
58	0.0000	0	3	23
59	0.0000	0	3	27
60	0.0200	2	4	22
61	0.0000	2	3	18
62	0.0100	2	4	24
63	0.0200	3	3	19
64	0.0000	0	1	0
65	0.0000	0	2	15

continued on next page

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration	Maximum number of Constraints generated in CP4
66	0.0300	4	4	18
67	0.0100	1	2	16
68	0.0200	3	4	22
69	0.0200	0	4	24
70	0.0000	1	2	19
71	0.0300	2	3	19
72	0.0200	1	3	19
73	0.0000	0	2	0
74	0.0000	0	3	27
75	0.0200	2	3	21
76	0.0000	0	3	22
77	0.0200	4	4	24
78	0.0200	4	4	25
79	0.0100	0	3	24
80	0.0000	0	1	0
81	0.0000	0	1	0
82	0.0000	0	2	15
83	0.0000	0	3	22
84	0.0100	4	4	21
85	0.0100	0	3	25
86	0.0100	3	3	21
87	0.0000	2	5	16
88	0.0000	0	1	0
89	0.0000	0	1	0
90	0.0000	0	2	0
91	0.0300	1	3	21
92	0.0100	4	4	22
93	0.0200	0	3	18
94	0.0000	2	3	20
95	0.0000	3	3	16
96	0.0000	0	1	0
97	0.0300	3	5	21
98	0.0000	1	2	10
99	0.0100	0	3	22
100	0.0100	0	2	15

B.3.2 $n = 10$

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration	Maximum number of Constraints generated in CP4
1	0.0700	13	15	99
2	0.1800	25	19	131
3	0.2000	29	21	111
4	0.0300	1	5	101
5	0.0700	12	9	107
6	0.2400	31	22	136
7	0.0800	10	12	117
8	0.0600	6	8	120
9	0.0500	6	10	123
10	0.0500	5	7	101
11	0.1000	19	14	111
12	0.0000	0	1	0
13	0.1000	2	6	127
14	0.1700	25	18	120
15	0.0500	1	6	104
16	0.0400	0	7	123
17	0.0400	0	7	111
18	0.1300	15	15	115
19	0.0900	12	13	114
20	0.0400	1	6	128
21	0.0600	7	11	125
22	0.0500	5	9	120
23	0.0700	9	10	108
24	0.0800	3	9	110
25	0.0300	1	7	112
26	0.0300	1	5	109
27	0.0400	0	6	132
28	0.0300	2	7	110
29	0.1300	12	12	107
30	0.0900	9	11	108
31	0.1600	27	20	105
32	0.1600	28	20	116
33	0.0900	14	14	123
34	0.0700	18	13	102
35	0.1100	9	11	94

continued on next page

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration	Maximum number of Constraints generated in CP4
36	0.1900	27	20	124
37	0.1200	18	14	102
38	0.0400	0	5	111
39	0.0500	3	6	109
40	0.0300	1	7	102
41	0.0300	0	6	117
42	0.0900	9	13	114
43	0.0700	5	8	125
44	0.2000	33	24	130
45	0.2600	48	34	131
46	0.1200	15	14	109
47	0.0300	0	7	119
48	0.0900	9	11	101
49	0.0300	1	5	121
50	0.0300	0	5	106
51	0.1300	25	19	116
52	0.0500	0	6	100
53	0.1000	13	16	98
54	0.0700	5	9	101
55	0.1500	26	19	132
56	0.0200	2	6	79
57	0.0300	3	7	96
58	0.1200	14	14	128
59	0.0600	9	10	104
60	0.0100	15	11	108
61	0.0300	1	6	117
62	0.0400	0	5	117
63	0.0400	6	9	102
64	0.0300	0	6	127
65	0.0600	4	8	118
66	0.1600	20	20	128
67	0.1300	18	14	114
68	0.0400	1	6	144
69	0.0400	1	7	119
70	0.0700	6	8	103

continued on next page

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration	Maximum number of Constraints generated in CP4
71	0.0200	2	7	101
72	0.1500	22	16	109
73	0.0400	3	8	115
74	0.0600	3	5	97
75	0.1400	15	15	119
76	0.0900	6	9	96
77	0.0200	4	8	115
78	0.1100	19	15	119
79	0.1600	21	16	107
80	0.0300	0	7	107
81	0.1100	16	12	120
82	0.0200	1	5	113
83	0.1000	9	12	101
84	0.0500	6	10	119
85	0.1300	29	21	105
86	0.1400	22	16	106
87	0.0500	1	6	118
88	0.0600	17	17	113
89	0.1700	27	19	126
90	0.1700	25	18	128
91	0.1500	26	23	110
92	0.0700	5	9	124
93	0.0500	2	7	112
94	0.0600	2	8	126
95	0.0400	1	6	122
96	0.0500	2	8	125
97	0.0200	0	5	114
98	0.1300	15	14	117
99	0.1600	30	20	128
100	0.0700	8	10	119

B.3.3 $n = 30$

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration	Maximum number of Constraints generated in CP4
1	90.0100	1749	1240	595
2	3.7300	49	52	607
3	230.1500	4007	2763	647
4	268.0600	4749	3333	617
5	123.6100	2319	1648	583
6	508.2200	10206	7265	604
7	226.9500	4016	2823	617
8	261.4100	4608	3188	617
9	333.6000	6206	4439	599
10	327.3400	5954	4179	623
11	28.2700	531	389	588
12	34.1300	580	409	619
13	513.1500	9339	6486	660
14	94.7600	1706	1210	613
15	226.0000	4112	2912	602
16	111.1600	1977	1383	635
17	339.9800	6262	4254	623
18	417.9000	7790	5480	624
19	126.4600	2191	1600	622
20	60.4500	1078	773	612
21	162.0100	2870	2032	595
22	68.0400	1233	914	609
23	311.2900	5618	3886	608
24	311.5900	5898	4148	592
25	193.5600	3452	2364	619
26	237.1200	4214	2917	620
27	444.0900	8184	5711	642
28	111.4100	1927	1342	626
29	54.7700	960	695	617
30	200.5000	3592	2581	570
31	249.5000	4422	3049	611
32	458.1400	8462	5756	614
33	132.3700	2569	1870	628
34	53.3700	911	645	592
35	334.7700	6125	4280	609

continued on next page

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration	Maximum number of Constraints generated in CP4
36	7.1300	114	94	604
37	58.6000	1065	799	573
38	183.1600	3411	2422	611
39	145.6000	2546	1774	614
40	246.3700	4719	3260	574
41	39.8300	710	535	603
42	2.1400	18	23	633
43	194.5200	3434	2401	624
44	6.0300	92	77	579
45	291.2100	5384	3796	592
46	47.0600	839	608	615
47	367.5900	6162	4250	589
48	287.0800	5144	3559	599
49	239.8900	4239	2970	605
50	361.4600	6519	4603	568
51	63.6400	1079	782	582
52	154.8000	2626	1824	612
53	127.2200	2169	1511	607
54	157.8900	2734	1994	606
55	363.2600	6438	4430	576
56	441.8300	8103	5617	621
57	100.0300	1754	1242	632
58	440.1400	7960	5635	611
59	365.7900	6498	4509	637
60	52.9000	864	609	593
61	142.6200	2313	1645	611
62	379.3400	6630	4626	598
63	95.9400	1591	1115	606
64	332.2200	5574	3879	622
65	342.8200	5932	4146	605
66	542.6400	9627	6675	577
67	288.7900	4052	2871	620
68	157.6400	2695	1907	598
69	192.2600	3285	2297	603
70	28.7200	507	365	641

continued on next page

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration	Maximum number of Constraints generated in CP4
71	302.3400	5520	3960	581
72	93.5900	1516	1071	582
73	254.5100	4475	3171	623
74	223.8000	4023	2864	590
75	480.2500	8625	6118	629
76	89.9600	1576	1085	601
77	1.5900	2	15	639
78	360.6900	6440	4440	623
79	207.5600	3683	2643	606
80	257.8400	4665	3327	589
81	6.8600	106	101	611
82	139.3500	2380	1682	609
83	198.7400	3386	2426	643
84	284.0400	4963	3531	609
85	324.9600	5891	4069	619
86	30.7200	508	377	627
87	255.5600	4576	3175	611
88	441.6500	7602	5296	639
89	37.6600	620	449	613
90	218.3300	3952	2695	594
91	776.6000	15006	10699	599
92	303.4800	5637	3950	607
93	514.3600	8820	6091	644
94	142.3000	2479	1716	580
95	122.7300	2139	1553	565
96	264.8200	4680	3196	597
97	96.4300	1709	1228	578
98	435.3200	7888	5448	631
99	276.5000	4792	3288	609
100	8.8400	135	117	654

B.4 Complete Result for Revised Branch-and-Bound Algorithm

B.4.1 $n = 5$

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration	Maximum number of Constraints generated in CP4
1	0.0200	4	4	27
2	0.0000	0	1	0
3	0.0400	3	4	23
4	0.0100	3	3	21
5	0.0000	2	4	22
6	0.0000	0	3	16
7	0.0000	1	3	24
8	0.0000	0	3	18
9	0.0100	4	4	17
10	0.0200	0	3	19
11	0.0100	1	2	14
12	0.0100	0	1	18
13	0.0400	4	4	24
14	0.0000	0	3	22
15	0.0000	0	3	19
16	0.0000	0	4	23
17	0.0000	0	2	18
18	0.0200	0	2	9
19	0.0000	0	2	0
20	0.0000	2	3	21
21	0.0200	3	4	20
22	0.0100	2	3	20
23	0.0100	0	3	16
24	0.0100	0	3	25
25	0.0100	4	4	21
26	0.0100	0	3	19
27	0.0000	0	3	21
28	0.0000	0	2	0
29	0.0000	3	4	24
30	0.0000	0	1	0

continued on next page

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration	Maximum number of Constraints generated in CP4
31	0.0000	1	3	12
32	0.0200	2	3	19
33	0.0100	0	2	18
34	0.0300	2	3	21
35	0.0100	0	3	22
36	0.0000	0	3	19
37	0.0200	1	3	21
38	0.0200	2	3	15
39	0.0000	0	1	0
40	0.0300	0	3	17
41	0.0100	0	2	16
42	0.0100	0	3	17
43	0.0000	2	4	23
44	0.0200	4	4	18
45	0.0200	0	3	16
46	0.0100	4	4	18
47	0.0000	0	1	0
48	0.0100	0	3	20
49	0.0000	0	2	13
50	0.0000	4	4	23
51	0.0100	1	3	13
52	0.0000	0	3	13
53	0.0200	0	2	13
54	0.0400	3	3	19
55	0.0000	0	3	16
56	0.0000	0	3	18
57	0.0100	0	3	14
58	0.0000	1	3	20
59	0.0100	0	3	27
60	0.0200	5	6	21
61	0.0000	4	4	20
62	0.0100	1	3	25
63	0.0100	1	2	15
64	0.0000	0	1	0
65	0.0100	2	3	15

continued on next page

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration	Maximum number of Constraints generated in CP4
66	0.0100	4	4	18
67	0.0100	1	2	16
68	0.0300	2	4	18
69	0.0000	0	3	20
70	0.0100	1	2	19
71	0.0100	0	2	16
72	0.0000	0	3	21
73	0.0000	0	3	20
74	0.0000	1	3	27
75	0.0100	2	3	21
76	0.0000	0	3	22
77	0.0000	4	4	25
78	0.0300	4	4	25
79	0.0100	2	4	24
80	0.0000	0	1	0
81	0.0000	0	1	0
82	0.0000	0	2	15
83	0.0100	0	3	22
84	0.0100	4	4	21
85	0.0300	2	4	19
86	0.0300	2	3	16
87	0.0200	2	5	16
88	0.0000	0	1	0
89	0.0000	0	1	0
90	0.0000	0	2	0
91	0.0000	0	3	21
92	0.0100	2	3	19
93	0.0100	3	4	21
94	0.0300	4	4	25
95	0.0100	2	3	16
96	0.0000	0	1	0
97	0.0000	0	2	0
98	0.0000	1	2	10
99	0.0100	0	2	16
100	0.0100	0	2	15

B.4.2 $n = 10$

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration	Maximum number of Constraints generated in CP4
1	0.0300	1	6	99
2	0.1600	28	21	109
3	0.1400	21	16	111
4	0.0500	3	5	101
5	0.0800	14	11	99
6	0.2100	33	23	135
7	0.1400	16	16	115
8	0.0400	0	7	113
9	0.0600	5	8	123
10	0.0300	2	5	99
11	0.1200	14	11	109
12	0.0000	0	1	0
13	0.0200	1	6	102
14	0.1400	22	16	120
15	0.0300	0	4	104
16	0.0500	3	8	124
17	0.0300	0	9	108
18	0.0200	0	5	107
19	0.1200	8	10	114
20	0.0800	6	11	128
21	0.0200	0	4	117
22	0.1200	14	13	120
23	0.0400	1	5	104
24	0.0300	1	7	110
25	0.0300	1	5	107
26	0.2000	21	18	118
27	0.0400	0	6	122
28	0.0300	0	4	113
29	0.0800	3	7	97
30	0.0500	2	6	104
31	0.1500	26	20	105
32	0.1500	20	16	106
33	0.0300	0	5	112
34	0.0900	17	13	107
35	0.0200	1	7	89

continued on next page

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration	Maximum number of Constraints generated in CP4
36	0.1800	22	16	124
37	0.1100	15	11	102
38	0.0400	2	6	109
39	0.0700	11	13	109
40	0.0200	0	7	102
41	0.0800	10	11	118
42	0.0300	0	5	108
43	0.0200	0	6	125
44	0.1300	20	15	116
45	0.1000	14	11	105
46	0.0200	0	4	107
47	0.0400	2	6	117
48	0.0200	1	6	101
49	0.1900	22	18	120
50	0.0300	1	5	108
51	0.1400	22	16	116
52	0.0400	5	8	100
53	0.0100	0	5	92
54	0.0700	5	10	101
55	0.1400	22	17	122
56	0.0300	2	6	79
57	0.0100	0	5	88
58	0.1300	9	11	116
59	0.0300	0	4	102
60	0.0700	14	11	108
61	0.0000	0	2	0
62	0.0100	0	4	113
63	0.0300	8	5	98
64	0.0500	5	8	115
65	0.0300	0	6	118
66	0.0400	1	6	126
67	0.1200	18	13	111
68	0.1400	22	18	130
69	0.0900	6	9	108
70	0.0700	5	8	103

continued on next page

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration	Maximum number of Constraints generated in CP4
71	0.0300	0	5	112
72	0.1100	19	14	109
73	0.0300	0	6	115
74	0.0300	0	5	101
75	0.1300	15	15	119
76	0.0900	9	10	99
77	0.0900	5	9	123
78	0.0800	12	10	108
79	0.1500	23	18	107
80	0.0400	0	5	119
81	0.1100	15	12	120
82	0.0300	0	5	108
83	0.0600	3	6	101
84	0.0800	12	12	119
85	0.1700	24	18	109
86	0.0900	12	9	102
87	0.0700	5	8	120
88	0.1200	12	14	124
89	0.1300	18	14	121
90	0.1800	25	18	128
91	0.0600	2	6	108
92	0.1200	15	15	135
93	0.0200	0	5	112
94	0.0300	0	5	119
95	0.0400	1	8	122
96	0.0400	1	7	125
97	0.0600	7	9	114
98	0.1500	13	14	116
99	0.2100	31	22	128
100	0.0600	2	9	119

B.4.3 $n = 30$

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration	Maximum number of Constraints generated in CP4
1	108.2700	1868	1285	695
2	4.0700	47	54	676
3	121.2800	2081	1466	727
4	204.9700	3623	2521	715
5	129.3000	2256	1605	667
6	389.5900	7372	5226	698
7	14.3300	228	172	683
8	47.1000	816	570	599
9	22.1300	378	289	669
10	218.3100	3793	2626	695
11	5.1900	66	67	687
12	22.4800	359	265	695
13	335.8600	5692	3935	729
14	50.6400	905	661	691
15	27.2700	460	338	667
16	10.5000	154	126	720
17	283.0800	4934	3437	710
18	236.0600	4135	2909	705
19	99.7100	1797	1323	714
20	27.6100	465	353	684
21	3.1200	28	32	706
22	130.7300	2317	1647	678
23	121.1200	2106	1479	676
24	241.9400	4249	3032	670
25	45.9200	78	569	683
26	177.7400	3063	2143	687
27	83.6900	1483	1073	715
28	108.7400	1872	1316	706
29	71.4800	1197	858	695
30	151.9000	2695	1945	658
31	211.4300	3657	2556	698
32	373.0300	6530	4599	696
33	87.2300	1586	1126	718
34	48.3700	814	576	663
35	231.3300	3992	2869	696

continued on next page

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration	Maximum number of Constraints generated in CP4
36	20.8100	357	268	706
37	45.6000	820	594	664
38	45.1300	830	595	664
39	209.9000	3601	2525	680
40	134.8200	2315	1651	654
41	2.8500	26	34	697
42	4.3700	60	61	716
43	146.4600	2494	1739	687
44	1.5000	0	16	667
45	97.7300	1969	1197	679
46	14.8600	248	190	697
47	308.3700	5346	3724	665
48	247.2600	4526	3214	675
49	150.2200	2665	1902	691
50	1.3100	0	13	670
51	34.9300	635	484	669
52	191.1400	3415	2415	683
53	180.1400	3102	2167	680
54	9.1800	152	123	683
55	296.0600	5057	3543	670
56	396.6900	7030	4919	703
57	42.8600	715	516	699
58	325.0700	5794	4061	692
59	174.3200	3088	2136	703
60	177.0100	3073	2117	583
61	3.0500	28	35	688
62	143.5900	2514	1768	675
63	2.8800	26	35	693
64	241.4900	4231	2916	701
65	83.9200	1438	1030	688
66	17.5800	315	234	664
67	280.7000	4854	3435	694
68	31.0600	550	406	678
69	1.5600	0	18	672
70	8.0200	122	101	722

continued on next page

Sample no.	Completion time (sec.)	Number of backtrack	Number of iteration	Maximum number of Constraints generated in CP4
71	205.2400	3626	2535	653
72	103.4000	1863	1329	670
73	36.1700	613	457	673
74	176.7000	3181	2265	676
75	376.4900	6688	4677	688
76	178.0400	3099	2211	686
77	32.6000	576	415	711
78	270.0300	4739	3341	697
79	108.9600	1977	1468	676
80	220.7600	3841	2743	678
81	134.9900	2411	1731	703
82	39.6100	666	483	700
83	122.5500	2008	1441	643
84	244.7400	4177	2952	688
85	190.2400	3379	2355	706
86	1.6300	1	18	697
87	189.5100	3274	2304	681
88	120.0100	2041	1457	719
89	50.5500	883	628	695
90	47.1000	848	606	680
91	8.0100	141	108	580
92	175.8100	3095	2199	675
93	375.1000	6601	4621	724
94	8.5000	141	108	678
95	58.1000	997	715	670
96	209.0200	3540	2453	675
97	32.7100	571	419	659
98	359.8600	6265	4438	687
99	235.1300	4137	2880	700
100	268.7100	4678	3374	726

Bibliography

[1] J. H. A.V. Aho and J. Ullman, *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] S. Cook, "The complexity of theorem proving processing," *Proceeding of the Third Annual ACM Symposium, Theory of Computing*, pp. 151–158, 1971.

[3] J. N. E.M. Reingold and N. Deo, *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall, 1977.

[4] M. F. P. Bengt Aspvall and R. E. Tarjan, "A linear-time algorithm for testing the truth of certain quantified boolean formulas," *Information Processing Letters*, vol. 8, pp. 121–123, Mar. 1979.

[5] A. B. Warren E. ADAMS and A. SUTTER, "Unconstrained 0-1 optimization and lagrangean relaxation," *Discrete Applied Mathematics*, vol. 29, pp. 131–142, 1990.

[6] J. Freeman, *Improvements to Propositional Satisfiability Search Algorithms.* PhD thesis, University of Pennsylvania, Philadelphia, PA, 1995.

[7] S. Cook, "The complexity of theorem proving procedures," *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, no. 151-158, 1971.

[8] M. C. R. Rojas, "From quasi-solutions to solution: An evolutionary algorithm to solve csp."

[9] J. P. Walser, *Integer Optimization by Local Search : A Domain-Independent Approach*, vol. 1637 of *Lecture Notes in Artificial Intelligence.* Springer, 1999.

[10] K. Marriott and P. J. Stuckey, *Programming with Constraints: An Introduction.* The MIT Press, 1998.

[11] P. M. H. Barbara M. Smith, Sally C. Brailsford and H. P. Williams, "The progressive party problem: Integer linear programming and constraint programming compared,"

[12] E. Boros, "Maximum renamable horn sub-cnfs," *Discrete Applied Mathematics*, vol. 96-97, pp. 29–40, 1999.

[13] P. Heusch, "The complexity of the falsifiability problem for pure implicational formulas," *Discrete Applied Mathematics*, vol. 96-97, pp. 127–138, 1999.

[14] H. v. M. Joost P. Waners, "Recognition of tractable satisfiability problems through balanced polynomial representations," *Discrete Applied Mathematics*, vol. 99, pp. 229–244, 2000.

[15] C. M. Li, "A constraint-based approach to narrow search trees for satisfiability," *Information Processing Letters*, vol. 71, pp. 75–80, 1999.

[16] C. Li, "Heuristics based on unit propagation for satisfiability problem," *Proc. IJCAI-97*, pp. 336–371, Aug. 1997. Nagoya, Japan.

[17] A. I. S. Even and A. Shamir, "On the complexity of timetable and multi-commodity flow problems," *SIAM, Journal of Computing*, vol. 5, no. 4, pp. 691–703, 1976.

[18] T. Schaefer, "The complexity of satisfiability problems," *Proceedings in the Theth Annual ACM Symposium, Theory of Computer Science*, pp. 216–226, 1978.

[19] B. S. D. Mitchell and H. Levesque, "Hard and easy distributions of sat problems," *Proceedings of 10th National Conference on Artificial Intelligence*, pp. 459–465, 1992.

[20] J. Crawford and L. Auton, "Experimental results on the crossover point in random 3-sat," *Artificial Intelligence*, vol. 81, pp. 31–57, 1996.

[21] G. L. Martin Davis and D. Loveland, "A machine program for theorem-proving," *Communications of the ACM*, 1974.

[22] M. Davis and H. Putman, "A computing procedure for quantification theory," *Journal of ACM*, vol. 7, pp. 201–215, 1960.

[23] A. Monfroglio, "Connectionist networks for pivot selection in linear programming," *Neurocomputing*, vol. 8, pp. 51–78, 1995.

[24] H. Zimmermann and A. Monfroglio, "Linear programs for constraint satisfaction problems," *European Journal of Operational Research*, vol. 97, pp. 105–123, 1997.

[25] E. Klerk, "Semidefinite programming approaches for satisfiability," Aug. 2000. http://ssor.twi.tudelft.nl/ deklerk/.

[26] C. R. E. de Klerk, J. Peng, "A scaled gauss-newton primal-dual search direction for semidefinite optimization," 2000. http://ssor.twi.tudelft.nl/ deklerk/.

[27] H. v. M. Etenne de Klerk and J. P. Warners, "Relaxations of the satisfiability problem using semidefinite programming," 2000. http://ssor.twi.tudelft.nl/ deklerk/.

[28] E. Klerk, "Approximating the stability number of a graph via copositive and semidefinite programming," Sept. 2000. http://ssor.twi.tudelft.nl/ deklerk/.

[29] E. de Klerk and H. van Maaren, "On semidefinite programming relaxations of (2+p)-sat," Apr. 2000. http://ssor.twi.tudelft.nl/ deklerk/.

[30] S. Waluliewicz, *Integer Programming, Mathematics and Its Application*, vol. 46. Kluwer Academic Publishers, 1990.

[31] R. S. Garfinkel and G. L. Nemhauser, *Integer Programming.* John Wiley & Sons, 1972.

[32] H. A. Taha, *Integer Programming Theory, Applications, and Computations.* Academic Press, 1975.

[33] L. A. Wolsey, *Integer Programming.* John Wiley & Sons, Inc., 1998.

[34] T. Hu and P. Tucker, "Optimal alphabetic trees for binary search," *Information Processing Letters*, vol. 67, pp. 137–140, 1998.

[35] F. S. Hiller and G. J. Liberman, *Introduction to Operations Research.* McGraw-Hill, Inc., 7th edition ed., 2001.

[36] R. Gomory, *An algorithm for integer solutions to linear programming, in recent advances in mathematical programming.* McGraw-Hill, 1962.

[37] R. G. Parker and R. L. Rardin, *Discrete Optimization.* Academic Press, Inc, 1998.

[38] S. H. Lu and A. C. Williams, "Roof duality of 0-1 optimization," *Mathematical Programming*, vol. 37, pp. 357–360, 1987.

[39] S.-C. Fang and J. Loetamonphong, "Optimization of fuzzy relation equations with max-product composition," *Fuzzy Sets and Systems*, vol. 118, pp. 509–517, 2001.

[40] P. V. Hentenryck, *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, Mass., c1989.