A Solution Scheme of Satisfiability Problem by Active Usage of Totally Unimodularity Property

By

Mei LONG

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

in

Department of Systems Engineering And Engineering Management

June 2003

THE CHINESE UNIVERSITY OF HONG KONG
DEPARTMENT OF
THE SYSTEMS ENGINEERING AND ENGINEERING
MANAGEMENT

The undersigned hereby certify that they have read and recommend to the Faculty of Graduate Studies for acceptance a thesis entitled **"A Solution Scheme of Satisfiability Problem by Active Usage of Totally Unimodularity Property"** by **Mei LONG** in partial fulfillment of the requirements for the degree of **Master of Philosophy**.

Dated: June 2003

External Examiner: ______________________
Xiaoling Sun

Research Supervisor: ______________________
Duan LI

Examing Committee: ______________________
Leung May-Yee, Janny

____
Shu Zhong Zhang

THE CHINESE UNIVERSITY OF HONG KONG

Date: **June 2003**

Author: **Mei LONG**

Title: **A Solution Scheme of Satisfiability Problem by Active Usage of Totally Unimodularity Property**

Department: **The Systems Engineering and Engineering Management**

Degree: **M. Phil** Convocation: **July** Year: **2003**

Permission is herewith granted to The Chinese University of Hong Kong to circulate and to have copied for non-commercial purposes, at its discretion, the above title upon the request of individuals or institutions.

Signature of Author

To All I Love.

Table of Contents

Table of Contents v

Abstract viii

Acknowledgements x

1 Introduction 1

1.1 Satisfiability Problem 1
1.2 Motivation of the Research 1
1.3 Overview of the Thesis 2

2 Satisfiability Problem 4

2.1 Satisfiability Problem 5
2.1.1 Basic Definition 5
2.1.2 Phase Transitions 5
2.2 History 6
2.3 The Basic Search Algorithm 8
2.4 Some Improvements to the Basic Algorithm 9
2.4.1 Satz by Chu-Min Li 9
2.4.2 Heuristics and Local Search 12
2.4.3 Relaxation 13
2.5 Benchmarks 14
2.5.1 Specific Problems 14
2.5.2 Randomly Generated Problems 14
2.6 Software and Internet Information for SAT solving 16
2.6.1 Stochastic Local Search Algorithms (incomplete) 16
2.6.2 Systematic Search Algorithms (complete) 16
2.6.3 Some useful Links to SAT Related Sites 17

3 Integer Programming Formulation for Logic Problem **18**
3.1 SAT Problem 19
3.2 MAXSAT Problem 19
3.3 Logical Inference Problem 19
3.4 Weighted Exact Satisfiability Problem 20

4 Integer Programming Formulation for SAT Problem **22**
4.1 From 3-CNF SAT Clauses to Zero-One IP Constraints 22
4.2 Integer Programming Model for 3-SAT 23
4.3 The Equivalence of the SAT and the IP 23
4.4 Example 24

5 Integer Solvability of Linear Programs **27**
5.1 Unimodularity 27
5.2 Totally Unimodularity 28
5.3 Some Results on Recognition of Linear Solvability of IP 32

6 TU Based Matrix Research Results **33**
6.1 2×2 Matrix's TU Property 33
6.2 Extended Integer Programming Model for SAT 34
6.3 3×3 Matrix's TU Property 35

7 Totally Unimodularity Based Branching-and-Bound Algorithm **38**
7.1 Introduction 38
7.1.1 Enumeration Trees 39
7.1.2 The Concept of Branch and Bound 42
7.2 TU Based Branching Rule 43
7.2.1 How to sort variables based on 2×2 submatrices 43
7.2.2 How to sort the rest variables 45
7.3 TU Based Bounding Rule 46
7.4 TU Based Branch-and-Bound Algorithm 47
7.5 Example 49

8 Numerical Result **57**
8.1 Experimental Result 57
8.2 Statistical Results of *ILOG CPLEX* 59

9 Conclusions **61**
9.1 Contributions 61
9.2 Future Work 62

A The Coefficient Matrix A for Example in Chapter 7 64

B The Detailed Numerical Information of Solution Process for Example in Chapter 7 66

C Experimental Result 67
C.1 # of variables: 20, # of clauses: 91 67
C.2 # of variables: 50, # of clauses: 218 70
C.3 # of variables: 75, # of clauses: 325 73
C.4 # of variables: 100, # of clauses: 430 76

D Experimental Result of ILOG CPLEX 80
D.1 # of variables: 20, # of clauses: 91 80
D.2 # of variables: 50, # of clauses: 218 83
D.3 # of variables: 75, # of clauses: 325 86
D.4 # of variables: 100, # of clauses: 430 89

Bibliography 93

Abstract

Satisfiability problem is a well-known NP-complete problem. It consists of testing whether the clauses in a Conjunctive 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 literature, many transformations have been proposed in converting the satisfiability problem into an integer programming problem. These transformations usually create nonlinear integer programming problems that are very difficult to solve.

The aim of this work is to generate a novel simple equivalent linear integer programming model. This simple integer programming model is then solved by our suggested branch-and-bound linear relaxation programming algorithm. The order principle in the branch-and-bound method is derived from the Totally Unimodularity property of the constraint matrix. Computational results show that the proposed algorithm is very effective for both randomly generated 3-SAT problems and some hard 3-SAT problems reported in literature.

摘 要

可滿足性問題(Satisfiability problem) 是一個非常著名的NP完全問題。它是指檢驗是否存在一種對一組布林變量的賦值使得由若干個子句組成的合取範式的集合爲真。如果存在賦值滿足全部子句，此問題被稱爲可滿足的，否則，此問題是不可滿足的。每個子句只含有三個文字的隨機可滿足性問題(3-SAT) 是可滿足性問題中的最小的NP完全問題。關於把滿足性問題轉化爲整數規劃問題，文獻中多有提及。遺憾的是這些轉換通常因爲生成了非線性的整數規劃問題而使問題變得更加難以解決。

我們這項研究工作的目的是產生一個新穎簡單而與原問題等價的線性整數規劃問題。這個簡單的整數規劃模型可用我們提出的分支定界線性鬆弛演算法求解。在分之定界法中的變量排序原則是來自約束係數矩陣的單模性質。大量的計算結果表明，對於隨機產生的3-SAT問題和一些非常難的3-SAT問題，我們的演算法都是非常有效的。

Acknowledgements

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

I am also thankful to Dr. Jun WANG, for his guidance through the early time of chaos and confusion.

I would also like to thank my parents and my boyfriend, for providing so much tangible support and help.

Finally, I wish to thank the following:

Han ZHANG and Ruizhang HUANG(for their friendships);
Yongwei HUANG (for discussion of some mathematical problems);
Tong WANG, Li CHEN, Li ZHANG, ..., (for all the unforgettable memories we had together).
They enlight me and set me free from the sadness and anxiety. Their support and encouragement activate the motion of my research.

Chapter 1

Introduction

1.1 Satisfiability Problem

The propositional satisfiability problem (SAT) consists of finding a truth assignment that satisfies all the clauses in S (*satisfiable*) or showing that none exists (*unsatisfiable*).

SAT problem has been classified as the first NP-complete problem. If each clause exactly contains r literals, the problem is called an r-SAT problem. 2-SAT problem is solvable in polynomial time ([AU74, Coo71, RD77, AT79]), and 3-SAT is the smallest NP-complete subproblem of SAT with its computation time $O(2^n)$. If the ratio of the number of clauses to the number of variables is approximately equal to around 4.25 for a random 3-SAT problem, the problem is very difficult to solve.

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. Many transformations have been proposed in literature, but they usually create nonlinear integer programming models which may not be solved as easily as the original ones. Based on this consideration, a novel simple model that is equivalent to the original problem is proposed.

In the section of solving the proposed integer programming model, we focus on the Totally Unimodularity property of the constraint matrix. From the Totally Unimodularity property theory, if the constraint matrix is Totally Unimodular, it can be solved by its linear relaxation. In literature, researchers have discussed how to solve SAT if the constraint matrix is Totally Unimodular[CC95], and how to recognize the Totally Unimodular matrix[CCKV01]. However, the majority of SAT problems is not totally unimodular in its initial setting. Thus, we develop a branch-and-bound rule that can make the constraint matrix closer and closer to a totally unimodular one in the process of fixing variables one by one. In this way, the probability of solving the SAT problem by linear program relaxation will increase in the middle of the solution process.

The above consideration motivates us to develop a procedure to convert the conjunctive norm form (CNF) SAT problem into a novel simple equivalent integer programming problem, and then solve it by our proposed branch and bound algorithm.

1.3 Overview of the Thesis

This thesis is organized as follows. Chapter 2 gives a brief review of the satisfiability problem, its history, some typical and popular solution techniques, and some useful information on the internet. We mainly discuss the basic DPL search algorithm and three important improvements to the basic algorithm: Satz, heuristics and local search, and relaxations. Integer programming formulation and its continuous relaxation are important tools for this research. We provide some basic integer programming formulations for logic problems in Chapter 3 and for the SAT problem in Chapter 4, and explain the equivalence between the IP formulation and the original SAT problem. An example is given in Chapter 4 to illustrate the model conversion. Some classes of logic problems are solvable by linear programming. We introduce two types of them in Chapter 5: unimodularity and totally unimodularity. Totally unimodularity is the theoretical foundation of this thesis. Based on totally unimodularity (TU) theorems, some matrix research results are described in Chapter 6. In Chapter 7, we introduce our TU-based branch-and-bound algorithm in details. A simple example is given to illustrate this algorithm step by step. In order to test the efficiency of our algorithm, we perform large-scale computational experiment in Chapter 8 for some hard problems posted on SAT-related web-pages. Chapter 9 summarizes the research contributions and discusses possible future work.

Chapter 2

Satisfiability Problem

In this chapter I give the background of this thesis. Satisfiability (SAT) problem is the first NP-complete problem[Coo71]. SAT is also a footstone of computational complexity theory, and it is of commercial importance because of the great benefit from a highly efficient SAT solver for thousands of practical combinatorial problems. Its applications include graph coloring, Boolean N-queen induction, circuit diagnosis and scheduling problem [Roj, Wal99].

There are 6 sections in this chapter. The first section presents the definition of satisfiability problem. The second briefly discusses the history of SAT. The third describes a basic search algorithm for solving SAT. The fourth describes the general improvements to this algorithm. The fifth discusses benchmarks for evaluating a SAT tester's performance. We will list some recently released solvers in the last section.

2.1 Satisfiability Problem

2.1.1 Basic Definition

In propositional logic area, *atomic propositions* $x_1, \cdots, x_i, \cdots, x_n$ can be either *true* or *false*. A *truth assignment* is an assignment array of "true" or "false" to every atomic proposition. A *literal* is an atomic proposition x_i or its negation (complement) $\bar{x}_i$. A *clause* is a disjunction of literals and is *satisfied* by a given truth assignment if at least one of its literals is true. Otherwise, the clause is unsatisfied. [CC95]
The set of $\mathcal{S}$ clauses can be represented by the *conjunctive normal form* (CNF)

$$\bigwedge_{i \in \mathcal{S}} \Big(\bigvee_{j \in P_i} x_j \vee \bigvee_{j \in N_i} \bar{x}_j \Big)$$

where P_i is the set of x's subscript in i-th clause, N_i is the set of $\bar{x}$'s subscript in i-th clause.
Consider a propositional formula $\mathcal{S}$ in Conjunctive Normal Form (CNF) on a set of Boolean variables $x_1, x_2, \cdots, x_n$, the satisfiability (SAT) problem consists of testing whether clauses in $\mathcal{S}$ can all be satisfied by some consistent assignment of truth values (1 or 0) to variables. If it is the case, $\mathcal{S}$ is said satisfiable; otherwise, $\mathcal{S}$ is said unsatisfiable. If each clause exactly contains r literals, the subproblem is called r-SAT problem. 3-SAT is the smallest NP-complete sub-problem of SAT. [Li99]

2.1.2 Phase Transitions

Phase transition phenomenon is an interesting property of uniform Random-3-SAT. i. e., when systematically change (increasing or decreasing) the number of clauses, k, for fixed problem size n, a rapid change in satisfiability occurs. More precisely, when the number of clauses, k, is small enough for problem size n, almost all problems are satisfiable; when k is increased to some critical $k = k'$, the problem suddenly turns to be very difficult to be satisfied, i.e., with probability zero, we can find a satisfiable assignment to such a problem. Beyond k', almost all instances are unsatisfiable. Intuitively, k' characterizes the transition between a region of underconstrained instances which are almost satisfiable and overconstrained instances which are mostly unsatisfiable[CKT91]. We call this k' the phase transition critical number/ratio. For Random-3-SAT, this phase transition phenomenon occurs approximately at $k' = 4.26n$ for large n; for smaller n, the critical ratio of clauses/variable (k'/n) is slightly higher(around 4.27). Furthermore, for growing n the transition critical value k' becomes increasingly sharp. The problems from phase transition region are generally called *hard* problems. Many researchers use test-sets sampled from the phase transition region of uniform Random-3-SAT to test their algorithms. Similar phase transition phenomena have been observed for other classes of SAT, including uniform Random k-SAT with $k \geq 4$. But uniform Random-3-SAT is still the most popular instances for solver testing and algorithm research. In section 2.5.2, we will talk about this phenomena again.

2.2 History

As early as 1971, Stephen Cook has proved that SAT is NP-complete in his paper[?] that defined the notion of NP-completeness.

SAT problems can be regarded as a class of special cases of *constraint satisfaction problems*(CSP), in which each variable can take one of a finite number from a set of possible values. A plenty of solution techniques on CSP can be found in literature.

Dechter and Mackworth both provide excellent overviews in 1992 [Dec92, Mac92].

The first SAT search algorithm is owed to Davis and Putnam [DP60] and has been named as the *Davis-Putnam procedure*, or simply DP. In fact, we should mention that 50 years earlier before Davis and Putnam published their algorithm in 1960, L. Löwenheim has actually discovered it [CS88]. The difference between DP and the later version contributed by Davis, Logeman, and Loveland [DLL62], which well known as DPL, is as follows: DPL uses a splitting rule to replace the original problem by two smaller subproblems, whereas DP uses a variable elimination rule to replace the original problem usually by one larger subproblem [DR94, Fre95]. DPL is implemented more often than DP due to the four key disadvantages of variable elimination rule: it is more difficult to implement than the splitting rule; it tends to rapidly increase the length and number of clauses; it tends to generate a lot of redundant clauses; and it rarely generates new unit clauses [DLL62, Fre95].

There has been a common understanding that the history of SAT search techniques since 1960 has largely been the history of the various techniques that researchers have proposed to speed up and improve DPL.

Recently, the international interest in SAT algorithms has never been so high. Many professional web-pages and recent conferences have been set up to emphasized both analytical and experimental research on SAT [BB93, Com93]. Many people have been involved in SAT research and have developed much fast SAT testers. We will list some in the later section.

2.3 The Basic Search Algorithm

In literature, many kinds of methods have been released for solving *Conjunctive Normal Form Satisfiability (CNF-SAT)* problems.

As we all know, The Davis-Putnam-Loveland procedure (DPL)[DLL62] is the best complete algorithm to solve SAT problems. It was named after Martin Davis, George Logemann and Donald Loveland in 1962 [DLL62]. It is also one of the major practical methods for the SAT problems. The basic idea of the DPL procedure was presented in [DP60]. Figure 2.1 shows the basic version of DPL. It is a depth-first search algorithm through the set of all possible truth assignments until it either finds a satisfying truth assignment or detects the entire possible solution space without finding any.

```
Function Search(S)=
  case Truth-Vector(S) of
    T⇒ (true, Truth-Assignment(S))
    F⇒ (false,Truth-Assignment(S))
    I⇒ let l=an open literal in the open clauses of S,
      (bool,v)=Search(S[l ← T])
      in if bool then (true, v) else Search(S[l ← F]) end;
```

Figure 2.1: The basic search algorithm

Initially, we call it with $S_0(F)$. This function takes an argument vector as the system state and returns a <truth value, truth assignment> pair. The literal l in this function is called the premise, and the proposition associated with l is called the branching proposition or branching variable.

2.4 Some Improvements to the Basic Algorithm

In this section we briefly describe the main ways in which we can improve the basic search algorithm and explain the basic idea behind each of them. These techniques are conceptually general and well known, although not all of them are necessarily useful in practice.

2.4.1 Satz by Chu-Min Li

To our best knowledge, *Satz*, contributed by Chu-Min Li, is the fastest DPL procedure on random 3-SAT problems [Li99]. Roughly speaking, *Satz* is a very simple DPL procedure in which the next branches on the variable reduce the largest number of clauses in S at every node. More precisely, let $w(x)$ be the number of clauses reduced when x is assigned 1, and $w(\bar{x})$ the number of clauses reduced when x is assigned 0. The weight of x is defined by the equation suggested by Freeman in his PhD thesis [Fre95]:

$$H(x) = w(\bar{x}) * w(x) * 1024 + w(\bar{x}) + w(x) \tag{2.4.1}$$

Satz branches on x with the largest $H(x)$. Note that there is a balance in this equation. If $w(x) >> w(\bar{x})$ or $w(\bar{x}) >> w(x)$, x will generally not be selected as a branching variable.

We have known that the basic idea of the DPL procedure is to construct a binary search tree for solving S, each recursive call constituting a node of the tree. It is well known that given the number of variables, some problems, when the ratio of clause number to variable number is approximately equal to around 4.25, are much harder than others, necessitating construction of a much larger tree. In [Li99], Chu-Min

Li pointed out that the mean height of a search tree is somewhat irrelevant in the hardness of random 3-SAT problems when using a DPL, and if a search tree is larger, it is only because the search tree is wider. One of the objectives to implement a DPL procedure is to minimize the mean height of search trees (depth-first algorithm in DPL). Li figures out through experimental study that the essential objective should be minimizing the width (instead of the mean height) of search trees, which roughly implies using constraints to find contradictions (or reach the dead-node) as early as possible.

Based on such consideration, Chu-Min Li makes some improvements to DPL procedure. First, Li modifies the branching rule of *Satz* in order to generate more and stronger constraints. It is indicated that i) the constraint is stronger if it suppresses more solutions; ii) binary clauses sharing complementary literals can remove much more solutions and have more chances to lead to a dead-node where all solutions are removed. The improved DPL procedure suggested by Li branches next on the variable that can generate subproblems in which more binary clauses share complementary literals. So, the weight of the literal x is revised to:

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

where l and l' denote two different literals, and $f(\bar{l})$ is the number of binary occurrences of $\bar{l}$ in S if there is a sufficient number (larger than 10 as suggested by Li) of binary clauses in S otherwise it is the number of weighted occurrences of $\bar{l}$ in S. A clause of length >2 is counted as $5^{-(r-2)}$ binary occurrences. The weight $w(\bar{x})$ can be similarly defined. The weight of variable x is then obtained by simply replacing the value of both $w(x)$ and $w(\bar{x})$ in Freeman's formula in (2.4.1).

Li also suggested to use a looking further forward technique to search a dead-node.

The idea of a lookahead algorithm, or constraint propagator, is to set up a function that takes a state vector S and returns a state vector S' such that the function runs in low-order polynomial time. The satisfaction of S' is equivalent with the satisfaction of S, but S' is in some sense easier to be satisfied than S. For example, S' may have more valued propositions than S. In other words, S' may have fewer open propositions or clauses than S, or S' may stipulate some relationship between the truth values of two of more open propositions in S [Fre95].

Typically, many SAT search methods use more than one lookahead algorithm at every node of the search tree to simplify the remaining problem as much as possible. Li's looking further forward technique actually uses a lookahead algorithm — unit propagation in two levels. 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-node which can be reached by further (second level) unit propagations. If $Unitpropagation(F \cup \{l\})$ reduces more than T (empirically fixed to 65 for hard random 3-SAT problems) clauses, then 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 $Unitpropagation(F \cup \{l\} \cup \{\bar{y}\})$ should be executed. If both propagations reach a dead-node, then $\bar{l}$ should be satisfied [Li99]. These two propagations are called *unit propagations of second level*. This technique enables *Satz* to reach dead-node earlier so as to narrow a search tree and speed up the resolution.

2.4.2 Heuristics and Local Search

Optimization methods can be classified to two main categories – *exact* and *approximate* methods. Exact methods perform a systematic search for optimal solutions, while approximate methods can not theoretically guarantee to find optimal or even feasible solutions. It is designed to find a relative "good" or near-optimal solutions quickly. In operations research, approximate methods are commonly termed *heuristics*[Wal99]. Heuristics have received much interests in recent years due to their practical applications [Ree93, RSOR96, AL97].

Local search is an important class of heuristics with a long history for combinatorial optimization. Research work on local search can date back to 1950s and 1960s, when methods for the travelling salesman problem are presented. The basic idea of local search is to start from one of a feasible solution and iteratively make changes to improve the current solution. All variations of local search methods in literature have the common idea of local moves which are transitions in the space of all possible solutions no matter it is feasible or infeasible, typically according to a strategy that works by improving the *local gradient* of a measure of the solution quality (a strategy called *hillclimbing*) [Wal99].

Recently, local search techniques have gotten much success for model finding in propositional satisfiability [SLM92, Gu92, GW93]. This kind of local search strategies is also termed as *iterative repair*: Given a problem stated in terms of some variables and some constraints, one first generates an initial truth assignment of all variables. Normally it will violate a number of constraints. Iteratively, variable values are changed in order to reduce the number of conflicts with the constraints, i. e., in order to repair the current variable assignment to iteratively close to a satisfying variable assignment.

Among many efficient local search strategies for SAT, the *Walksat Strategy* contributed by Selman, Kautz, and Cohen[SKC94, MSK97] is the most successful one. The basic Walksat strategy performs a greedy local search equipped with a "noise" strategy. In [Wal99], Joachim Paul Walser discribes the method as follows: Initially, all variables are assigned a random value from {0,1}. It then iteratively selects a violated clause, from which it selects a variable such that changing its value yields the largest increase in the total number of satisfied clauses. If no such variable exists, a variable from this clause is selected randomly according to some detailed scheme. Such variable changes are repeated a fixed maximal number of iterations and then a restart takes place. If no satisfying assignment is found after a fixed number of restart, the procedure is terminated unsuccessfully.

2.4.3 Relaxation

If there exist some special cases of SAT and other appropriate problems which can be solved in low-order polynomial time, we can use *Relaxation*. Part of the algorithm proposed in this thesis use the idea of *Relaxation*. Given a CNF formula S, the idea of *Relaxation* here is to construct a subproblem $SP(S)$ such that it can be solved in low-order polynomial time, and solving it can sometimes indicate the satisfiability of the original problem. Although the subproblem need not be a SAT problem, generally it is. Two general techniques are used to construct such low-order subproblem [Fre95]: 1). deleting some clauses from S until the resulting problem is a special case (easy problem) of SAT[JSD93], or 2). deleting literals from each of the clauses in S until the resulting problem is a special case of SAT. The special cases of SAT are 2-SAT (every clause has at most 2 literals) and Horn-SAT (every clause has at most one positive literal). Both of the two special cases are solvable in linear time[APT79, DG84, Scu90].

2.5 Benchmarks

We need some benchmark problems to evaluate the performance of a SAT tester/solver. Generally, there are two main classes: specific problems, which may be encodings of real-world practical problems; and randomly generated problems, which can be very difficult to solve but with rare practical application.

2.5.1 Specific Problems

There are two widely known collections of specific problems. The fist was contributed by Mitterreiter and Radermacher in 1991 [MR91], and the second was created in conjunction form with the Second DIMACS Implementation Challenge in 1993 [Com92]. The first collection is available via anonymous FTP from `dimacs.rutgers.edu/pub/challenge/sat/benchmarks/cnf/faw.cnf.Z`. The second collection are currently available via anonymous FTP from `ftp://dimacs.rutgers.edu/pub/challenge/sat/benchmarks`.

2.5.2 Randomly Generated Problems

We can also create benchmark problems by some random problem generators. One of this type of benchmarks is *fixed probability problems* which is due to Goldberg[Gol79]. There are three parameters to generate the instances (P, N, ρ), where P is the number of propositions, N is the number of clauses, and ρ $(0 < \rho \leq 0.5)$ is a fixed number indicating the common appearance probability for each proposition l; Its complement $\bar{l}$ appears with probability ρ; neither l nor $\bar{l}$ appears with probability $1 - 2\rho$; clauses containing 0 or 1 literals are not allowed. The N clauses are generated independently, and within each clause, a given proposition l also appears independently.

Another class of randomly generated problems is *fixed clause length problems*, which is due to Franco and Paull [FP83]. Instances are generated from three parameters (P, N, K), where P and N have the same meaning with fixed probability problems generator, and K is the number of literals per clause. So each instance consists of N clauses, and each clause contains exactly K literals. Each clause is selected independently and randomly from the set of $\binom{P}{K}2^K$ possible clauses.

In Section 2.1.2, we have known that some problems are very difficult to solve when the ratio of clause number/variable number is close to some fixed number. Koutsoupias, Papadinitriou and Mitchell *et al.* described this phenomenon in [KP92, DBL92]. Koutsoupias and Papadinitriou pointed that the majority of fixed clause length problems are very easy to satisfy (for $K = 3$), i. e., a greedy local search algorithm can always succeed to find a satisfying assignment if one exists. Mitchell *et al.* showed experimentally that, when there exists some relationship in $< P, N, K >$, the problems are very difficult to solve on the average. Dubois's estimates of the crossover points (N/P) for 9 values of K are listed in Figure 2.2 [DABC96].

Because the problems near the crossover point are very difficult to solve on average, they are suitable to be benchmarks.

k	Crossover Point
3	4.24
4	9.88
5	21.05
6	43.31
7	87.70
8	176.41
9	353.88
10	708.78
...	...
20	726816.49

Figure 2.2: Some crossover points for Random k-SAT

2.6 Software and Internet Information for SAT solving

We can find many SAT testers/solvers on the Internet. We list here some of them in two classes.

2.6.1 Stochastic Local Search Algorithms (incomplete)

- GSAT, Version 41 (contributed by Henry Kautz and Bart Selman)
- WalkSAT, Version 35 (contributed by and Bart Selman)

2.6.2 Systematic Search Algorithms (complete)

- EQSATZ (version 2.0 of Feb. 2001; contributed by Chu-Min Li)
- 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, Version 1.0 (contributed by Roberto Bayardo)
- REL_SAT, E-mail access (maintained by Roberto Bayardo)
- REL_SAT-rand, Version 1.0 (contributed by Henry Kautz)
- SATO, Version 3.2.1 (contributed by Hantao Zhang)
- Satz213 (new version) (contributed by Chu-Min Li)
- Satz (contributed by Chu-Min Li)
- Satz-rand, Version 4.7 (contributed by Henry Kautz)
- Satz-rand, Version 2.0 (contributed by Carla Gomes, Henry Kautz, and Bart Selman)

2.6.3 Some useful Links to SAT Related Sites

- SATLIB — The Satisfiability Library:
  `http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/`
- SATLIVE — Up-to-date links to satisfiability problem:
  `http://www.satlive.org`
- The Sat-Ex Site: The experimentation web site around the satisfiability problem:
  `http://www.lri.fr/~simon/satex/`

Chapter 3

Integer Programming Formulation for Logic Problem

In this chapter, I give a survey of the connection between propositional logic and integer programming.

In propositional logic, several problems, such as satisfiability, MAX SAT and logical inference, can be formulated as integer programs.

A truth assignment satisfies the set of $\mathcal{S}$ clauses:

$$\bigvee_{j \in P_i} x_j \vee (\bigvee_{j \in N_i} \bar{x_j}) \qquad \text{for all } i \in s,$$

if and only if the corresponding 0,1 vector satisfies the following system of inequalities: [CC95]

$$\sum_{j \in P_i} x_j - \sum_{j \in N_i} x_j \geq 1 - |N_i| \qquad \text{for all } i \ \in s,$$

where the value of $|N_i|$ is the number of $\bar{x}s$ in the i-th clause.

Given a 0, $\pm$ 1 matrix A, $n(A)$ is the vector whose i-th component $n_i(A)$ is the number of -1's in the i-th row of A. The vector of all 1's is denoted by $\mathbf{1}$. Under such denotement, the above system of inequalities takes the form

$$Ax \geq \mathbf{1} - n(A) \tag{3.0.1}$$

3.1 SAT Problem

Given a set $\mathcal{S}$ of clauses, the *satisfiability problem*(SAT) consists of finding a truth assignment that satisfies all the clauses in $\mathcal{S}$ or show that none exists. Equivalently, SAT consists in finding a 0,1 solution x to (3.0.1) or show that none exists.[CC95]

3.2 MAXSAT Problem

Given a set $\mathcal{S}$ of clauses and a weight vector w whose components are indexed by the clauses in $\mathcal{S}$, the *weighted maximum satisfiability problem (MAXSAT)* is to find a truth assignment that maximizes the total number of weighted satisfing clauses. The integer programming formulation of MAXSAT is:

$$Min \qquad \sum_{i=1}^{m} w_i s_i$$

$$Ax + s \geq \mathbf{1} - n(A)$$

$$x \in \{0,1\}^n, s \in \{0,1\}^m$$

where A is a 0, $\pm$ 1 matrix.

3.3 Logical Inference Problem

Given a set $\mathcal{S}$ of clauses (the premises) and a clause C(the conclusion), *logical inference* in propositional logic consists of deciding whether every truth assignment that satisfies $\mathcal{S}$ also satisfies the conclusion C.

The clause C can be formulated by an inequality using transformation (3.0.1):

$$cx \geq 1 - n(c),$$

where c is a 0, $\pm$ 1 vector and $n(c)$ is the number of components in (c) which is equal to -1. Therefore, C cannot be deduced from $\mathcal{S}$ if and only if the integer programming problem

$$min\{cx : Ax \geq \mathbf{1} - n(A), x \in \{0,1\}^n\} \tag{3.3.1}$$

has a solution with the optimal value $-n(c)$.

3.4 Weighted Exact Satisfiability Problem

Let a vector w be the weights associated with the atomic propositions vecter x, and let $\mathcal{S}$ be a set of clauses, $\mathcal{S}'$ be a subset of $\mathcal{S}$. The *weighted exact satisfiability problem* consists of finding a truth assignment (if any) such that[CC95]:

- Every clause in $\mathcal{S}$ is satisfied and, in every clause of $\mathcal{S}'$, there exists exactly one literal that assumes the value true, and
- The sum of the weights of the atomic propositions that assume the value true is maximized.

The formulation is the following integer programming model:

$$\begin{aligned} Max \quad & \textstyle\sum_{i=1}^{m} w_i x_i \\ & Ax \geq \mathbf{1} - n(A) \\ & A'x = \mathbf{1} - n(A') \\ & x \in \{0,1\}^n \end{aligned}$$

where A' is the row submatrix of A corresponding to $\mathcal{S}'$. Note that the logical inference problem is a special case of the weighted exact satisfiability problem.

The above four problems are NP-hard in general but SAT and logical inference can be solved efficiently for Horn clauses, clauses with at most two literals and several other related clauses[CH91, Tru90].

Chapter 4

Integer Programming Formulation for SAT Problem

In the last chapter, we introduce the integer programming formulations for some logic problems. In this chapter, we will focus on the integer programming formulation for 3-SAT.

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

In literature, many transformation have been proposed for converting the satisfiability problem into an integer programming problem. These transformations usually create nonlinear integer programming problems, which are generally very difficult to solve. Actually, we can convert the CNF clauses into IP constraints by a novel simple way: we can interpret "$\bar{x}_i$" as "$1 - x_i$", and the symbol of "$\bigvee$" as "+" operation.
In addition, each literal can only take a boolean value 0 or 1, and each clause will be true if at least one literal takes the value 1. Therefore, we can convert the CNF

clause into an IP constraint. For example:

$$(x_1 \vee x_2 \vee x_3) \Rightarrow \quad x_1 + x_2 + x_3 \geq 1$$

$$(x_1 \vee \bar{x}_2 \vee x_3) \Rightarrow \quad x_1 + (1 - x_2) + x_3 \geq 1$$

$$(\bar{x}_1 \vee \bar{x}_2 \vee x_3) \Rightarrow \quad (1 - x_1) + (1 - x_2) + x_3 \geq 1$$

$$(\bar{x}_2 \vee \bar{x}_3 \vee \bar{x}_5) \Rightarrow \quad (1 - x_2) + (1 - x_3) + (1 - x_5) \geq 1$$

4.2 Integer Programming Model for 3-SAT

According to the clause transformation rule of last section, we construct an integer programming model for 3-SAT problem:

$$(IP) \qquad Min \qquad \mathbf{1}'s \tag{4.2.1}$$

$$s.t. \qquad Ax + s \geq \mathbf{1} - n(A) \tag{4.2.2}$$

$$x \in \{0,1\}^n, \quad s \in \{0,1\}^m \tag{4.2.3}$$

where A is an m by n, $0, \pm 1$ matrix, $n(A)$ is a vector whose i-th component $n_i(A)$ is the number of "-1" in the i-th row of A, and $\mathbf{1}$ is a vector whose components are all 1s.

4.3 The Equivalence of the SAT and the IP

If the original SAT problem is feasible (satisfiable), the corresponding integer programming problem (IP) should achieve the optimal value of 0. If the original SAT problem is infeasible (unsatisfiable), the corresponding integer programming problem (IP) cannot achieve zero optimal value. Therefore, the original SAT problem and the above integer programming model (IP) are equivalent.

4.4 Example

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

$$\bar{x}_4 \vee \bar{x}_2 \vee \bar{x}_1$$

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

$$x_3 \vee \bar{x}_1 \vee x_4$$

$$\bar{x}_4 \vee \bar{x}_2 \vee x_1$$

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

$$\bar{x}_5 \vee \bar{x}_2 \vee x_1$$

$$\bar{x}_5 \vee x_3 \vee \bar{x}_2$$

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

$$x_1 \vee x_4 \vee x_5$$

$$\bar{x}_4 \vee \bar{x}_5 \vee \bar{x}_3$$

$$\bar{x}_5 \vee x_3 \vee x_4$$

$$x_2 \vee x_1 \vee x_4$$

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

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

$$\bar{x}_4 \vee \bar{x}_2 \vee x_1$$

$$x_5 \vee \bar{x}_3 \vee x_1$$

$$x_4 \vee \bar{x}_1 \vee x_3$$

$$\bar{x}_2 \vee x_1 \vee x_5$$

$$x_5 \vee x_4 \vee x_2$$

$$\bar{x}_1 \vee x_4 \vee x_5$$

$$x_3 \vee x_1 \vee \bar{x}_2$$

$$\bar{x}_2 \vee \bar{x}_4 \vee \bar{x}_1$$

Then, we convert this SAT problem into integer programming problem:

$$\begin{aligned}
min \quad & \textstyle\sum_{i=1}^{22} s_i \\
s.t. \quad & -x_1 - x_2 - x_4 + s_1 \geq -2 \\
& x_1 + x_2 - x_3 + s_2 \geq 0 \\
& -x_1 + x_3 + x_4 + s_3 \geq 0 \\
& x_1 - x_2 - x_4 + s_4 \geq -1 \\
& -x_1 + x_2 + x_3 + s_5 \geq 0 \\
& x_1 - x_2 - x_5 + s_6 \geq -1 \\
& -x_2 + x_3 - x_5 + s_7 \geq -1 \\
& -x_1 - x_3 - x_5 + s_8 \geq -2 \\
& x_1 + x_4 + x_5 + s_9 \geq 1 \\
& -x_3 - x_4 - x_5 + s_{10} \geq -2 \\
& x_3 + x_4 - x_5 + s_{11} \geq 0 \\
& x_1 + x_2 + x_4 + s_{12} \geq 1 \\
& -x_2 - x_3 + x_5 + s_{13} \geq -1 \\
& -x_1 + x_4 - x_5 + s_{14} \geq -1 \\
& x_1 - x_2 - x_4 + s_{15} \geq -1 \\
& x_1 - x_3 + x_5 + s_{16} \geq 0
\end{aligned}$$

$$-x_1 + x_3 + x_4 + s_{17} \geq 0$$

$$x_1 - x_2 + x_5 + s_{18} \geq 0$$

$$x_2 + x_4 + x_5 + s_{19} \geq 1$$

$$-x_1 + x_4 + x_5 + s_{20} \geq 0$$

$$x_1 - x_2 + x_3 + s_{21} \geq 0$$

$$-x_1 - x_2 - x_4 + s_{22} \geq -2$$

$$x_i \in \{0, 1\}, s_i \in \{0, 1\}$$

Now, we can use a branch-and-bound algorithm to solve the above zero-one linear integer problem, then check the satisfiability of the original problem. Branch-and-bound algorithm is a simple method for solving IP problems. In Chapter 6, we will derive our reasonable branching rule and bound rule for our problem formulation.

Chapter 5

Integer Solvability of Linear Programs

In general, linear programming problems are much easier to solve than discrete optimization problems, and the algorithms for linear programming are important in their own right. A natural question is when we will be lucky to find an integral optimal solution to a linear programming relaxation of an integer optimization problem. It turns out that if the polyhedron possesses the integer extrema property, then the linear program always achieves its optimum at an integer point. In this section, we will present some classic results in literature.

5.1 Unimodularity

We consider the integer programming problem:

$$\begin{aligned} (IP) \quad & Min \quad cx \\ s.t. \quad & Ax = b \\ & x \in Z_+^n. \end{aligned}$$

Definition 5.1.1. The constraint matrix A is said to be *Unimodular* if every basis matrix B of A has determinant, $det(B) = \pm 1$.

The following classic result of Veinott and Dantzig (1968) [VD68] shows the implications for integer solvability.

Theorem 5.1.1. ***(Unimodularity and Equality Linear Programs).*** *Let A be an integer matrix with linearly independent rows. Then the following are equivalent:*

1. *A is unimodular.*

2. *Extreme points of $S^{=} = \{x : Ax = b, x \geq 0\}$ are integral for any integer right-hand-side b.*

3. *Every basis submatrix B of A has an integral inverse B^{-1}.*

Now returning to (IP), it is clear that when A is unimodular, the linear programming relaxation $min\{cx : AX = b, x \in R^{+}\}$ solves (IP).

5.2 Totally Unimodularity

We consider the integer programming problem:

$$\begin{aligned} (IP) \quad & Max \quad cx \\ s.t. \quad & Ax \leq b \\ & x \in Z^n_+, \end{aligned}$$

where A is an integer matrix with full row rank, and b is an integer column vector.

From the linear programming theory, we know that basic feasible solutions (including slack variables) take the form: $x = (x_B, x_N) = (B^{-1}b, 0)$ where B is an $m \times m$ nonsingular submatrix of (A, I) and I is an $m \times m$ identity matrix.

Observation 5.2.1. (Sufficient Condition) If the optimal basis B has $det(B) = \pm 1$, the linear programming relaxation solves (IP) with integral b.[Wol98]

Proof. From Cramer's rule, $B^{-1} = B^*/det(B)$ where B^* is the adjoint matrix. The entries of B^* are all products of terms of B. Thus B^* is an integral matrix, and as $det(B) = \pm 1$, B^{-1} is also integral. Thus $B^{-1}b$ is integral for all integral b. □

Now, we have another question — when one will always be lucky, i. e., when do all bases or all optimal bases satisfy $det(B) = \pm 1$?

Definition 5.2.1. A matrix A is *totally unimodular (TU)* if every square submatrix of A has determinant +1, -1 or 0.

Hoffman and Kruskal's (1956) classic result on total unimodularity[HK56] is as follows:

Theorem 5.2.1. ***(Totally Unimodularity and Inequality Linear Programs)*** *Let A be an integer matrix. Then the following are equivalent.*

1. *Every susbmatrix of A has determinant ± 1 or 0.*

2. *Extreme points of $S^{\geq} = \{x : Ax \geq b, x \geq 0\}$ are integral for any integer right-hand-side* $\boldsymbol{b}$.

3. *Every nonsingular submatrix of A has an integer inverse.*

Proof. Please refer to the detailed proof in [PR88]. □

Example 5.2.1. Matrices that are not TU:

$$\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \begin{pmatrix} -1 & -1 \\ -1 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}, \cdots\cdots$$

Example 5.2.2. Matrices that are TU:

$$\begin{pmatrix} 1 & 1 \\ -1 & -1 \end{pmatrix}, \begin{pmatrix} 1 & -1 & -1 & 0 \\ -1 & 0 & 0 & 1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}, \cdots\cdots$$

The most prominent class of totally unimodular matrices are those that arise from the vertex-arc incidence matrix of a directed graph.

Theorem 5.2.2. ***Totally Unimodularity of Vertex-Arc Incidence Matrices.*** *Every vertex-arc incidence matrix of a directed graph is totally unimodular.*

Proof. Please refer to the detailed proof in [PR88]. □

Observation 5.2.2. If A is TU, $a_{ij} \in \{+1, -1, 0\}$ for all i, j.[Wol98]

Proposition 5.2.3. *A matrix A is TU if and only if [Wol98]*

1. *the transpose matrix A^T is TU.*

2. *the matrix (A, I) is TU.*

3. *the matrix $\binom{A}{I}$ is TU.*

From this proposition, we know that the linear solvability of the model in Section 4.2 is in fact the totally unimodularity of matrix $[A, I]$, further more, the totally unimodularity of matrix A itself.

Proposition 5.2.4. *(Sufficient Conditions) A matrix A is TU if [Wol98]*

1. $a_{ij} \in \{+1, -1, 0\}$ *for all* i, j.

2. *Each column contains at most two nonzero coefficients* ($\sum_{i=1}^{m} |a_{ij}| \leq 2$).

3. *There exists a partition* (M_1, M_2) *of the set* M *of rows such that each column* j *contains two nonzero coefficients satisfies* $\sum_{i \in M_1} a_{ij} - \sum_{i \in M_2} a_{ij} = 0$.

Proof. Please refer to the detailed proof in [Wol98] □

Now returning to (IP), it is clear that when A is totally unimodular, the linear programming relaxation $max\{cx : Ax \leq b, x \in R^+\}$ solves (IP).

What if a matrix's origin is not known to be a vertex-arc incidence? It may still be possible to be totally unimodular, i.e., it may also be that the matrix is totally unimodular but not of network origin.

The nice property of TU matrix motivates us to find a branching order to force a revised coefficient matrix A to be closer to TU, and eventually, to find the optimal integer solution by solving linear programs.

Seymour's decomposition theorem of totally unimodular matrices [Sey80] represents a recent elegant result. The decompositions involved in his theorem are 1-separations, 2-separations and 3-separations which Seymour defined in [Sey80]. He used matroid theory to prove this decomposition theorem.

5.3 Some Results on Recognition of Linear Solvability of IP

Perfect, ideal and balanced matrices have beautiful polyhedral properties that have been recognized in the last 30 years due to their special structures.

A $0, \pm 1$ matrix A is *perfect* if the fractional generalized set packing polytope $\{x : Ax \leq 1 - n(A), 0 \leq x \leq 1\}$ has only integral extreme points. It is *ideal* if the fractional generalized set covering polyhedron $\{x : Ax \geq 1 - n(A), 0 \geq x \geq 1\}$ has only integral extreme points. It is *balanced* if, in every square submatrix with two nonzero entries per row and per column, the sum of the entries is a multiple of four. The study of the characteristics of TU in [HK56] shows that a totally unimodular matrix is both perfect and ideal. The class of balanced $0, \pm 1$ matrices also properly includes totally unimodular $0, \pm 1$ matrices [CCKV01].

As far as we know, no algorithm is known for perfection and idealness recognition. However, Conforti et al give a polynomial time algorithm for checking balancedness [CCKV00]. This algorithm is complicated and its computational complexity, although polynomial, is rather high.

Chapter 6

TU Based Matrix Research Results

It is obvious that totally unimodular matrices are highly desirable in discrete optimization, especially in SAT problems, because they assure an integer solvability (for integer right-hand-sides). When the matrices are known to be arise from a vertex-arc incidence matrix, we have already seen (in Theorem 5.2.2) that total unimodularity is guaranteed.

But in the real world applications, the constraint matrices are often not of totally unimodular. We have the following results from investigation of all $0, \pm 1$ 2×2, and $0, 1$ 3×3 matrices.

6.1 2×2 Matrix's TU Property

There are 81 ($=3^4$) 2×2 $(0, \pm 1)$ matrices, among which only 8 cases are non-totally unimodular (determinant is not equal to 0, 1 or -1).

$$\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}, \begin{pmatrix} -1 & -1 \\ 1 & -1 \end{pmatrix}, \begin{pmatrix} -1 & -1 \\ -1 & 1 \end{pmatrix},$$

$$\begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}, \begin{pmatrix} -1 & 1 \\ 1 & 1 \end{pmatrix}, \begin{pmatrix} 1 & -1 \\ -1 & -1 \end{pmatrix}, \begin{pmatrix} -1 & 1 \\ -1 & -1 \end{pmatrix}$$

Observation 6.1.1. For any 2×2 $(0, \pm 1)$ matrix A, it is not totally unimodular if and only if the two entries in one row have the same sign, while the two entries in the other row have different sign.

6.2 Extended Integer Programming Model for SAT

In problem (IP), if $\bar{x}_i$ is regarded as a new variable x_{n+i} (suppose n variables), we can get the extended IP model with no negative coefficient in the constraint matrix for the original SAT problem.

For example, n=5:

$$
\begin{aligned}
\bar{x}_1 \vee \bar{x}_2 \vee x_3 \quad &\Rightarrow \quad x_3 + x_6 + x_7 &\geqslant 1 \\
& \quad x_1 + x_6 &= 1 \\
& \quad x_2 + x_7 &= 1 \\
\bar{x}_2 \vee \bar{x}_3 \vee \bar{x}_5 \quad &\Rightarrow \quad x_7 + x_8 + x_{10} &\geqslant 1 \\
& \quad x_2 + x_7 &= 1 \\
& \quad x_3 + x_8 &= 1 \\
& \quad x_5 + x_{10} &= 1
\end{aligned}
$$

In this extended model, we have doubled variables, and n more clauses. We can formulate this model by the following matrix form

$$
\begin{aligned}
& & Min \quad & \mathbf{1}'s \\
(IP) \quad & s.t. & A_{new}x + s_1 &\geq \mathbf{1} \\
& & (I, I)x + s_2 &= \mathbf{1} \\
& & x \in \{0,1\}^{2n}, &\quad s \in \{0,1\}^{m+n}
\end{aligned}
$$

where A_{new} is an m by $2n$ 0-1 matrix, I is an n by n matrix, and $s = [s_1', s_2']'$.
It is easy to see the linear solvability of this new model is in fact the totally unimodularity of the matrix $\begin{bmatrix} \begin{bmatrix} A_{new} & \\ I_n & I_n \end{bmatrix} & I_{m+n} \end{bmatrix}$, and further more, the totally unimodularity of matrix $\begin{bmatrix} A_{new} & \\ I_n & I_n \end{bmatrix}$.

6.3 3×3 Matrix's TU Property

Since all 2×2 0-1 matrices are totally unimodular, here, we investigate the totally unimodularity property of 3×3 $0-1$ matrices.
There are 512 (=2^9) candidates. We enumerate all possible 3×3 0-1 matrices and exclude those whose determinant is 0, 1 or -1, there are 108 non-totally unimodular instances left. Furthermore, we find these instances have certain common characteristic by induction.

Lemma 6.3.1. *Given four points: $A(a_1, a_2, a_3)$, $B(b_1, b_2, b_3)$, $C(c_1, c_2, c_3)$ and $D(d_1, d_2, d_3)$, the volume of the tetrahedron constituted by the four points is:*

$$V = \left| \frac{1}{6} det(\begin{bmatrix} a_1 & a_2 & a_3 & 1 \\ b_1 & b_2 & b_3 & 1 \\ c_1 & c_2 & c_3 & 1 \\ d_1 & d_2 & d_3 & 1 \end{bmatrix}) \right|$$

Proof. This is a well-known result in analytic geometry, we give a brief proof here.
V is sixth of the volume of the parallelepiped constituted by $\overrightarrow{AB}$, $\overrightarrow{AC}$, $\overrightarrow{AD}$. And the volume of the latter one is $|(\overrightarrow{AB}, \overrightarrow{AC}, \overrightarrow{AD})|$. $\overrightarrow{AB} = \{b_1 - a_1, b_2 - a_2, b_3 - a_3\}$, $\overrightarrow{AC} = \{c_1 - a_1, c_2 - a_2, c_3 - a_3\}$, $\overrightarrow{AD} = \{d_1 - a_1, d_2 - a_2, d_3 - a_3\}$. So,

$$V = \left|\frac{1}{6}det([\overrightarrow{AB}, \overrightarrow{AC}, \overrightarrow{AD}])\right| = \left|\frac{1}{6}det(\begin{bmatrix} b_1 - a_1 & b_2 - a_2 & b_3 - a_3 \\ c_1 - a_1 & c_2 - a_2 & c_3 - a_3 \\ d_1 - a_1 & d_2 - a_2 & d_3 - a_3 \end{bmatrix})\right|$$

$$= \left|\frac{1}{6}det(\begin{bmatrix} a_1 & a_2 & a_3 & 1 \\ b_1 & b_2 & b_3 & 1 \\ c_1 & c_2 & c_3 & 1 \\ d_1 & d_2 & d_3 & 1 \end{bmatrix})\right|$$

Note that V should be the absolute value of the determinant. □

Theorem 6.3.2. *For a 3×3 0-1 matrix, only when there is exactly one zero per row and per column, its determinant is not 1, -1 or 0, i. e., it is not Totally Unimodular.*

Proof. From Lemma 6.3.1, the volume of the tetrahedron constituted by the four points $A(a_1, a_2, a_3)$, $B(b_1, b_2, b_3)$, $C(c_1, c_2, c_3)$ and $D(0, 0, 0)$ should be:

$$V = \left|\frac{1}{6}det(\begin{bmatrix} a_1 & a_2 & a_3 & 1 \\ b_1 & b_2 & b_3 & 1 \\ c_1 & c_2 & c_3 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix})\right| = \left|\frac{1}{6}det(\begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{bmatrix})\right|$$

So,

$$\left|det(\begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{bmatrix})\right| = 6V$$

A spacial point A, if the elements of A can only have the value 0 or 1, can be one of these 8 points: (0, 0, 0), (1, 0, 0), (0, 1, 0), (0, 0, 1), (1, 1, 0), (1, 0, 1), (0, 1, 1), (1, 1, 1). They are just the 8 vertices of an unit cube (see Figure 6.1).

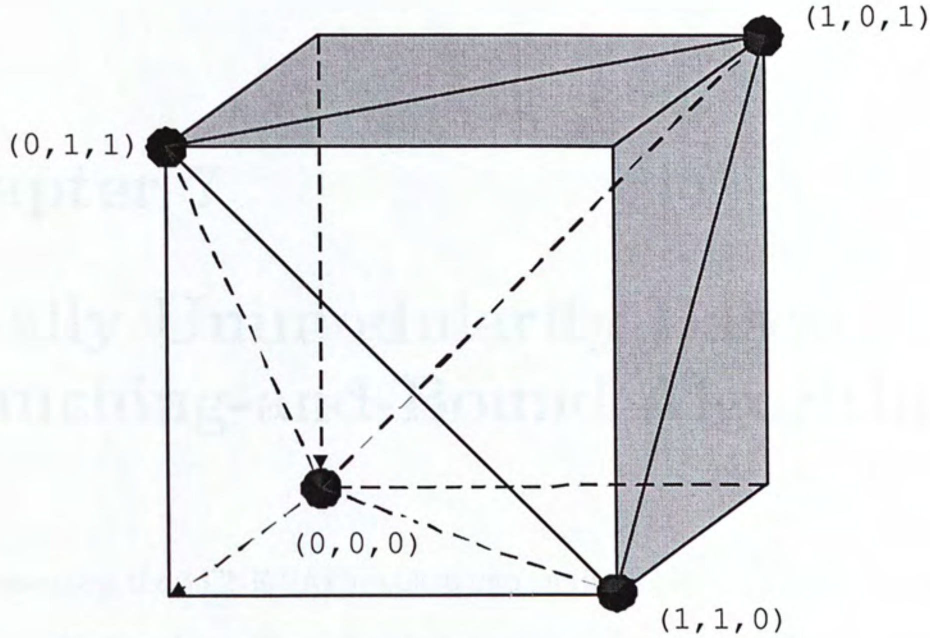

Figure 6.1: Determinant-Volume Relationship

If we fix the origin (0, 0, 0), together with any other 3 vertices of this cube, say, $A(a_1, a_2, a_3)$, $B(b_1, b_2, b_3)$ and $C(c_1, c_2, c_3)$, it forms a tetrahedron. Only the tagged vertices in Figure 6.1 can form a tetrahedron with volume 1/3. Any other 3 vertices together with the origin form tetrahedron with volume 0 or 1/6, i. e., according to lemma 6.3.1, only the determinant formed by points (0, 1, 1), (1, 0, 1), (1, 1, 0) can not get the value of 0 or 1 (=$6V$). □

Chapter 7

Totally Unimodularity Based Branching-and-Bound Algorithm

After converting the 3-CNF-SAT problem into an IP problem, a totally unimodularity based branching-and-bound method is proposed to find out whether the problem is feasible, and furthermore, what the feasible solution is.

7.1 Introduction

There are many methods proposed in literature to solve an integer programming problem, such as Branch-and-Bound methods, Cutting Planes methods, etc. Among the methods for solving an integer programming problem, branch-and-bound and cutting planes are two typical solution schemes. Although branch-and-bound method is very popular, the time complexity is $O(2^n)$ in the worst case where n is the number of variables. In this thesis, we propose a totally unimodularity based branch-and-bound method as the skeleton of our searching algorithm. In this algorithm, we set up a branching rule, and bounding rule. Then a binary search tree is constructed (see Figure 7.1) for the search procedure. Each node of the tree represents one recursive

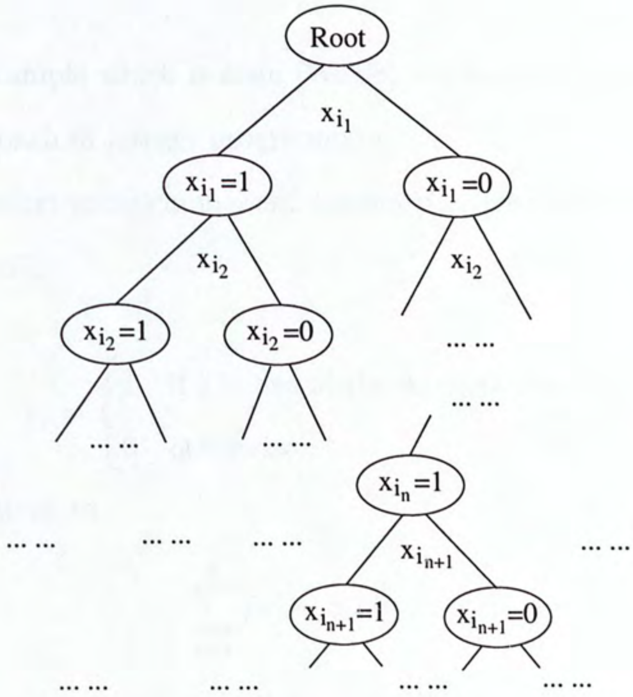

Figure 7.1: Binary Search Tree

call. We prune the hopeless branch according our proposed bound rule, and continue the branching procedure in the rest promising branches (nodes) till finding out the feasible solution for the SAT problem.

7.1.1 Enumeration Trees

Enumeration trees analysis belongs to enumerative approaches to integer programming. These approaches take advantage of the fact that in a bounded integer linear programming (ILP) or mixed integer linear programming (MILP), the set of values of the integer variables is finite. The basic idea of enumerative methods can be explained using a tree.[Wol98]

Example

We use a simple example which is from [Wol98] to illustrate the application of enumeration tree approach in integer programming.

If we are asked to select some numbers in distinct positive integers to make them sum to 8, how can we do?

Letting

$$x_j = \begin{cases} 1 & \text{if } j \text{ is one of the integers chosen,} \\ 0 & \text{otherwise.} \end{cases}$$

we require all solutions to

$$\sum_{j=1}^{8} j x_j = 8 \qquad (7.1.1)$$

$$x_j = 0, 1 \quad \text{for all } j$$

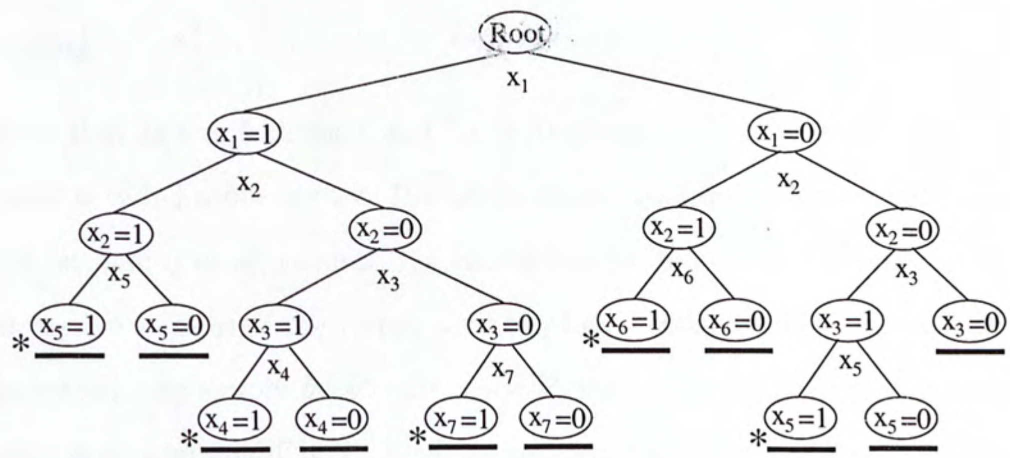

Figure 7.2: Example for Enumerate search tree

The solutions are given by the unique paths from vertex *Root* to each of the vertices marked by an asterisk in Figure 7.2. Each edge imposes a constraint, and each vertex

j represents the constraint set of (7.1.1) in addition to the constraints given by the edges along the unique path P_j from v_0 to v_j. A line underneath a vertex indicates that no further exploration from that vertex can be profitable. Such vertex is said to be *fathomed.*

Suppose that the problem is to find some $x \in S$, then vertex j restricts x to S_j, where S_j is the intersection of S with the set of points satisfying the constraints given by the edges of P_j. If P_j has $k+1$ vertices:

$$v_0 = v_{j(0)}, v_{j(1)}, ..., v_{j(k-1)}, v_{j(k)} = v_j$$

$$\text{then} \qquad S = S_{j(0)} \supseteq S_{j(1)} \supseteq ... \supseteq S_{j(k)} = S_j$$

$v_{j(k-1)}$ is called the *predecessor* of v_j, which in turn is called a *successor* of its predecessor. Note that a vertex has a unique predecessor but generally more than one successor[GN72].

Branching

A vertex that is not fathomed and its corresponding constraint set has not been separated is called a *live* vertex. *Branching* means choosing a live vertex to consider next for fathoming or separation. A common rule for branching is branching to one of the successive vertices of the vertex currently being considered. If the current vertex j is fathomed, one simply *backtracks* along P_j until a vertex having at least one live successor is encountered[GN72]. One can select one of these successive vertices to do branching. If no live vertices are left, the enumeration process is complete.

7.1.2 The Concept of Branch and Bound

Branch and bound is an optimization technique that uses the basic tree enumeration described in the previous section. It involves calculating upper bounds and lower bounds on the objective function, in order to accelerate the fathoming process and thereby to curtail the enumeration. For the problem

$$max \;\; z(x), \qquad x \in S. \tag{7.1.2}$$

The bounds are determined as follows.

Upper Bounds

If the enumeration is at vertex j. The problem to be considered at v_j is

$$max \;\; z(x), \qquad x \in S_j \tag{7.1.3}$$

Let

$$z_j^* = \begin{cases} z(x^*(j)) & \text{if } x^*(j) \text{ solves (7.1.3)}, \\ -\infty & \text{if } S_j = \varnothing, \\ \infty & \text{if (7.1.3) is unbounded.} \end{cases}$$

An upper bound $\bar{z}_j \geq z_j^*$ may be determined by considering the relaxation of (7.1.3).

Lower Bounds

A lower bound $\underline{z}_j$ satisfies $\underline{z}_j \leq z_j^*$. One way to calculate a lower bound is to find any $x' \in S_j$ and let $\underline{z}_j = z(x')$. If v_k is the predecessor of v_j, then $\underline{z}_j \leq z_k^*$, which yields an important result that $\underline{z}_j \leq z_0^*$.

Fathoming by Bounds

Vertex j is fathomed if either

(a) $\bar{z}_j = \underline{z}_j$, or

(b) $\bar{z}_j \leq \underline{z}_k^*$

In case (a) no better solution can be found to (7.1.3). When case (b) occurs, no successor of v_j can yield a solution that improves on the best known solution to (7.1.2).

7.2 TU Based Branching Rule

Since totally unimodularity is a very nice property for solving an integer programming problem, it is very natural to force the constraint matrix to be close to this state by fixing some variables. In the last chapter, we have presented the non-totally unimodular (bad) cases for 2×2 $0, \pm 1$ and 3×3 $0, 1$ matrices. The branching variable selection rule can be expressed like the follows:

> *The variable that will yield the largest decrease in the number of "bad" cases by fixing this variable should be selected as the next branching variable.*

7.2.1 How to sort variables based on 2×2 submatrices

From Observation 6.1.1, the specific form of 2×2 $0, \pm 1$ non-totally unimodular matrix is known. We design the following algorithm to get a variable order based on the above branching rule:

1. Construct the non-TU counter E_{ij}:

   Considering columns i and j, define a variable $E_{ij}(i < j)$, for x_i and x_j, to measure the number of such "bad" matrices produced by columns i and j :

   $$E_{ij} = e^s_{ij} \times e^d_{ij}$$

   where e^s_{ij} is the number of rows which have the pairs with the same signs, and e^d_{ij} the rows which have the pairs with different signs.

2. Get the variable order

   (a) Generate a table (E_{ij}).

   (b) The weight vector for each variable, denoted by w (with component w_i), is defined as follows:

   $$w_i = \sum_{j>i} E_{ij} + \sum_{j<i} E_{ji}$$

   (c) Find the largest one from w_i, say it is w_k, then the current branch variable is x_k.

   (d) Change the weight vector(for each i):

   $$w_i = \begin{cases} w_i - E_{ik} & \text{if } i < k \\ 0 & \text{if } i = k \\ w_i - E_{ki} & \text{if } i > k \end{cases}$$

   (e) If there exists any $w_i > 0$, goto step 2c; otherwise, stop this part of sorting process.

7.2.2 How to sort the rest variables

In order to reduce the computation time, we first order the variables according to the 2×2 totally unimodular rule. After this process, all of the remaining 2×2 submatrices in the coefficient matrix constituted by the rest variables are totally unimodular. We need to consider the TU property of its 3×3 submatrices.

Recall that for a 3×3 0-1 matrix, only when there is exactly one zero per row and per column, the determinant is not 1, -1 or 0, i.e., it is not Totally Unimodular. For example:

$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}, \cdots\cdots$$

In fact, they are all constituted by row vectors: (1, 1, 0), (1, 0, 1) and (0, 1, 1). Naturally, we have the following branch algorithm:

1. Model revision

   We revise the current (0, ±1) model to our extended IP model (0-1 model in Chapter 6.2) after deleting those variables ordered in the previous section 7.2.1.

2. Construct the non-TU counter E_{ijk}

   Considering columns i, j and k, we define a variable $E_{ijk}(i < j < k)$, for x_i, x_j and x_k, to measure the number of such "bad" matrices produced by the columns i, j and k :

   $$E_{ijk} = e^f_{ijk} \times e^s_{ijk} \times e^t_{ijk}$$

   where e^f_{ijk} is the number of rows which have exactly one zero in the first position i, e^s_{ijk} is the number of rows which have exactly one zero in the second position j, and e^t_{ijk} is the number of rows which have exactly one zero in the third position k.

3. Get the variable order

   (a) Generate a table (E_{ijk}).

   (b) The weight vector for each variable, denoted by w (with component w_i), is defined as follows:

$$w_i = \sum_{j,k:i<j<k} E_{ijk} + \sum_{j,k:j<i<k} E_{jik} + \sum_{j,k:j<k<i} E_{jki}$$

   (c) Find the largest one from w_i, say it is w_k, then the current branch variable is x_k.

   (d) Change the weight vector(for each i):

$$w_i = \begin{cases} w_i - \sum\limits_{j:i<j<k} E_{ijk} - \sum\limits_{j:i<k<j} E_{ikj} - \sum\limits_{j:j<i<k} E_{jik} & \text{if } i < k \\ 0 & \text{if } i = k \\ w_i - \sum\limits_{j:k<i<j} E_{kij} - \sum\limits_{j:j<k<i} E_{jki} - \sum\limits_{j:k<j<i} E_{kji} & \text{if } i > k \end{cases}$$

   (e) If there exists any $w_i > 0$, goto step 3c; otherwise, ordering process complete.

7.3 TU Based Bounding Rule

The most common way to solve integer programs is to use implicit enumeration, or *Branch-and-Bound*, in which linear programming relaxations provide the bounds generally. In this work, we propose another bound rule according to the relationship between the satisfiability problem and the associated integer programming problem.

1. If the optimal objective value is nonzero, this node is pruned.
2. If the optimal objective value is zero, and the optimal solution is integral, this node is one of the feasible solutions to the original SAT problem.
3. If the optimal objective value is zero, but the optimal solution is not integral, this node is active, i. e., it needs further branching.

7.4 TU Based Branch-and-Bound Algorithm

Having a branching order and bound rule, our branch-and-bound procedure shapes. In this section, we list the algorithm step by step:

1. Solve the *linear relaxation* of the original problem.
   (a) If the *optimal objective value* $z > 0$, terminate with a conclusion that the problem is *unsatisfiable*.
   (b) If $z = 0$ and the solution is *integer*, the problem is *satisfiable*. Stop.
2. Sort variables according section 7.2.1, get the first part of the branching order.
3. $i \Leftarrow 1$
4. Choose the i-th variable x_j according to the order. Then assign value 0, 1 to it. Every assignment is corresponding to a *linear programming*.

   Solve the *linear programming* problem with $x_j = 1$ and $x_j = 0$.

   (a) If $z > 0$, prune this branch

(b) If $z = 0$ and the solution is an integer, the problem is *satisfiable*. The optimal solution of the linear programming is the feasible solution of the original problem. Stop.

(c) If $z = 0$, but the linear programming optimal solution is not integer, this node is still active. We enter its two children nodes into binary search tree with the $(i+1)th$ variable setting at 0 and 1, respectively.

5. Deal with the active nodes with width-first rule using the same branching strategy.

6. If all in being branching variables are used up, sort the rest variables according to section 7.2.2, and add all unsorted variables to the tail in a natural order. If no active node left, stop with a decision that the problem is unsatisfiable; otherwise, $i \Leftarrow i+1$, switch the search to the next layer, and go to step 4.

In this thesis, we use *ILOG Cplex* as a solver for all the linear relaxations of the subproblems. *ILOG Cplex* delivers high-performance, robust, flexible optimizers for solving linear, mixed-integer and quadratic programming problems in mission-critical resource allocation applications. CPLEX Callable Library provides a C application program interface (API) that allows all CPLEX features to be accessed from multiple programming languages. In this thesis, we use C language to implement the whole search procedure, including calling *ILOG Cplex* as a linear programming solver in C procedure.

7.5 Example

In Chapter 2, we have known that the randomly generated problems with a number of clauses/number of variables ratio close to the crossover point are very difficult to solve on average. Thus they are suitable to serve as benchmarks for tester/solvers comparison.

We use the program mkcnf.c (sparc executable suplied is mkcnf) to generate a random constant-clause-length CNF formula in Dimacs challenge format (number of variables: 10, number of clauses: 42): The first clause means $x_{10} \vee \bar{x}_7 \vee x_4$.

10	-7	4	-3	-10	-2
9	2	3	-4	-10	-6
-6	-3	1	4	3	-6
-1	-4	-10	8	1	9
-1	8	10	2	-5	1
-10	-7	-8	-7	-9	5
1	-8	5	10	-8	7
6	10	-2	-9	3	5
-8	-7	4	-4	-7	-10
3	-1	-7	-9	-8	-3
2	5	6	-10	8	-7
-3	-1	-8	-2	-9	5
4	6	7	-2	-6	8
5	-6	7	6	-1	-2
6	7	-9	4	-6	1
2	-9	-10	8	-1	-6
7	-1	-2	-7	5	8
-3	-4	2	7	-4	3
-9	-10	-8	-6	-10	-9
-3	-4	1	-1	9	-3
-7	-8	-9	6	-7	8

We convert the Dimacs challenge format to our integer programming model:

$$Min \quad \sum_{i=1}^{42} s_i$$

$$x_{10} - x_7 + x_4 + s_1 \geq 0 \qquad -x_3 - x_{10} - x_2 + s_2 \geq -2$$

$$x_9 + x_2 + x_3 + s_3 \geq 1 \qquad -x_4 - x_{10} - x_6 + s_4 \geq -2$$

$$-x_6 - x_3 + x_1 + s_5 \geq -1 \qquad x_4 + x_3 - x_6 + s_6 \geq 0$$

$$-x_1 - x_4 - x_{10} + s_7 \geq -2 \qquad x_8 + x_1 + x_9 + s_8 \geq 1$$

$$-x_1 + x_8 + x_{10} + s_9 \geq 0 \qquad x_2 - x_5 + x_1 + s_{10} \geq 0$$

$$-x_{10} - x_7 - x_8 + s_{11} \geq -2 \qquad -x_7 - x_9 + x_5 + s_{12} \geq -1$$

$$x_1 - x_8 + x_5 + s_{13} \geq 0 \qquad x_{10} - x_8 + x_7 + s_{14} \geq 0$$

$$x_6 + x_{10} - x_2 + s_{15} \geq 0 \qquad -x_9 + x_3 + x_5 + s_{16} \geq 0$$

$$-x_8 - x_7 + x_4 + s_{17} \geq -1 \qquad -x_4 - x_7 - x_{10} + s_{18} \geq -2$$

$$x_3 - x_1 - x_7 + s_{19} \geq -1 \qquad -x_9 - x_8 - x_3 + s_{20} \geq -2$$

$$x_2 + x_5 + x_6 + s_{21} \geq 1 \qquad -x_{10} + x_8 - x_7 + s_{22} \geq -1$$

$$-x_3 - x_1 - x_8 + s_{23} \geq -2 \qquad -x_2 - x_9 + x_5 + s_{24} \geq -1$$

$$x_4 + x_6 + x_7 + s_{25} \geq 1 \qquad -x_2 - x_6 + x_8 + s_{26} \geq -1$$

$$x_5 - x_6 + x_7 + s_{27} \geq 0 \qquad x_6 - x_1 - x_2 + s_{28} \geq -1$$

$$x_6 + x_7 - x_9 + s_{29} \geq 0 \qquad x_4 - x_6 + x_1 + s_{30} \geq 0$$

$$x_2 - x_9 - x_{10} + s_{31} \geq -1 \qquad x_8 - x_1 - x_6 + s_{32} \geq -1$$

$$x_7 - x_1 - x_2 + s_{33} \geq -1 \qquad -x_7 + x_5 + x_8 + s_{34} \geq 0$$

$$-x_3 - x_4 + x_2 + s_{35} \geq -1 \qquad x_7 - x_4 + x_3 + s_{36} \geq 0$$

$$-x_9 - x_{10} - x_8 + s_{37} \geq -2 \qquad -x_6 - x_{10} - x_9 + s_{38} \geq -2$$

$$-x_3 - x_4 + x_1 + s_{39} \geq -1 \qquad -x_1 + x_9 - x_3 + s_{40} \geq -1$$

$$-x_7 - x_8 - x_9 + s_{41} \geq -2 \qquad x_6 - x_7 + x_8 + s_{42} \geq 0$$

$$x_i \in \{0, 1\}, \qquad s_i \in \{0, 1\}$$

The constraint can be rewritten simply as $Ax + s \geq r$. Coefficient matrix A is a 0, ± 1 matrix in Appendix A. Now, we use our TU-based branch-and-bound algorithm to check whether this problem is satisfiable.

Step 1 Do linear programming for the relaxation of the original problem. The optimal objective value is 0, but the optimal solution is not integer-valued. We need to do branch-and-bound.

Step 2 Sort the variables. We compute the value of E_{ij} in Table 7.1 for every 2 columns (variables) in A:

E_{ij}	1	2	3	4	5	6	7	8	9	10
1	-	3×0	2×3	2×1	1×1	1×3	1×1	2×3	1×1	1×1
2	-	-	2×1	0×1	1×2	2×2	0×1	0×1	2×1	1×2
3	-	-	-	3×1	1×0	1×1	1×1	2×0	2×2	1×0
4	-	-	-	-	0×0	2×2	2×3	0×1	0×0	4×0
5	-	-	-	-	-	1×1	1×2	1×1	0×3	0×0
6	-	-	-	-	-	-	2×2	1×2	1×1	3×0
7	-	-	-	-	-	-	-	3×4	2×1	4×1
8	-	-	-	-	-	-	-	-	4×0	3×2
9	-	-	-	-	-	-	-	-	-	3×0

Table 7.1: Table E_{ij} for every pair columns

Where the first number in each lattice is the number of rows which have the pairs with the same signs, and the second number is the number of rows which have the pairs with different signs, the product is the value of E_{ij}.

Now the weight vector w_i in Table 7.2 can be calculated by formula:

$$w_i = \sum_{j>i} E_{ij} + \sum_{j<i} E_{ji}$$

The largest w_i is w_7, so the first branching variable should be x_7. We should revise w_i for cutting out the seventh column (x_7). We can get a revised w_i in

x_i	1	2	3	4	5	6	7	8	9	10
w_i	21	12	17	15	7	20	32	27	10	13

Table 7.2: Original table of variable weight w_i

x_i	1	2	3	4	5	6	7	8	9	10
w_i	20	12	16	9	5	16	0	15	8	9

Table 7.3: The first-revised table of variable weight w_i

Table 7.3 by:

$$w_i = \begin{cases} w_i - E_{i7} & if\ i < 7 \\ 0 & if\ i = 7 \\ w_i - E_{7i} & if\ i > 7 \end{cases}$$

The largest number in w_i is w_1, so the second branching variable is x_1.

Then, we revise w_i in Table 7.4. The largest number is w_6. So the third branching variable is x_6.

We continue to revise w_i in Table 7.5. The largest number in w_i is w_3. So the fourth branching variable is x_3.

Revision continues in Table 7.6. The largest number is w_{10}. So the fifth branching variable is x_{10}.

The fifth revision is given in Table 7.7. The largest number is w_2. So the sixth

x_i	1	2	3	4	5	6	7	8	9	10
w_i	0	12	10	7	4	13	-	9	7	8

Table 7.4: The second-revised table of variable weight w_i

x_i	1	2	3	4	5	6	7	8	9	10
w_i	-	8	9	3	3	0	-	7	6	8

Table 7.5: The third-revised table of variable weight w_i

x_i	1	2	3	4	5	6	7	8	9	10
w_i	-	6	0	0	3	-	-	7	2	8

Table 7.6: The fourth-revised table of variable weight w_i

branching variable is x_2.

The sixth revision is given in Table 7.8. The largest number is w_5. So the seventh branching variable is x_5.

The seventh revision is given in Table 7.9. Now, all the components of w_i are of non-positive values. i. e., all unsorted variables are incomparable in the sense of 2×2 submatrices. We have the variable order:

$$x_7 \rightarrow x_1 \rightarrow x_6 \rightarrow x_3 \rightarrow x_{10} \rightarrow x_2 \rightarrow x_5$$

Step 3 Initialization $i \Leftarrow 1$.

Step 4 Do linear programming for every nodes in the binary search tree.

The first node is $\{x_7 = 1\}$. We get the optimal objective value 0 of the linear relaxation by setting x_7 to 1, the optimal solution is not integral; the second

x_i	1	2	3	4	5	6	7	8	9	10
w_i	-	4	0	0	3	-	-	1	2	0

Table 7.7: The fifth-revised table of variable weight w_i

x_i	1	2	3	4	5	6	7	8	9	10
w_i	-	0	-	0	1	-	-	1	0	-

Table 7.8: The sixth-revised table of variable weight w_i

x_i	1	2	3	4	5	6	7	8	9	10
w_i	-	-	-	0	0	-	-	0	0	-

Table 7.9: The seventh-revised table of variable weight w_i

node is $\{x_7 = 0\}$. We get the optimal objective value 0, optimal solution is still fractional. Figure 7.3 depicts all the linear relaxations, once we get a positive optimal objective value at one node, we prune this branch (which is marked by a pair of shears in the figure). In this example, we find a satisfying solution using only the branching variable from the consideration of 2×2 non-totally unimodular submatrix. Empirically, most small satisfiable problems with the number of variables no more than 50 can be solved in the first phase. The information of each linear programming iteration including optimal objective value and optimal variable values is listed in Appendix B.

As depicted in Figure 7.3, only 15 linear programming (nodes) and 4 layers in a binary search tree are needed to find the satisfying truth assignment (which is marked by an asterisk in the figure), and its computation time is 0.01 second. Compared with the worst scenario of this problem (computation time $O(2^{10})$), our algorithm appears to be very promising.

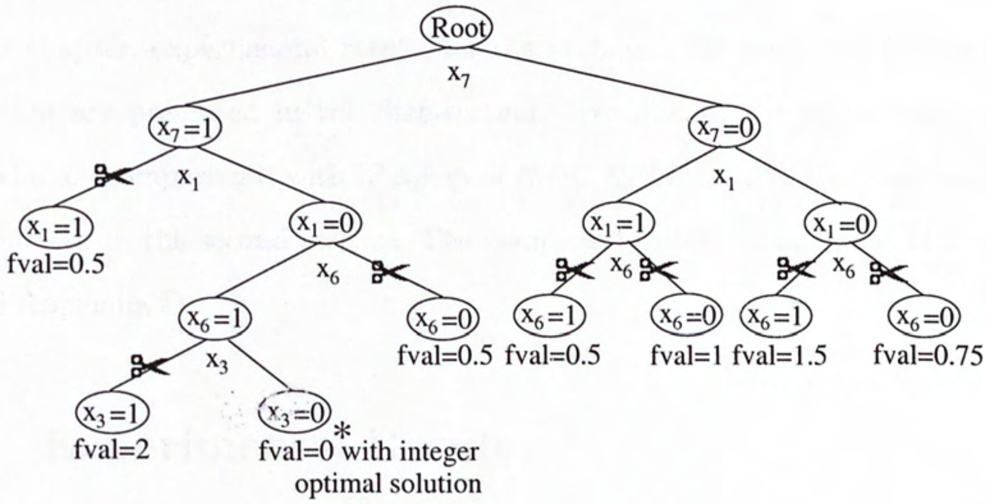

Figure 7.3: Binary Search Tree of the Example

Chapter 8

Numerical Result

In this chapter, experimental results for the proposed TU-based branch-and-bound algorithm are presented in the first section. We discuss the performance of this algorithm by comparing it with IP solver of *ILOG CPLEX* under the same computing environment in the second section. The complete results can be found in Appendix C and Appendix D.

8.1 Experimental Result

We use Uniform Random-3-SAT as the main test-set for our algorithm. Uniform Random-3-SAT is a family of SAT problems obtained by randomly generating 3-CNF formulae in the following way: For an instance with n variables and k clauses, each of the k clauses is constructed from 3 literals which are randomly drawn from the $2n$ possible literals (the n variables and their negations) such that each possible literal is selected with the same probability of $1/2n$. Clauses are not accepted for the construction of the problem instance if they contain multiple copies of the same literal or if they are tautological (i.e., they contain a variable and its negation as a literal). Each choice of n and k thus induces a distribution of Random-3-SAT instances. Uniform Random-3-SAT is the union of these distributions over all n and k. The test-sets provided in `http://www.intellektik.informatik.tu-darmstadt.de /SATLIB/Benchmarks/SAT/RND3SAT/descr.html` are sampled from the phase transition region of uniform Random 3-SAT. We use the test-sets for n =20, 50, 75 and 100.

Table 8.1, Table 8.2, Table 8.3 and Table 8.4 list the mean, standard deviation, minimal value and maximal value of the completion time, the number of layers and nodes being searched in the binary search tree for n=20, n=50, n=75 and n=100 on 100 samples, respectively.

	# of Layers	# of Nodes	Completion Time (sec.)
Mean	2.82	10.89	0.0228
S.D.	1.6229	8.3906	0.0136
Min	0	1	0.0000
Max	6	37	0.0600

Table 8.1: Statistical Result of TU-based B&B for n=20, m=91

	# of Layers	# of Nodes	Completion Time (sec.)
Mean	6.13	92.83	0.3931
S.D.	2.5172	76.4462	0.2736
Min	0	1	0.0400
Max	15	343	1.31

Table 8.2: Statistical Result of TU-based B&B for n=50, m=218

When $n \leq 100$, all of the selected randomly generated 3-SAT instances can be solved by the TU-based branch-and-bound algorithm within 3 minutes.

	# of Layers	# of Nodes	Completion Time (sec.)
Mean	9.11	703.98	4.858
S.D.	3.1555	1008.597	5.6417
Min	3	8	0.07
Max	18	6502	37.07

Table 8.3: Statistical Result of TU-based B&B for n=75, m=325

	# of Layers	# of Nodes	Completion Time (sec.)
Mean	11.52	3165.91	37.1802
S.D.	3.6362	3316.752	35.8724
Min	3	12	0.42
Max	19	17478	164.44

Table 8.4: Statistical Result of TU-based B&B for n=100, m=430

8.2 Statistical Results of *ILOG CPLEX*

For the sake of comparison, we use the IP solver of *ILOG CPLEX* to solve the same problems under the same computing environment.

Table 8.5 lists the mean, variance, minimal value and maximal value of the completion time on 100 samples using *CPLEX* IP solver. Table 8.6 lists the mean, variance, minimal value and maximal value of the nodes used to solve the problem by *CPLEX* IP solver. *CPLEX* excels our algorithm only at 3 indexes (denotes by an asterisk).

	n=20, m=91	n=50, m=218	n=75, m=325	n=100, m=430
Mean	0.0394	0.5842	5.5548	39.8416
S.D.	0.01994	0.4614	6.2550	40.0356
Min	0	0.02 *	0.07	1.18
Max	0.09	2.46	37.36	222.64

Table 8.5: Completion Time of *CPLEX* IP Solver

	n=20, m=91	n=50, m=218	n=75, m=325	n=100, m=430
Mean	9.2698 *	144.5618	964.9897	4139.81
S.D.	8.8831	149.6657	1498.664	4622.361
Min	1	1	1 *	15
Max	41	692	12472	27900

Table 8.6: Number of Nodes Used for *CPLEX* IP Solver

A complete numerical results for *CPLEX* can be found in Appendix D.

Table 8.7 lists the gain of our approach on 3 important indexes: mean number of nodes being searched, mean completion time and maximum number of nodes being searched among those 100 sample instances against *CPLEX* IP solver. Clearly, our approach outperforms *CPLEX* IP solver.

Gain in	n=20,m=91	n=50,m=218	n=75,m=325	n=100,m=430
Mean no. of nodes	-15%	56%	37%	31%
Mean time	73%	49%	14%	7%
Max no. of nodes	11%	102%	92%	60%

Table 8.7: The gain of TU-based B&B on *CPLEX* IP solver

Chapter 9

Conclusions

This chapter summarizes our research contributions and discusses potential future work.

9.1 Contributions

The main contributions of this research are as follows:

1. We actively use the Totally Unimodular theory to solve SAT problems. In literature, people usually pay attention to the solvability of the problem if the constraint matrix has already been totally unimodular, and how to check the totally unimodularity of the constraint matrix. In many situations, however, the constraint matrix of an integer optimization problem is not totally unimodular. Our research can thus deal with general problems without any assumption of totally unimodularity.

2. We find the common characteristics for all non-totally unimodular 2×2 $(0, \pm 1)$ and 3×3 0-1 matrices, and prove the form exclusiveness of the non-totally unimodular $3{\times}3$ 0-1 matrices. See Theorem 6.3.2.

3. Based on the matrix research results, we derive an efficient algorithm to make the constraint matrix closer to the state of totally unimodularity step by step. It can also be regarded as the process of approaching a solvable state by linear relaxation.

4. Through the comparison with *ILOG CPLEX* – a very powerful solver for optimization problems, we find our algorithm is very efficient when n is not more than 100.

5. This algorithm can be also extended to situations where the decision variables are not binary and the components of the coefficient matrix take values from -1, 0, and 1.

9.2 Future Work

One obvious extension of this research would be to study the common characteristics of non-totally unimodularity for high-order square matrixces (we only studied 2×2 and 3×3 matrices). This will help us to get more efficient branching order for branch-and-bound process.

In fact, after we exclude the 3×3 non-totally unimodular submatrices, we have noticed that, for 0,1 square matrix, if there are at most 3 1s in every row, all even-sized matrices are totally unimodular(e.g. 4×4, 6×6, $\cdots$). In the case of odd-sized matrix, when there are exactly two 1s per row and per column, the matrix is non-totally unimodular.

Despite the good performance on many instances, the proposed TU-based branch-and-bound algorithm does not have a very good performance on many other large-sized problems. Especially, for unsatisfiable SAT problem, our algorithm is not so good because we must search the entire binary tree to get such unsatisfiability conclusion. It could take months or even years to solve some of them. And, optimizing the variable sorting part is another extension of this research.

Appendix A

The Coefficient Matrix A for Example in Chapter 7

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
0	0	0	1	0	0	-1	0	0	1
0	-1	-1	0	0	0	0	0	0	-1
0	1	1	0	0	0	0	0	1	0
0	0	0	-1	0	-1	0	0	0	-1
1	0	-1	0	0	-1	0	0	0	0
0	0	1	1	0	-1	0	0	0	0
-1	0	0	-1	0	0	0	0	0	-1
1	0	0	0	0	0	0	1	1	0
-1	0	0	0	0	0	0	1	0	1
1	1	0	0	-1	0	0	0	0	0
0	0	0	0	0	0	-1	-1	0	-1
0	0	0	0	1	0	-1	0	-1	0
1	0	0	0	1	0	0	-1	0	0
0	0	0	0	0	0	1	-1	0	1
0	-1	0	0	0	1	0	0	0	1
0	0	1	0	1	0	0	0	-1	0
0	0	0	1	0	0	-1	-1	0	0
0	0	0	-1	0	0	-1	0	0	-1
-1	0	1	0	0	0	-1	0	0	0
0	0	-1	0	0	0	0	-1	-1	0
0	1	0	0	1	1	0	0	0	0
0	0	0	0	0	0	-1	1	0	-1

cont'd

-1	0	-1	0	0	0	0	-1	0	0
0	-1	0	0	1	0	0	0	-1	0
0	0	0	1	0	1	1	0	0	0
0	-1	0	0	0	-1	0	1	0	0
0	0	0	0	1	-1	1	0	0	0
-1	-1	0	0	0	1	0	0	0	0
0	0	0	0	0	1	1	0	-1	0
1	0	0	1	0	-1	0	0	0	0
0	1	0	0	0	0	0	0	-1	-1
-1	0	0	0	0	-1	0	1	0	0
-1	-1	0	0	0	0	1	0	0	0
0	0	0	0	1	0	-1	1	0	0
0	1	-1	-1	0	0	0	0	0	0
0	0	1	-1	0	0	1	0	0	0
0	0	0	0	0	0	0	-1	-1	-1
0	0	0	0	0	-1	0	0	-1	-1
1	0	-1	-1	0	0	0	0	0	0
-1	0	-1	0	0	0	0	0	1	0
0	0	0	0	0	0	-1	-1	-1	0
0	0	0	0	0	1	-1	1	0	0

Appendix B

The Detailed Numerical Information of Solution Process for Example in Chapter 7

fval	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
0.000	0.143	0.571	0.000	0.714	0.714	0.571	0.714	0.143	0.714	0.000
0.000	0.000	0.667	0.000	1.000	0.667	0.667	1.000	0.333	0.667	0.000
0.000	0.222	0.444	0.444	0.444	0.667	0.667	0.000	0.111	0.667	0.111
0.500	1.000	0.500	0.500	0.500	0.500	0.500	1.000	0.500	0.500	0.500
0.000	0.000	0.667	0.000	1.000	0.667	0.667	1.000	0.333	0.667	0.000
0.000	1.000	0.000	0.500	0.500	0.500	0.500	0.000	0.500	0.500	0.500
0.000	0.000	1.000	0.500	0.500	0.500	0.500	0.000	0.500	0.500	0.500
0.000	0.000	0.500	0.000	1.000	0.500	1.000	1.000	0.500	0.000	0.000
0.500	0.000	0.500	0.500	0.500	0.500	0.000	1.000	0.500	0.500	0.500
0.500	1.000	0.000	0.500	0.500	1.000	1.000	0.000	0.500	0.500	0.500
1.000	1.000	0.000	0.500	0.500	1.000	0.000	0.000	0.500	0.500	0.500
1.500	0.000	0.750	0.500	0.500	1.000	1.000	0.000	0.750	0.250	0.750
0.750	0.000	0.750	0.500	0.500	0.750	0.000	0.000	0.750	0.000	0.750
2.000	0.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.000	0.000
0.000	0.000	1.000	0.000	1.000	1.000	1.000	1.000	1.000	0.000	0.000

Appendix C

Experimental Result

C.1 # of variables: 20, # of clauses: 91

Problem Name	Number of Layers	Number of Nodes	Completion Time(sec.)
uf20-01.cnf	3	10	0.0200
uf20-010.cnf	2	4	0.0200
uf20-0100.cnf	4	22	0.0400
uf20-01000.cnf	5	20	0.0400
uf20-0101.cnf	2	5	0.0200
uf20-0102.cnf	3	10	0.0200
uf20-0103.cnf	4	22	0.0400
uf20-0104.cnf	2	6	0.0200
uf20-0105.cnf	5	20	0.0400
uf20-0106.cnf	2	6	0.0200
uf20-0107.cnf	6	6	0.0400
uf20-0108.cnf	3	10	0.0100
uf20-0109.cnf	6	25	0.0400
uf20-011.cnf	1	3	0.0100
uf20-0110.cnf	2	4	0.0200
uf20-0111.cnf	4	17	0.0200
uf20-0112.cnf	3	10	0.0100
uf20-0113.cnf	3	11	0.0300
uf20-0114.cnf	2	4	0.0000
uf20-0115.cnf	3	15	0.0200

Problem Name	Number of Layers	Number of Nodes	Completion Time(sec.)
uf20-0116.cnf	4	19	0.0300
uf20-0117.cnf	4	22	0.0300
uf20-0118.cnf	5	24	0.0400
uf20-0119.cnf	4	16	0.0400
uf20-012.cnf	1	3	0.0000
uf20-0120.cnf	2	4	0.0200
uf20-0121.cnf	3	8	0.0100
uf20-0122.cnf	5	4	0.0300
uf20-0123.cnf	3	10	0.0300
uf20-0124.cnf	2	6	0.0100
uf20-0125.cnf	5	18	0.0400
uf20-0126.cnf	0	1	0.0000
uf20-0127.cnf	1	3	0.0000
uf20-0128.cnf	3	9	0.0300
uf20-0129.cnf	2	4	0.0100
uf20-013.cnf	0	1	0.0000
uf20-0130.cnf	0	1	0.0000
uf20-0131.cnf	1	2	0.0000
uf20-0132.cnf	3	14	0.0300
uf20-0133.cnf	4	16	0.0300
uf20-0134.cnf	4	22	0.0400
uf20-0135.cnf	2	5	0.0200
uf20-0136.cnf	6	28	0.0500
uf20-0137.cnf	0	1	0.0000
uf20-0138.cnf	3	11	0.0200
uf20-0139.cnf	0	1	0.0000
uf20-014.cnf	3	9	0.0100
uf20-0140.cnf	3	13	0.0300
uf20-0141.cnf	4	21	0.0300
uf20-0142.cnf	4	14	0.0300
uf20-0143.cnf	1	2	0.0200
uf20-0144.cnf	0	1	0.0000
uf20-0145.cnf	6	22	0.0300
uf20-0146.cnf	4	24	0.0300

Problem Name	Number of Layers	Number of Nodes	Completion Time(sec.)
uf20-0147.cnf	3	8	0.0100
uf20-0148.cnf	3	9	0.0200
uf20-0149.cnf	4	20	0.0300
uf20-015.cnf	1	3	0.0100
uf20-0150.cnf	4	14	0.0400
uf20-0151.cnf	3	9	0.0100
uf20-0152.cnf	0	1	0.0200
uf20-0153.cnf	2	4	0.0100
uf20-0154.cnf	4	14	0.0300
uf20-0155.cnf	6	37	0.0500
uf20-0156.cnf	2	6	0.0200
uf20-0157.cnf	0	1	0.0100
uf20-0158.cnf	0	1	0.0000
uf20-0159.cnf	5	30	0.0300
uf20-016.cnf	3	8	0.0300
uf20-0160.cnf	2	6	0.0200
uf20-0161.cnf	3	10	0.0200
uf20-0162.cnf	5	19	0.0400
uf20-0163.cnf	1	2	0.0200
uf20-0164.cnf	0	1	0.0100
uf20-0165.cnf	2	6	0.0300
uf20-0166.cnf	2	6	0.0300
uf20-0167.cnf	1	2	0.0000
uf20-0168.cnf	3	12	0.0300
uf20-0169.cnf	1	2	0.0200
uf20-017.cnf	4	18	0.0300
uf20-0170.cnf	3	8	0.0100
uf20-0171.cnf	1	3	0.0200
uf20-0172.cnf	0	1	0.0200
uf20-0173.cnf	5	26	0.0500
uf20-0174.cnf	2	6	0.0200
uf20-0175.cnf	5	23	0.0400
uf20-0176.cnf	4	22	0.0300
uf20-0177.cnf	3	9	0.0200

Problem Name	Number of Layers	Number of Nodes	Completion Time(sec.)
uf20-0178.cnf	2	5	0.0100
uf20-0179.cnf	3	8	0.0300
uf20-018.cnf	4	19	0.0300
uf20-0180.cnf	4	19	0.0400
uf20-0181.cnf	5	33	0.0600
uf20-0182.cnf	2	7	0.0200
uf20-0183.cnf	4	16	0.0300
uf20-0184.cnf	3	11	0.0300
uf20-0185.cnf	3	12	0.0200
uf20-0186.cnf	2	6	0.0300
uf20-0187.cnf	2	4	0.0100
uf20-0188.cnf	4	13	0.0200

C.2 # of variables: 50, # of clauses: 218

Problem Name	Number of Layers	Number of Nodes	Completion Time(sec.)
uf50-01.cnf	8	198	0.7300
uf50-010.cnf	5	42	0.1900
uf50-0100.cnf	10	192	0.8200
uf50-01000.cnf	4	27	0.1400
uf50-0101.cnf	7	118	0.5800
uf50-0102.cnf	6	56	0.2500
uf50-0103.cnf	6	59	0.2700
uf50-0104.cnf	7	105	0.4600
uf50-0105.cnf	3	9	0.0800
uf50-0106.cnf	6	64	0.3100
uf50-0107.cnf	8	132	0.5100
uf50-0108.cnf	6	51	0.2700
uf50-0109.cnf	0	1	0.0400
uf50-011.cnf	9	211	0.8900
uf50-0110.cnf	9	180	0.6900
uf50-0111.cnf	8	109	0.4800

Problem Name	Number of Layers	Number of Nodes	Completion Time(sec.)
uf50-0112.cnf	10	243	0.8700
uf50-0113.cnf	7	196	0.6300
uf50-0114.cnf	5	46	0.2100
uf50-0115.cnf	2	7	0.0800
uf50-0116.cnf	4	18	0.0900
uf50-0117.cnf	5	32	0.1600
uf50-0118.cnf	8	167	0.5800
uf50-0119.cnf	5	54	0.2800
uf50-012.cnf	4	18	0.1000
uf50-0120.cnf	6	62	0.2900
uf50-0121.cnf	6	52	0.2800
uf50-0122.cnf	4	19	0.1000
uf50-0123.cnf	8	216	0.7600
uf50-0124.cnf	8	238	0.7300
uf50-0125.cnf	7	112	0.4700
uf50-0126.cnf	5	56	0.2100
uf50-0127.cnf	4	16	0.1000
uf50-0128.cnf	7	168	0.6700
uf50-0129.cnf	14	256	0.9900
uf50-013.cnf	5	42	0.2400
uf50-0130.cnf	6	70	0.3700
uf50-0131.cnf	3	12	0.1000
uf50-0132.cnf	4	26	0.1600
uf50-0133.cnf	5	46	0.2600
uf50-0134.cnf	10	169	0.8500
uf50-0135.cnf	6	66	0.2900
uf50-0136.cnf	0	1	0.0400
uf50-0137.cnf	7	74	0.3200
uf50-0138.cnf	5	45	0.1600
uf50-0139.cnf	6	68	0.3400
uf50-014.cnf	6	77	0.3400
uf50-0140.cnf	7	111	0.4900
uf50-0141.cnf	6	80	0.3500
uf50-0142.cnf	9	143	0.6000

Problem Name	Number of Layers	Number of Nodes	Completion Time(sec.)
uf50-0143.cnf	6	74	0.3400
uf50-0144.cnf	5	46	0.2700
uf50-0145.cnf	6	75	0.2900
uf50-0146.cnf	6	75	0.3500
uf50-0147.cnf	7	116	0.5500
uf50-0148.cnf	10	180	0.7500
uf50-0149.cnf	3	12	0.0700
uf50-015.cnf	5	39	0.2200
uf50-0150.cnf	6	82	0.3300
uf50-0151.cnf	7	188	0.7500
uf50-0152.cnf	6	81	0.3900
uf50-0153.cnf	6	94	0.3900
uf50-0154.cnf	8	228	0.6800
uf50-0155.cnf	6	66	0.3000
uf50-0156.cnf	5	38	0.2100
uf50-0157.cnf	5	38	0.1500
uf50-0158.cnf	7	118	0.5200
uf50-0159.cnf	7	83	0.3600
uf50-016.cnf	5	62	0.2400
uf50-0160.cnf	7	98	0.4200
uf50-0161.cnf	5	36	0.1600
uf50-0162.cnf	4	20	0.1200
uf50-0163.cnf	15	343	1.1800
uf50-0164.cnf	0	1	0.0500
uf50-0165.cnf	6	72	0.3600
uf50-0166.cnf	3	35	0.1900
uf50-0167.cnf	9	207	0.8500
uf50-0168.cnf	4	17	0.1200
uf50-0169.cnf	6	78	0.4200
uf50-017.cnf	6	100	0.4400
uf50-0170.cnf	5	37	0.1800
uf50-0171.cnf	7	118	0.4900
uf50-0172.cnf	9	329	1.3100
uf50-0173.cnf	10	162	0.6700

Problem Name	Number of Layers	Number of Nodes	Completion Time(sec.)
uf50-0174.cnf	7	112	0.4800
uf50-0175.cnf	3	12	0.0900
uf50-0176.cnf	9	203	0.7500
uf50-0177.cnf	10	229	0.8900
uf50-0178.cnf	2	7	0.0400
uf50-0179.cnf	6	55	0.3000
uf50-018.cnf	2	5	0.0700
uf50-0180.cnf	11	240	0.9100
uf50-0181.cnf	4	23	0.1500
uf50-0182.cnf	8	102	0.4400
uf50-0183.cnf	4	26	0.1600
uf50-0184.cnf	4	19	0.1000
uf50-0185.cnf	7	118	0.5100
uf50-0186.cnf	6	60	0.3500
uf50-0187.cnf	7	123	0.5100
uf50-0188.cnf	5	41	0.2200

C.3 # of variables: 75, # of clauses: 325

Problem Name	Number of Layers	Number of Nodes	Completion Time(sec.)
uf75-01.cnf	8	253	2.2800
uf75-010.cnf	3	8	0.2000
uf75-0100.cnf	13	1578	9.4100
uf75-011.cnf	9	560	4.6700
uf75-012.cnf	13	1526	9.0600
uf75-013.cnf	6	84	0.6900
uf75-014.cnf	8	398	2.7100
uf75-015.cnf	11	628	4.5200
uf75-016.cnf	6	80	0.8900
uf75-017.cnf	6	96	0.5700
uf75-018.cnf	4	18	0.2800
uf75-019.cnf	10	535	4.1000

Problem Name	Number of Layers	Number of Nodes	Completion Time(sec.)
uf75-02.cnf	11	1288	8.6100
uf75-020.cnf	14	1676	12.1500
uf75-021.cnf	10	792	5.6800
uf75-022.cnf	10	366	3.2600
uf75-023.cnf	12	1054	7.2000
uf75-024.cnf	12	930	6.2200
uf75-025.cnf	6	107	0.7400
uf75-026.cnf	12	1715	11.1900
uf75-027.cnf	7	136	1.0100
uf75-028.cnf	9	397	2.8600
uf75-029.cnf	7	188	1.6900
uf75-03.cnf	13	910	7.2000
uf75-030.cnf	11	1020	6.3800
uf75-031.cnf	8	256	1.9800
uf75-032.cnf	7	214	1.7900
uf75-033.cnf	6	103	0.8800
uf75-034.cnf	9	466	2.7300
uf75-035.cnf	8	252	1.7600
uf75-036.cnf	8	323	2.5400
uf75-037.cnf	15	6502	37.0700
uf75-038.cnf	8	316	3.0200
uf75-039.cnf	12	1672	12.5100
uf75-04.cnf	8	281	3.0100
uf75-040.cnf	9	354	2.8300
uf75-041.cnf	10	448	3.7400
uf75-042.cnf	9	633	4.6300
uf75-043.cnf	11	517	4.1200
uf75-044.cnf	7	145	1.5700
uf75-045.cnf	5	51	0.4300
uf75-046.cnf	12	1038	7.8200
uf75-047.cnf	7	204	1.6000
uf75-048.cnf	12	1321	9.1300
uf75-049.cnf	7	207	1.6300
uf75-05.cnf	12	2305	16.0100

Problem Name	Number of Layers	Number of Nodes	Completion Time(sec.)
uf75-050.cnf	3	10	0.2100
uf75-051.cnf	18	5341	24.6300
uf75-052.cnf	10	746	6.1000
uf75-053.cnf	9	327	2.7100
uf75-054.cnf	6	94	0.9900
uf75-055.cnf	10	741	5.7700
uf75-056.cnf	10	471	3.5600
uf75-057.cnf	14	946	7.3500
uf75-058.cnf	10	668	4.8400
uf75-059.cnf	6	125	0.8100
uf75-06.cnf	12	1858	13.0900
uf75-060.cnf	7	173	1.6900
uf75-061.cnf	5	35	0.3900
uf75-062.cnf	8	302	2.9200
uf75-063.cnf	9	333	2.5300
uf75-064.cnf	8	245	1.7900
uf75-065.cnf	10	720	5.0400
uf75-066.cnf	10	727	5.4400
uf75-067.cnf	7	202	1.6600
uf75-068.cnf	9	379	2.8100
uf75-069.cnf	8	343	2.5900
uf75-07.cnf	6	85	0.8800
uf75-070.cnf	13	851	7.2500
uf75-071.cnf	11	1808	12.7500
uf75-072.cnf	8	255	2.2500
uf75-073.cnf	9	559	3.4100
uf75-074.cnf	8	235	2.4100
uf75-075.cnf	10	770	7.0100
uf75-076.cnf	17	2036	14.1900
uf75-077.cnf	5	38	0.4700
uf75-078.cnf	6	108	1.0700
uf75-079.cnf	17	1164	8.6100
uf75-08.cnf	9	626	4.8000
uf75-080.cnf	4	19	0.29000

Problem Name	Number of Layers	Number of Nodes	Completion Time(sec.)
uf75-081.cnf	7	142	1.2400
uf75-082.cnf	10	995	7.5300
uf75-083.cnf	8	248	1.8500
uf75-084.cnf	10	306	2.8000
uf75-085.cnf	3	13	0.0700
uf75-086.cnf	16	4705	25.5100
uf75-087.cnf	3	14	0.2200
uf75-088.cnf	6	112	1.0500
uf75-089.cnf	8	302	2.2500
uf75-09.cnf	5	33	0.4100
uf75-090.cnf	13	1168	8.3700
uf75-091.cnf	12	476	3.6100
uf75-092.cnf	8	352	2.8000
uf75-093.cnf	8	236	1.8700
uf75-094.cnf	9	358	2.1500
uf75-095.cnf	7	202	2.0200
uf75-096.cnf	11	1086	7.2100
uf75-097.cnf	15	866	7.5100
uf75-098.cnf	11	1146	8.0800
uf75-099.cnf	8	257	2.5500

C.4 # of variables: 100, # of clauses: 430

Problem Name	Number of Layers	Number of Nodes	Completion Time(sec.)
uf100-01.cnf	10	1694	20.7900
uf100-010.cnf	14	4471	52.6700
uf100-0100.cnf	6	99	1.4400
uf100-01000.cnf	9	750	12.7300
uf100-0101.cnf	10	999	15.7400
uf100-0102.cnf	15	3323	48.5700
uf100-0103.cnf	14	4869	62.2300
uf100-0104.cnf	18	17478	164.4400

Problem Name	Number of Layers	Number of Nodes	Completion Time(sec.)
uf100-0105.cnf	9	657	10.5500
uf100-0106.cnf	15	7563	96.1100
uf100-0107.cnf	6	95	1.4900
uf100-0108.cnf	16	6762	75.2600
uf100-0109.cnf	9	826	11.7500
uf100-011.cnf	15	6790	80.6000
uf100-0110.cnf	14	8361	93.4500
uf100-0111.cnf	11	2012	24.7500
uf100-0112.cnf	12	1674	20.6800
uf100-0113.cnf	16	9511	101.7100
uf100-0114.cnf	13	3079	41.3300
uf100-0115.cnf	8	263	3.6000
uf100-0116.cnf	12	1931	25.4700
uf100-0117.cnf	10	933	12.9000
uf100-0118.cnf	6	101	1.2200
uf100-0119.cnf	7	185	2.1400
uf100-012.cnf	14	2978	31.6000
uf100-0120.cnf	8	403	5.9900
uf100-0121.cnf	15	11854	124.8900
uf100-0122.cnf	12	2342	26.2700
uf100-0123.cnf	13	4206	51.4400
uf100-0124.cnf	5	68	0.8900
uf100-0125.cnf	10	1157	17.3000
uf100-0126.cnf	17	8710	110.5600
uf100-0127.cnf	19	10210	115.3500
uf100-0128.cnf	9	561	8.4400
uf100-0129.cnf	14	2976	37.6500
uf100-013.cnf	15	3998	50.7300
uf100-0130.cnf	16	9555	101.0200
uf100-0131.cnf	11	2315	29.1700
uf100-0132.cnf	14	5090	70.5900
uf100-0133.cnf	15	12727	145.8600
uf100-0134.cnf	13	4736	58.1100
uf100-0135.cnf	13	1568	21.1000

Problem Name	Number of Layers	Number of Nodes	Completion Time(sec.)
uf100-0136.cnf	8	354	5.2400
uf100-0137.cnf	11	1696	25.3100
uf100-0138.cnf	7	164	1.9600
uf100-0139.cnf	12	2842	29.8800
uf100-014.cnf	14	3807	39.9900
uf100-0140.cnf	11	2308	26.0400
uf100-0141.cnf	13	3713	52.6900
uf100-0142.cnf	9	679	9.0500
uf100-0143.cnf	13	3128	39.4400
uf100-0144.cnf	11	1732	23.0200
uf100-0145.cnf	16	4380	54.6300
uf100-0146.cnf	11	1072	15.6100
uf100-0147.cnf	6	126	2.1200
uf100-0148.cnf	16	3349	45.5500
uf100-0149.cnf	10	1540	18.7500
uf100-015.cnf	11	1588	18.6700
uf100-0150.cnf	9	832	8.3800
uf100-0151.cnf	5	41	0.8600
uf100-0152.cnf	14	5774	73.6200
uf100-0153.cnf	12	3473	36.8800
uf100-0154.cnf	19	10995	122.7200
uf100-0155.cnf	12	3136	37.3200
uf100-0156.cnf	11	1584	20.8500
uf100-0157.cnf	18	2038	25.7500
uf100-0158.cnf	15	8964	92.8700
uf100-0159.cnf	14	3269	35.1800
uf100-016.cnf	18	7376	75.8500
uf100-0160.cnf	14	5451	52.2400
uf100-0161.cnf	10	1117	15.9000
uf100-0162.cnf	6	95	1.4900
uf100-0163.cnf	14	6066	66.7200
uf100-0164.cnf	5	32	0.4700
uf100-0165.cnf	10	1052	12.4600
uf100-0166.cnf	12	3099	35.7400

Problem Name	Number of Layers	Number of Nodes	Completion Time(sec.)
uf100-0167.cnf	12	2795	34.8200
uf100-0168.cnf	14	3698	41.9000
uf100-0169.cnf	11	1143	16.4100
uf100-017.cnf	15	7271	83.0200
uf100-0170.cnf	8	275	3.5700
uf100-0171.cnf	14	5856	77.1900
uf100-0172.cnf	11	1803	21.5400
uf100-0173.cnf	14	3306	43.5000
uf100-0174.cnf	12	2438	31.6900
uf100-0175.cnf	10	1423	21.7900
uf100-0176.cnf	10	1139	13.0900
uf100-0177.cnf	13	3467	32.2400
uf100-0178.cnf	11	1286	16.0900
uf100-0179.cnf	11	1029	16.2500
uf100-018.cnf	5	37	0.7100
uf100-0180.cnf	17	5853	65.5500
uf100-0181.cnf	8	436	5.9600
uf100-0182.cnf	4	18	0.4200
uf100-0183.cnf	13	4334	61.1800
uf100-0184.cnf	12	1725	21.9100
uf100-0185.cnf	5	55	0.9900
uf100-0186.cnf	5	41	0.5800
uf100-0187.cnf	9	399	5.3400
uf100-0188.cnf	3	12	0.4500

Appendix D

Experimental Result of ILOG CPLEX

D.1 # of variables: 20, # of clauses: 91

Problem Name	Number of Nodes	Completion Time(sec.)
uf20-01.cnf	-[1]	0.0300
uf20-010.cnf	14	0.0600
uf20-0100.cnf	-	0.0100
uf20-01000.cnf	7	0.0500
uf20-0101.cnf	10	0.0700
uf20-0102.cnf	17	0.0600
uf20-0103.cnf	11	0.0500
uf20-0104.cnf	-	0.0200
uf20-0105.cnf	41	0.0900
uf20-0106.cnf	13	0.0600
uf20-0107.cnf	17	0.0500
uf20-0108.cnf	12	0.0700
uf20-0109.cnf	2	0.0400
uf20-011.cnf	2	0.0400
uf20-0110.cnf	-	0.0200
uf20-0111.cnf	6	0.0600
uf20-0112.cnf	-	0.0200

[1] Here the mark "-" means no branch-and-bound algorithm used in solving this integer programming problem

Problem Name	Number of Nodes	Completion Time(sec.)
uf20-0113.cnf	14	0.0500
uf20-0114.cnf	17	0.0600
uf20-0115.cnf	-	0.0200
uf20-0116.cnf	12	0.0500
uf20-0117.cnf	14	0.0600
uf20-0118.cnf	-	0.0200
uf20-0119.cnf	27	0.0700
uf20-012.cnf	-	0.0200
uf20-0120.cnf	-	0.0300
uf20-0121.cnf	2	0.0500
uf20-0122.cnf	5	0.0400
uf20-0123.cnf	3	0.0400
uf20-0124.cnf	-	0.0100
uf20-0125.cnf	4	0.0400
uf20-0126.cnf	-	0.0100
uf20-0127.cnf	3	0.0500
uf20-0128.cnf	3	0.0300
uf20-0129.cnf	2	0.0200
uf20-013.cnf	-	0.0100
uf20-0130.cnf	-	0.0100
uf20-0131.cnf	-	0.0300
uf20-0132.cnf	2	0.0600
uf20-0133.cnf	-	0.0200
uf20-0134.cnf	-	0.0300
uf20-0135.cnf	-	0.0300
uf20-0136.cnf	32	0.0900
uf20-0137.cnf	-	0.0400
uf20-0138.cnf	5	0.0500
uf20-0139.cnf	-	0.0200
uf20-014.cnf	14	0.0600
uf20-0140.cnf	3	0.0600
uf20-0141.cnf	13	0.0500
uf20-0142.cnf	-	0.0400
uf20-0143.cnf	-	0.0200

Problem Name	Number of Nodes	Completion Time(sec.)
uf20-0144.cnf	-	0.0100
uf20-0145.cnf	4	0.0400
uf20-0146.cnf	3	0.0400
uf20-0147.cnf	2	0.0300
uf20-0148.cnf	2	0.0500
uf20-0149.cnf	-	0.0300
uf20-015.cnf	-	0.0100
uf20-0150.cnf	1	0.0500
uf20-0151.cnf	3	0.0500
uf20-0152.cnf	-	0.0000
uf20-0153.cnf	-	0.0300
uf20-0154.cnf	3	0.0400
uf20-0155.cnf	3	0.0500
uf20-0156.cnf	2	0.0300
uf20-0157.cnf	-	0.0100
uf20-0158.cnf	-	0.0000
uf20-0159.cnf	9	0.0400
uf20-016.cnf	29	0.0700
uf20-0160.cnf	3	0.0400
uf20-0161.cnf	30	0.0400
uf20-0162.cnf	-	0.0100
uf20-0163.cnf	4	0.0400
uf20-0164.cnf	-	0.0100
uf20-0165.cnf	2	0.0400
uf20-0166.cnf	3	0.0400
uf20-0167.cnf	2	0.0400
uf20-0168.cnf	2	0.0400
uf20-0169.cnf	-	0.0200
uf20-017.cnf	12	0.0700
uf20-0170.cnf	15	0.0700
uf20-0171.cnf	-	0.0200
uf20-0172.cnf	-	0.0100
uf20-0173.cnf	5	0.0500
uf20-0174.cnf	2	0.0600

Problem Name	Number of Nodes	Completion Time(sec.)
uf20-0175.cnf	2	0.0500
uf20-0176.cnf	24	0.0700
uf20-0177.cnf	8	0.0400
uf20-0178.cnf	18	0.0600
uf20-0179.cnf	22	0.0700
uf20-018.cnf	11	0.0500
uf20-0180.cnf	5	0.0500
uf20-0181.cnf	-	0.0200
uf20-0182.cnf	-	0.0200
uf20-0183.cnf	3	0.0400
uf20-0184.cnf	-	0.0300
uf20-0185.cnf	-	0.0300
uf20-0186.cnf	12	0.0600
uf20-0187.cnf	2	0.0200
uf20-0188.cnf	9	0.0600

D.2 # of variables: 50, # of clauses: 218

Problem Name	Number of Nodes	Completion Time(sec.)
uf50-01.cnf	356	1.0400
uf50-010.cnf	7	0.2100
uf50-0100.cnf	214	0.8800
uf50-01000.cnf	8	0.1900
uf50-0101.cnf	74	0.4200
uf50-0102.cnf	-	0.0700
uf50-0103.cnf	100	0.3700
uf50-0104.cnf	130	0.4900
uf50-0105.cnf	-	0.0300
uf50-0106.cnf	330	1.3300
uf50-0107.cnf	231	0.7900
uf50-0108.cnf	91	0.4900
uf50-0109.cnf	-	0.0400

Problem Name	Number of Nodes	Completion Time(sec.)
uf50-011.cnf	358	1.4000
uf50-0110.cnf	96	0.5800
uf50-0111.cnf	22	0.3100
uf50-0112.cnf	240	0.8500
uf50-0113.cnf	30	0.2900
uf50-0114.cnf	120	0.6400
uf50-0115.cnf	1	0.1200
uf50-0116.cnf	156	0.7200
uf50-0117.cnf	7	0.2900
uf50-0118.cnf	251	0.8200
uf50-0119.cnf	65	0.4600
uf50-012.cnf	116	0.6000
uf50-0120.cnf	161	0.5900
uf50-0121.cnf	154	0.6700
uf50-0122.cnf	164	0.5900
uf50-0123.cnf	76	0.4500
uf50-0124.cnf	54	0.4000
uf50-0125.cnf	352	1.3900
uf50-0126.cnf	17	0.2400
uf50-0127.cnf	552	1.6400
uf50-0128.cnf	232	1.0400
uf50-0129.cnf	157	0.7500
uf50-013.cnf	95	0.4000
uf50-0130.cnf	4	0.2400
uf50-0131.cnf	34	0.4100
uf50-0132.cnf	160	0.6600
uf50-0133.cnf	188	0.8700
uf50-0134.cnf	68	0.4200
uf50-0135.cnf	131	0.7300
uf50-0136.cnf	-	0.0400
uf50-0137.cnf	-	0.0500
uf50-0138.cnf	-	0.0400
uf50-0139.cnf	663	2.1000
uf50-014.cnf	72	0.4600

Problem Name	Number of Nodes	Completion Time(sec.)
uf50-0140.cnf	2	0.1300
uf50-0141.cnf	-	0.0300
uf50-0142.cnf	72	0.3800
uf50-0143.cnf	100	0.6100
uf50-0144.cnf	216	0.9800
uf50-0145.cnf	107	0.6100
uf50-0146.cnf	81	0.5900
uf50-0147.cnf	692	2.4600
uf50-0148.cnf	33	0.3500
uf50-0149.cnf	120	0.5400
uf50-015.cnf	32	0.3500
uf50-0150.cnf	225	0.9500
uf50-0151.cnf	197	0.7800
uf50-0152.cnf	4	0.1900
uf50-0153.cnf	128	0.7900
uf50-0154.cnf	21	0.2600
uf50-0155.cnf	119	0.6800
uf50-0156.cnf	3	0.1500
uf50-0157.cnf	3	0.2100
uf50-0158.cnf	100	0.5900
uf50-0159.cnf	224	0.8200
uf50-016.cnf	49	0.3500
uf50-0160.cnf	84	0.4200
uf50-0161.cnf	-	0.0200
uf50-0162.cnf	89	0.5500
uf50-0163.cnf	544	1.8200
uf50-0164.cnf	-	0.0200
uf50-0165.cnf	-	0.0300
uf50-0166.cnf	17	0.2700
uf50-0167.cnf	64	0.5500
uf50-0168.cnf	45	0.3700
uf50-0169.cnf	2	0.1300
uf50-017.cnf	161	0.6900
uf50-0170.cnf	87	0.5700

Problem Name	Number of Nodes	Completion Time(sec.)
uf50-0171.cnf	299	1.0800
uf50-0172.cnf	97	0.6700
uf50-0173.cnf	252	1.0300
uf50-0174.cnf	53	0.4500
uf50-0175.cnf	115	0.5300
uf50-0176.cnf	374	1.2000
uf50-0177.cnf	11	0.2500
uf50-0178.cnf	13	0.1900
uf50-0179.cnf	212	0.9200
uf50-018.cnf	27	0.2800
uf50-0180.cnf	213	0.8100
uf50-0181.cnf	507	1.7300
uf50-0182.cnf	455	1.3300
uf50-0183.cnf	5	0.1800
uf50-0184.cnf	8	0.2500
uf50-0185.cnf	-	0.0500
uf50-0186.cnf	82	0.5800
uf50-0187.cnf	130	0.5700
uf50-0188.cnf	55	0.4700

D.3 # of variables: 75, # of clauses: 325

Problem Name	Number of Nodes	Completion Time(sec.)
uf75-01.cnf	4	0.5200
uf75-010.cnf	5	0.3000
uf75-0100.cnf	870	5.0800
uf75-011.cnf	1642	6.9600
uf75-012.cnf	1539	8.6800
uf75-013.cnf	528	3.2500
uf75-014.cnf	451	3.2000
uf75-015.cnf	312	2.4100
uf75-016.cnf	1594	10.8000

Problem Name	Number of Nodes	Completion Time(sec.)
uf75-017.cnf	234	2.0000
uf75-018.cnf	94	1.3300
uf75-019.cnf	1829	11.9800
uf75-02.cnf	91	0.7400
uf75-020.cnf	408	2.9700
uf75-021.cnf	1057	5.7800
uf75-022.cnf	1243	7.4300
uf75-023.cnf	194	2.1700
uf75-024.cnf	565	3.6300
uf75-025.cnf	176	1.5700
uf75-026.cnf	3744	15.2300
uf75-027.cnf	6	0.5100
uf75-028.cnf	1292	5.2900
uf75-029.cnf	390	3.3600
uf75-03.cnf	1299	9.4300
uf75-030.cnf	42	0.8500
uf75-031.cnf	446	3.5400
uf75-032.cnf	173	2.0200
uf75-033.cnf	480	3.8000
uf75-034.cnf	1108	6.4000
uf75-035.cnf	1055	6.7400
uf75-036.cnf	3154	37.3600
uf75-037.cnf	862	4.8000
uf75-038.cnf	752	4.3400
uf75-039.cnf	12472	10.6100
uf75-04.cnf	1836	10.0500
uf75-040.cnf	228	2.2500
uf75-041.cnf	313	2.9700
uf75-042.cnf	1959	10.1100
uf75-043.cnf	162	1.8300
uf75-044.cnf	2461	16.0000
uf75-045.cnf	516	3.0300
uf75-046.cnf	1938	9.6100
uf75-047.cnf	494	3.0800

Problem Name	Number of Nodes	Completion Time(sec.)
uf75-048.cnf	30	0.9500
uf75-049.cnf	779	4.9700
uf75-05.cnf	671	5.5800
uf75-050.cnf	-	0.0700
uf75-051.cnf	624	3.9500
uf75-052.cnf	536	4.2100
uf75-053.cnf	155	1.9100
uf75-054.cnf	150	1.2400
uf75-055.cnf	4082	19.9700
uf75-056.cnf	292	2.7500
uf75-057.cnf	1172	7.3700
uf75-058.cnf	79	1.2000
uf75-059.cnf	2269	12.2000
uf75-06.cnf	4504	31.0300
uf75-060.cnf	165	1.5800
uf75-061.cnf	232	2.2800
uf75-062.cnf	1262	9.3800
uf75-063.cnf	251	2.3700
uf75-064.cnf	303	1.8100
uf75-065.cnf	182	1.3500
uf75-066.cnf	166	1.8300
uf75-067.cnf	1	0.4100
uf75-068.cnf	196	1.6900
uf75-069.cnf	749	4.7000
uf75-07.cnf	2305	13.4600
uf75-070.cnf	607	4.0100
uf75-071.cnf	1055	6.2200
uf75-072.cnf	238	2.3300
uf75-073.cnf	12	0.6500
uf75-074.cnf	144	1.6800
uf75-075.cnf	1013	5.7600
uf75-076.cnf	400	3.0100
uf75-077.cnf	306	2.2200
uf75-078.cnf	859	4.8300

Problem Name	Number of Nodes	Completion Time(sec.)
uf75-079.cnf	2112	14.4800
uf75-08.cnf	10	0.6200
uf75-080.cnf	167	1.88000
uf75-081.cnf	2400	26.0900
uf75-082.cnf	775	5.0100
uf75-083.cnf	669	4.5300
uf75-084.cnf	896	5.5800
uf75-085.cnf	-	0.0800
uf75-086.cnf	907	7.7800
uf75-087.cnf	154	1.1300
uf75-088.cnf	537	3.3700
uf75-089.cnf	339	2.2900
uf75-09.cnf	360	2.7400
uf75-090.cnf	1476	13.5100
uf75-091.cnf	592	3.6500
uf75-092.cnf	167	2.0700
uf75-093.cnf	176	1.7700
uf75-094.cnf	189	1.7100
uf75-095.cnf	1476	8.9100
uf75-096.cnf	1150	6.1400
uf75-097.cnf	3025	16.6300
uf75-098.cnf	-	0.0800
uf75-099.cnf	220	2.4500

D.4 # of variables: 100, # of clauses: 430

Problem Name	Number of Nodes	Completion Time(sec.)
uf100-01.cnf	4920	53.9200
uf100-010.cnf	5728	48.3900
uf100-0100.cnf	2437	19.3300
uf100-01000.cnf	979	10.1600
uf100-0101.cnf	6489	68.7500

Problem Name	Number of Nodes	Completion Time(sec.)
uf100-0102.cnf	1199	15.7400
uf100-0103.cnf	7435	68.4200
uf100-0104.cnf	386	6.8600
uf100-0105.cnf	3568	33.3000
uf100-0106.cnf	20261	158.3400
uf100-0107.cnf	2424	27.0000
uf100-0108.cnf	297	4.0500
uf100-0109.cnf	10874	111.3400
uf100-011.cnf	2590	25.2700
uf100-0110.cnf	7139	63.0900
uf100-0111.cnf	3571	36.7100
uf100-0112.cnf	609	7.7800
uf100-0113.cnf	16003	133.9400
uf100-0114.cnf	9756	103.6300
uf100-0115.cnf	8267	85.9100
uf100-0116.cnf	2613	28.1100
uf100-0117.cnf	332	5.2900
uf100-0118.cnf	16693	133.2900
uf100-0119.cnf	15	1.1800
uf100-012.cnf	407	4.3700
uf100-0120.cnf	955	10.7800
uf100-0121.cnf	6215	59.0400
uf100-0122.cnf	1287	14.9400
uf100-0123.cnf	11495	114.9100
uf100-0124.cnf	414	5.3400
uf100-0125.cnf	5851	62.8900
uf100-0126.cnf	14508	141.9500
uf100-0127.cnf	8209	76.6900
uf100-0128.cnf	4885	55.3600
uf100-0129.cnf	3543	38.3500
uf100-013.cnf	6072	52.8900
uf100-0130.cnf	4773	46.4800
uf100-0131.cnf	1147	13.6400
uf100-0132.cnf	2524	27.9900

Problem Name	Number of Nodes	Completion Time(sec.)
uf100-0133.cnf	1811	17.5300
uf100-0134.cnf	1973	21.0600
uf100-0135.cnf	1697	16.1200
uf100-0136.cnf	73	2.0400
uf100-0137.cnf	777	8.6900
uf100-0138.cnf	3337	28.2900
uf100-0139.cnf	2788	27.9800
uf100-014.cnf	1584	16.3700
uf100-0140.cnf	5884	53.2300
uf100-0141.cnf	3858	44.2400
uf100-0142.cnf	6652	62.0300
uf100-0143.cnf	1281	13.3000
uf100-0144.cnf	192	3.2700
uf100-0145.cnf	3705	42.9100
uf100-0146.cnf	669	8.3300
uf100-0147.cnf	492	5.6400
uf100-0148.cnf	2636	27.9500
uf100-0149.cnf	2884	25.4400
uf100-015.cnf	6806	70.9800
uf100-0150.cnf	4094	36.1200
uf100-0151.cnf	2616	29.0000
uf100-0152.cnf	9855	99.1700
uf100-0153.cnf	9443	69.1100
uf100-0154.cnf	10813	109.5400
uf100-0155.cnf	1276	16.8500
uf100-0156.cnf	609	7.4100
uf100-0157.cnf	78	2.3300
uf100-0158.cnf	3796	36.8400
uf100-0159.cnf	4943	53.2400
uf100-016.cnf	6525	65.1300
uf100-0160.cnf	3800	36.2500
uf100-0161.cnf	407	5.5200
uf100-0162.cnf	6205	62.2000
uf100-0163.cnf	4741	44.2400

Problem Name	Number of Nodes	Completion Time(sec.)
uf100-0164.cnf	214	3.5100
uf100-0165.cnf	706	8.8100
uf100-0166.cnf	2731	22.6900
uf100-0167.cnf	382	5.7500
uf100-0168.cnf	5205	53.5900
uf100-0169.cnf	1992	22.1700
uf100-017.cnf	27900	222.6400
uf100-0170.cnf	3443	40.3100
uf100-0171.cnf	680	8.9000
uf100-0172.cnf	583	7.5800
uf100-0173.cnf	5257	53.7800
uf100-0174.cnf	256	3.7300
uf100-0175.cnf	3081	35.5200
uf100-0176.cnf	440	5.1300
uf100-0177.cnf	2593	21.2500
uf100-0178.cnf	9384	90.2600
uf100-0179.cnf	857	9.8200
uf100-018.cnf	550	7.2900
uf100-0180.cnf	1641	16.6500
uf100-0181.cnf	7155	77.4800
uf100-0182.cnf	2243	21.5900
uf100-0183.cnf	4999	58.4700
uf100-0184.cnf	3717	39.6200
uf100-0185.cnf	1088	9.6800
uf100-0186.cnf	102	1.9400
uf100-0187.cnf	1081	12.6200
uf100-0188.cnf	531	6.2400

Bibliography

[AL97] E. Aarts and J. K. Lenstra, editors. *Local Search in Combinatorial Optimization*. Wiley-Interscience Series in Discrete Mathematics and Optimization. Wiley, 1997.

[APT79] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.

[AT79] 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*, 8:121–123, March 1979.

[AU74] J. H. A. V. Aho and J. Ullman. *The Design and Analysis of Computer Algorithm*. Addison-Wesley, 1974.

[BB93] Michael Buro and Hans Kleine Bűning. Report on a SAT competition. *EATCS Bulletin*, 49:143–151, 1993.

[CC95] Michele Conforti and Gérard Cornuéjols. A class of logic problems solvable by linear programming. *Journal of the ACM*, 42(5):1107–1112, 1995.

[CCKV00] Michele Conforti, Gérard Cornuéjols, Ajai Kapoor, and Kristina Vušković. Balanced $0, \pm 1$ matrices part i: Decomposition. Technical report, Septemper 2000.

[CCKV01] Michele Conforti, Gérard Cornuéjols, Ajai Kapoor, and Kristina Vušković. Perfect, ideal and balanced matrices. *European Journal of Operations Research*, 133:455–461, 2001.

[CH91] V. Chandru and J. N. Hooker. Extended horm sets in propositional logic. *Journal of the ACM*, 38:205–221, January 1991.

[CKT91] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the Really Hard Problems Are. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, IJCAI-91, Sidney, Australia*, pages 331–337, 1991.

[Com92] September 1992. DIMACS Challenge Committee. The Second DIMACS International Algotithm Implementation Challenge: General Information. Available via anunumous FTP from `dimacs.rutgers.edu`.

[Com93] May 1993. DIMACS Challenge Committee. Satisfiability Suggested Format. Available via anunymous FTP from `dimacs.rutgers.edu`.

[Coo71] S. Cook. The complexity of theorem proving processing. In *Proceedings of the Third Annual ACM Symposium, Theory of Computing*, pages 151–158, 1971.

[CS88] Vašek Chvátal and Endre Szemerédi. Many hard examples for resolution. *Journal of the ACM*, 35(4):759–768, 1988.

[DABC96] O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier. SAT versus UNSAT. *Discrete Mathematics and Theoretical Computer Science*, 26:415–436, 1996.

[DBL92] Mitchell D., Selman B., and H. Levesque. Hard and easy distributions of SAT problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 459–465, 1992.

[Dec92] Rina Dechter. Constraint networks. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*. John Wiley and Sons, New York, 2nd edition, 1992.

[DG84] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formula. *Journal of Logic Programming*, 3:267–284, 1984.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DP60] M. Davis and H. Putman. A computing procedure for quantification theory. *Journal of ACM*, 7:201–215, 1960.

[DR94] Rina Dechter and Irina Rish. Directional resolution: The Davis-Putnam procedure, revisited. In Jon Doyle, Erik Sandewall, and Pietro Torasso, editors, *KR'94: Principles of Knowledge Representation and Reasoning*, pages 134–145. Morgan Kaufmann, San Francisco, California, 1994.

[FP83] J. Franco and M. Paull. Probabilistic analysis of the Davis-Putnam procedure for solving the satisfiability problem. *Discrete Applied Mathematics*, 5:77–87, 1983.

[Fre95] Jon William Freeman. *Improvement to Propositional Satisfiability Search Algorithms.* PhD thesis, University of Pennsylvania, 1995.

[GN72] Robert S. Garfinkel and George L. Nemhauser. *Integer Programming.* Wiley-Interscience, 1972.

[Gol79] Allen Goldberg. On the complexity of the satisfiability problem. Technical Report 16, Courant Institute of Mathematical Science, New York University, New York, 1979.

[Gu92] J. Gu. Efficient local search for very large-scale satisfiability problems. *SIGART Bulletin 3*, 1(1992):8–12, 1992.

[GW93] I. Gent and T. Walsh. Towards an understanding of hill-climbing procedures for SAT. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pages 28–33, 1993.

[HK56] A. J. Hoffman and J. B. Kruskal. Integral boundary points of convex polyhedar. In H. W. Kuhn and A. W. Tucker, editors, *Linear Inequalities and Related Systems*, pages 223–246. Princeton University Press, Princeton, 1956.

[JSD93] Brigitte Jaumard, Mihnea Stan, and Jacques Desrosiers. Tabu search and a quadratic relaxation for the satisfiability problem, October 1993. Presented at the DIMACS Challenge II Workshop.

[KP92] Elias Koutsoupias and Christos Papadimitriou. On the greedy algorithm for satisfiability. *Information Processing Letters*, 43:53–55, 1992.

[Li99] Chu-Min Li. A constraint-based approach to narrow search trees for satisfiability. *Information Processing Letters*, 71:75–80, 1999.

[Mac92] Alan K. Mackworth. Constraint satisfaction. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*. John Wiley and Sons, New York, 2nd edition, 1992.

[MR91] Mitterreiter and F. J. Radermacher. Experiments on the running time behavior of some algorithms solving propositional logic problems. Technical report, Forschungsinstitut für anwendungsorientierte Wissensverarbeitung, Ulm, 1991.

[MSK97] D. McAllester, B. Selman, and H. Kautz. Evidence for invariants in local search. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, 1997.

[PR88] R. Gary Parker and Ronald L. Rardin. *Discrete Optimization.* Academic Press, 1988.

[RD77] J. N. E. M. Reingold and N. Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall, 1977.

[Ree93] C. R. Reeves, editor. *Modern Heuristic Techniques for Combinatirial Priblems.* Halsted Press, 1993.

[Roj] M. C. R. Rojas. *From quasi-solutions to solution: An evolutionary algorithm to solve csp.*

[RSOR96] V. Rayward-Smith, I. Osman, and C. Reeves, editors. *Modern Heuristic Search Methods.* Wiley, 1996.

[Scu90] Maria Brazia Scutellà. A note on dowling and gallier's top-down algorithm for propositional horn satisfiability. *Journal of Logic Programming*, 8(3):265–273, May 1990.

[Sey80] P. D. Seymour. Decomposition of regular matroids. *Journal of Combinatorial Theory (B)*, 28:305–359, 1980.

[SKC94] B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 337–343, 1994.

[SLM92] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 440–446, 1992.

[Tru90] K. Truemper. Polynomial theorem proving I. central matrices. *Tech. Rep. UTDCS*, pages 34–90, 1990.

[VD68] A. F. Jr. Veinott and G. B. Dantzig. Integral extreme points. *SIAM Review*, 10:371–372, 1968.

[Wal99] J. P. Walser. *Integer Optimization by Local Search: A Domain-Independent Approach*, volume 1637 of *Lecture Notes in Artificial Intelligence.* Springer, 1999.

[Wol98] L. A. Wolsey. *Integer Programming.* Wiley, New York, 1998.