ABSTRACT

Li, Xiao Yu. *Optimization Algorithms for the Minimum-Cost Satisfiability Problem.* (Under the direction of Dr. Matthias F. Stallmann and Dr. Franc Brglez)

Given a Boolean satisfiability (*Sat*) problem whose variables have non-negative weights, the minimum-cost satisfiability (*MinCostSat*) problem finds a satisfying truth assignment that minimizes a weighted sum of the truth values of the variables. Many NP-optimization problems are either special cases of *MinCostSat* or can be transformed into *MinCostSat* efficiently. However, in the past, these problems have been largely considered in isolation. In this dissertation, we (1) classify existing *MinCostSat* problems, (2) study factors affecting the performance of *MinCostSat* solvers, (3) propose algorithms for *MinCostSat* problems, and (4) implement and validate the performance of state-of-the-art solvers for special cases of *MinCostSat*, including set and binate covering, *Max-Sat*, and *group-partial Max-Sat.*

We categorize *MinCostSat* problems as either native or non-native. Non-native problems can only be transformed into *MinCostSat* by adding slack variables. These problems include the *Max-Sat*, *partial Max-Sat*, and *group-partial Max-Sat* problems which have applications ranging from course assignment to FPGA detailed routing. Native problems are various sub-cases of *MinCostSat.* We further divide these into two groups: covering problems and non-covering problems. The covering problems include the unate or set covering and the binate covering problems. They have applications in Operations Research (e.g., routing and scheduling) and logic synthesis (e.g., logic minimization and finite state machine minimization). In the covering problems, all or most clauses contain no complemented variables and a feasible solution is relatively easy to find. The non-covering problems contain clauses that are highly constrained, and sometimes only a small fraction of the variables are weighted. The non-covering problems have applications in minimum-size test pattern and minimum-length plans.

We study two important performance factors, among others, in branch-and-bound *MinCostSat* solvers. They are the lower-bounding and upper-bounding strategies. For lower bounding, we incorporate two advanced techniques: linear programming relaxation and cutting planes. Both methods can provide much higher quality lower-bounds than previous methods based on maximum independent sets of rows. For upper bounding, we show that our local-search *MinCostSat* solver, when initialized and terminated properly, can find the best upper-bound quickly.

Other techniques that contribute to the engineering of state-of-the-art solvers for applications of *MinCostSat* are also introduced. This work has led to the development of (1) a solver for covering problems that consistently outperforms previous leading solvers by as much as two orders of magnitude, (2) a logic minimizer that is able to solve three benchmark problems whose solution has eluded solvers for more than a decade, (3) a *Max-Sat* solver that challenges the dominance of linear programming solvers, particularly *cplex*, on *Max-2-Sat* benchmarks, and (4) a stochastic local-search solver for *group-partial Max-Sat* (with applications to FPGA routing) that finds known optima quickly and is able to find better than previously-known solutions on benchmarks whose optima remain unknown.

Optimization Algorithms for the Minimum-Cost Satisfiability Problem

by

Xiao Yu Li

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

August 2004

Approved By:

To my family . . .

Biography

Personal

Born in Wuhan, Hubei, China, on the 13th of August, 1977

Education

- North Carolina State University, Ph.D. in Computer Science, August 2004
- North Carolina State University, M.S. in Computer Science, December 2002
- University of Louisiana at Monroe, B.S. in Computer Science, May 2000

Publications

1. X.Y. Li and M.F. Stallmann. "New bounds on the barycenter heuristic for bipartite graph drawing." *Information Processing Letters*, 82 (2002) pp. 293-298.
2. F. Brglez, X.Y. Li, and M.F. Stallmann. "On SAT Instance Classes and a Method for Reliable Performance Experiments with SAT Solvers." *Annals of Mathematics and Artificial Intelligence*, 2004. In print.
3. F. Brglez, X.Y. Li, M.F. Stallmann and M. Burkhard. "Evolutionary and alternative algorithms: reliable cost predictions for finding optimal solutions to the LABS problem." *Information Science Journal*, 2004. In print.
4. X.Y. Li, M.F. Stallmann and F. Brglez. "A local search SAT solver using an effective switching strategy and an efficient unit propagation." *Springer-Verlag Lecture Notes in Computer Science*, vol 2919 (2004), pp. 53-68.
5. F. Brglez, X.Y. Li, M.F. Stallmann and B. Militzer. "Reliable cost predictions for finding optimal solutions to LABS problem: evolutionary and alternative algorithms." In Proceedings of *FEA*, Cary, NC, September 2003.

6. X.Y. Li, M.F. Stallmann and F. Brglez. “QingTing: A fast SAT solver using local search and efficient unit propagation.” In Proceedings of *SAT2003*, S. Margherita Ligure - Portofino, Italy, May 2003.

7. F. Brglez, M.F. Stallmann and X.Y. Li. “SATbed: an environment for reliable performance experiments with SAT instance classes and algorithms.” In Proceedings of *SAT2003*, S. Margherita Ligure - Portofino, Italy, May 2003

8. F. Brglez, X.Y. Li and M.F. Stallmann. “The role of a skeptic agent in testing and benchmarking of SAT algorithms.” In Proceedings of *SAT2002*, Cincinnati, OH, May 2002.

Awards

- Microsoft Fellowship (2003 - 2004)
- Academic Scholarship with Out-of-State Tuition Waiver (1996 - 2000)
- President’s and Dean’s List (1996 - 2000)
- Math. Assoc. of America (MS/LA) Competition 5th Place Winner (1999)
- Math. Assoc. of America (MS/LA) Competition 9th Place Winner (1998)
- Univ. of Louisiana at Monroe Chemistry Competition 1st Place Winner (1997)

Acknowledgments

I take this opportunity to thank Dr. Matt Stallmann and Dr. Franc Brglez without whose direction this thesis would not have been possible. I owe a lot of my knowledge, writing, and organization skills to them. They have spent countless hours in shaping my research career. Their constant drive for excellence has finally instilled in me a desire for excelling in whatever I do.

I gratefully acknowledge the time and commitment of the other two members of my committee, Dr. Dennis Bahler and Dr. Purush Iyer. I want to thank the staff of the department, especially Margery Page, for doing the administrative work. The four-inch thick folder in the departmental office with my name on it reminds me how much you have helped me over the four years. I also want to thank the Department of Computer Science for supporting me throughout my graduate studies.

Most of my research would not have taken place without the software packages and benchmarks shared with us by our dear colleagues. These generous individuals include (in alphabetical order) Fadi Aloul, Brian Borchers, Robert Brayton, Luca Carloni, Olivier Coudert, Edward Hirsch, Henry Kautz, Paolo Liberatore, Sharad Malik, Vasco Manquinho, João Marques-Silva, Jordi Planes, Richard Rudell, Bart Selman, Tiziano Villa, Hui Xu, Hantao Zhang, and Lintao Zhang. I also want to thank Dr. Rudra Dutta for lending his machine and software to us at the final and most critical stages of writing this thesis. Thanks also goes to Network Appliances for the filer you provided to our research group. The technical support team, Jason Corley and Sarah Williams, I appreciate your timely help on many occasions.

I cherish the time I have spent with the many friends while at NC State. Suzanne, Ergune, Laura, Chris, Feng Fang, Cai Xin, Zhou Tong, Ran Xia, Liang Xun, Xie Bin, the tennis partners, and the basketball teammates, you will always be in my thoughts.

Last, but the most. If it were not for my parents and sister, I would not have started my adventure nine years ago and finally made it today. I dedicate this thesis to them. And Ling, how can I forget you, thanks for pushing me so hard in the past two years. We made it together.

Table of Contents

List of Figures **ix**

List of Tables **x**

1 Introduction **1**
- 1.1 Definition of MinCostSat . . . 2
- 1.2 Motivating Examples . . . 3
  - 1.2.1 MinCostSat and 0-1 Integer Programming . . . 6
- 1.3 MinCostSat Problem Classification . . . 7
- 1.4 Approaches and Challenges . . . 9
- 1.5 Thesis Organization . . . 11

2 Thesis Overview **13**
- 2.1 Special Cases of MinCostSat . . . 13
  - 2.1.1 The Covering Problems . . . 15
  - 2.1.2 Non-covering Problems . . . 17
  - 2.1.3 MaxSat Problems . . . 19
  - 2.1.4 Partial MaxSat Problems . . . 20
  - 2.1.5 Group-Partial MaxSat Problems . . . 22
- 2.2 Our Contributions . . . 23
  - 2.2.1 Branch-and-Bound UCP/BCP Algorithms . . . 23
  - 2.2.2 Local-search Algorithms for FPGA Detailed Routing . . . 25
  - 2.2.3 Branch-and-Bound Algorithms for MaxSat . . . 25
  - 2.2.4 Branch-and-Bound Algorithm for Minimum-Size Test Pattern Problem . . . 26
- 2.3 Summary . . . 26

3 Survey of Branch-and-Bound Algorithms **28**
- 3.1 Classical Branch-and-Bound Algorithm . . . 28
  - 3.1.1 Reduction Techniques . . . 29

3.1.2 Lower-Bounding Technique 39
3.1.3 Search Pruning Technique 44
3.1.4 Branching Variable Selection 45
3.1.5 The Classical Branch-and-Bound Algorithm 46
3.2 SAT-based Algorithms 47
3.2.1 SAT-based Linear Search Algorithm 47
3.2.2 SAT-based Branch-and-Bound Algorithm 48
3.3 Summary 48

4 Engineering an Efficient Branch-and-Bound MinCostSat Solver 49
4.1 Introduction to the Eclipse Algorithm 50
4.2 Key Features 50
4.3 Experimental Setup 52
4.4 Performance Factors 53
4.4.1 Factor 1: Lower-Bounding Techniques 53
4.4.2 Factor 2: Upper-bounding Techniques 59
4.4.3 Factor 3: Heuristic Search-tree Exploration 64
4.4.4 Factor 4: Branching Variable Selection 67
4.4.5 Factor 5: Search Pruning 68
4.4.6 Factor 6: Reduction Techniques: 68
4.4.7 Factor 7: Data Structures 69
4.5 Experimental Results 70
4.5.1 The Solvers 70
4.5.2 Unate Covering Comparisons 71
4.5.3 Binate Covering Comparisons 78
4.5.4 Runtime Analysis 81
4.6 Summary 81

5 Local Search Algorithms 86
5.1 Introduce the Eclipse-stoc Algorithm 86
5.2 Benchmark Profiling 87
5.2.1 The Static Measure 88
5.2.2 The Dynamic Measure 88
5.3 Addressing the Feasibility Issue 92
5.3.1 The unitwalk Algorithm Adapted for BCP 92
5.3.2 The walksat Algorithm Adapted for BCP 95
5.4 Addressing the Quality Issue 95
5.5 The Eclipse-stoc Algorithm 97
5.6 Experimental Results 97
5.6.1 Unate Covering Comparisons Under Termination A 98
5.6.2 Binate Covering Comparisons Under Termination A 100
5.6.3 Experiments Under Termination Criterion B 100

5.7 Summary 102

6 Applications and Case Studies 103
6.1 UCP and Two-Level Logic Minimization 103
6.1.1 Background 104
6.1.2 Minimizing the Two-Level Logic 105
6.1.3 Two-Level Logic Minimizer – Espresso 107
6.1.4 Experimental Results 107
6.1.5 Summary 110
6.2 Group-Partial MaxSat and FPGA Routing 110
6.2.1 The RCS Formulation 111
6.2.2 The sub_SAT Approach 113
6.2.3 The Local-Search Approach 114
6.2.4 Experimental Results 115
6.2.5 Summary 117
6.3 MaxSat 118
6.3.1 The Branch-and-Bound Algorithm – qtmax 118
6.3.2 Experimental Results with qtmax 122
6.3.3 Solving MaxSat as IP and MinCostSat 128
6.3.4 Summary 131
6.4 Minimum-size Test Pattern Problem 131
6.4.1 The BCP Formulation 132
6.4.2 Experimental Results 132
6.4.3 Summary 133
6.5 SAT Comparisons 135

7 Conclusions and Future Work 137

Bibliography 140

List of Figures

1.1 Unate covering example: the players and their positions represented in a matrix format. 3
1.2 Classification of *MinCostSat*. 8

2.1 The *MinCostSat* problems studied in this thesis. 14

3.1 The row dominance reduction algorithm. 32
3.2 The column dominance reduction algorithm. 35
3.3 The overall reduction algorithm. 38
3.4 The greedy algorithm for constructing the MIS and calculating the lower-bound. 40
3.5 The improved algorithm for constructing MIS and calculating the lower-bound. 43
3.6 The branch-and-bound algorithm for *MinCostSat* in [1]. 46

4.1 The seven performance factors in *eclipse*. 54
4.2 The breakdown of execution time for various components of *eclipse-lpr* (top) and *eclipse-cp* (bottom). 82

5.1 The algorithm for calculating the variable immunity 89
5.2 The *unitwalk* algorithm adapted for *MinCostSat*. 93
5.3 The walksat algorithm adapted for *MinCostSat*. 94
5.4 The neighborhood-search algorithm that searches for optimal solutions. 96
5.5 The eclipse-stoc algorithm for *MinCostSat*. 97

6.1 A small unsatisfiable routing example (3 nets and 3 tracks) and its RCS formulation. 113
6.2 The *wpack* algorithm for optimizing FPGA detailed routing. 115
6.3 A recursive DPLL-based *Max-Sat* algorithm. 119
6.4 Asymptotic behavior of three solvers' runtime and nodes for the *Max-2-Sat* benchmarks. 130

List of Tables

2.1 The unate covering benchmarks from logic synthesis, set covering and randomly generated instances. 16
2.2 The binate covering benchmarks from logic synthesis. 18
2.3 The non-covering benchmarks from `ATPG` and *Sat*. 19
2.4 The non-native benchmarks from random *Max-2-Sat* and *Max-3-Sat*. 21
2.5 The non-native benchmarks from *group-partial Max-Sat* (FPGA detailed routing). 24

4.1 Comparison of four lower bounding techniques on unate covering benchmarks. 57
4.2 Comparison of four lower-bounding techniques on binate covering benchmarks. 58
4.3 Comparison of two local-search initialization strategies in *eclipse*: random initialization vs. heuristic initialization. 62
4.4 Comparison of four different upper-bounding methods with *eclipse*. . 65
4.5 Comparison of two search-tree exploration strategies: depth-first search vs. best-first search. 67
4.6 Comparison of *scherzo*, *aura*II, *cplex*, *eclipse-lpr* and *eclipse-cp* on unate covering benchmarks (reference instances only) from logic minimization. 72
4.7 Comparison of *cplex*, *eclipse-lpr* and *eclipse-cp* on P-classes of size 32 for the unate covering benchmarks from two-level logic minimization. 74
4.8 Comparison of *scherzo*, *aura*II, *cplex*, *eclipse-lpr* and *eclipse-cp* on unate covering benchmarks (reference instances only) from set covering and randomly generated set. 76
4.9 Comparison of *cplex*, *eclipse-lpr* and *eclipse-cp* on P-classes of size 32 for the unate covering benchmarks from set covering and randomly generated set. 77
4.10 Comparison of *scherzo*, *bsolo*, *cplex*, *eclipse-lpr* and *eclipse-cp* on binate covering benchmarks (reference instances only) from logic synthesis. . 79

4.11 Comparison of *cplex*, *eclipse-lpr*, and *eclipse-cp* on P-classes of size 32 for the binate covering benchmarks from logic synthesis. 80

5.1 The static and dynamic measures of structuredness on unate covering benchmarks. 90
5.2 The static and dynamic measures of structuredness on binate covering benchmarks. 91
5.3 The *eclipse-stoc* results on unate covering benchmarks under termination criterion A. 99
5.4 The *eclipse-stoc* results on binate covering benchmarks under termination criterion A. 101
5.5 Find new best solutions with *eclipse-stoc*. 102

6.1 Comparison of *espresso*-exact and *latte* on three previous unsolved benchmarks. 108
6.2 Comparison of solution quality between local-search and branch-and-bound methods on two-level logic minimization benchmarks 109
6.3 Runtime comparisons between *sub_SAT*, *wpack* and *eclipse-stoc* on FPGA detailed routing benchmarks. 116
6.4 Solution quality comparisons between *sub_SAT* and *wpack* on benchmarks whose optima are unknown. 117
6.5 Backtrack comparisons for three variants of *qtmax* on random *Max-2-Sat* and *Max-3-Sat* (reference instances only). 121
6.6 Average performance comparisons of *maxsat*, *LB2+MOMS*, *LB2+JW* and *qtmax* on PC-classes of size 32 from *Max-2-Sat* benchmarks. . . . 124
6.7 Average performance comparisons of *maxsat*, *LB2+MOMS*, *LB2+JW* and *qtmax* on PC-classes of size 32 from *Max-3-Sat* benchmarks. . . . 125
6.8 Backtrack and runtime comparisons for *maxsat*, *LB2+MOMS*, *LB2+JW* and *qtmax* on PC-classes of size 32 from two random *Max-Sat* instances. 126
6.9 Backtrack and runtime comparisons for *maxsat*, *LB2+MOMS*, *LB2+JW* and *qtmax* on PC-classes of size 32 from two structured *Max-Sat* instances. 127
6.10 Results for *qtmax*, *cplex* and *eclipse* on PC-classes of size 32 from random *Max-2-Sat* and *Max-3-Sat* benchmarks. 129
6.11 Comparisons between *bsolo* and *eclipse-bf* on P-classes of size 16 of the `ATPG` benchmarks. 134
6.12 Comparison of *cplex*, *eclipse-cp* and *zchaff* on P-Classes of size of 32 from the SAT benchmarks. 135

Chapter 1

Introduction

The propositional satisfiability problem (*Sat*) underlies many applications such as artificial intelligence (e.g., planning [2, 3, 4]), electronic design automation [5] (e.g., test pattern generation [6], equivalence checking [7, 8], routing [9, 10, 11, 12, 13]), and operations research (e.g., scheduling [14]). In recent years, *Sat* research has attracted much attention and its tremendous advances [15, 16, 17] continue to motivate researchers to formulate problems from various domains into *Sat*. The *Sat* formulation is attractive: (1) state-of-the-art *Sat* solvers are general purpose and extremely fast, (2) improvements are constantly being made to *Sat* solvers and problems formulated as *Sat* can immediately take advantage of these improvements.

Despite these clear advantages, the *Sat* formulation also has its innate limitations. *Sat* is a decision problem and this means that a *Sat* solver can either find a feasible solution or claim that no solution exists for a given problem instance. However, the *Sat* approach has no control over the quality of the solution it finds. As a result, *Sat* solvers are not capable of directly solving optimization problems, in which high quality solutions are desirable.

In this thesis, we study an optimization problem that is closely related to *Sat*: the minimum-cost Satisfiability (*MinCostSat*) problem. Given a *Sat* problem whose variables have non-negative weights, the *MinCostSat* problem finds a satisfying truth assignment that minimizes a weighted sum of the truth values of the variables. *MinCostSat* problems arise in practice in many occasions. For example, in *Sat*-based planning [2], in order to find a minimum-length plan that minimizes the number of operators, the weight of each variable for the operators is assigned 1 and the weight of all other variables is assigned 0. The goal is then to find a satisfying assignment to a *Sat* formula with the least number of operator variables set to 1.

1.1 Definition of MinCostSat

In order to formally define *MinCostSat*, we need to first define *Sat*[1]. An instance of *Sat* is a Boolean formula F that is made up of three components:

1. a set of propositional variables $X = \{x_1, x_2, \cdots, x_n\}$ where $x_i \in \{0, 1\}$ for integer $i \in [1, n]$.

2. a set of literals each of which is in the form of variable x or its complement $\bar{x}$ where $x \in X$.

3. a set of clauses $Y = \{y_1, y_2, \cdots, y_m\}$ where each clause consists of literals combined by logical *or* ($\vee$) connectives.

The *Sat* problem is: Does there exists an assignment to the variables such that the following formula in conjunctive normal form (CNF) is true:

$$y_1 \wedge y_2 \wedge y_3 \cdots \wedge y_m$$

where $\wedge$ is the logic *and* connective?

Definition 1 (MinCostSat) *Given a Boolean formula F in conjunctive normal form with n variables and m clauses, and a cost function that assigns non-negative c_i*

[1]We restrict our attention to formulas in conjunctive normal form.

to x_i for integer $i \in [1, n]$ and $x_i \in \{0, 1\}$, the MinCostSat problem is the problem of finding a satisfying assignment for F that minimizes the objective function:

$$\sum_{i=1}^{n} c_i x_i. \tag{1.1}$$

MinCostSat is NP-hard. For its approximability, please refer to [18].

1.2 Motivating Examples

The unate and binate covering problems [19, 1], not traditionally viewed as *MinCostSat* problems, are also sub-cases of *MinCostSat*. To help introduce *MinCostSat* and motivate our work, we present an NBA trading scenario that can be formulated by the covering problems.

A Unate Covering Example. Assume the NBA team *Raleigh Wolves* is buying players to fill three back-up positions: guard, forward and center. Some potential candidates and the positions they play are shown in the following 0-1 matrix (for example, Duncan plays both the forward and center position, but not the guard position): The trade values for the players are: Duncan – \$5 million, Francis – \$4

$$\begin{array}{rccccc} & Duncan & Francis & Malone & Stockton & Yao \\ Guard & 0 & 1 & 0 & 1 & 0 \\ Forward & 1 & 0 & 1 & 0 & 0 \\ Center & 1 & 0 & 1 & 0 & 1 \end{array}$$

Figure 1.1: Unate covering example: the players and their positions represented in a matrix format.

million, Malone – \$6 million, Stockton – \$2 million, and Yao – \$3 million. The *Raleigh*

Wolves manager wants to sign up enough players to cover all the positions with the least cost.

This is an example of the unate covering problem (*UCP*), also known as the set covering problem . We create one Boolean variable, denoted by the first letter of the last name, for each player. For example, the variable Y is 1 if Yao is signed; Y is 0 otherwise. To cover each of the three positions, at least one of the players who play the position has to be chosen. Therefore, we create the following three Boolean clauses:

$$F + S \qquad \text{for the guard position} \tag{1.2}$$

$$D + M \qquad \text{for the forward position} \tag{1.3}$$

$$Y + D + M \qquad \text{for the center position} \tag{1.4}$$

To minimize the cost of acquiring the players, the manager needs to minimize the following linear function:

$$5 \cdot D + 4 \cdot F + 6 \cdot M + 2 \cdot S + 3 \cdot Y \tag{1.5}$$

where the coefficient for each variable is the cost of acquiring the player corresponding to that variable. Clearly, this is an instance of the *MinCostSat* problem. Duncan and Malone are the only players that play two positions. Since Duncan costs less than Malone, we can pick Duncan to cover the forward and the center positions. To cover the guard position, we choose Stockton, who is less expensive than Francis, the only other player for the guard position. The solution (Duncan and Stockton) constructed in this greedy fashion happens to be the optimal solution and it has a cost of \$7 million.

A Binate Covering Example. Suppose that, in addition to the constraints in the *UCP* example above, there are two other constraints: (1) Duncan and Stockton don't get along so they can't be signed together, (2) if one of Malone and Stockton is signed, the other one must be signed, too (after 18 years, they still want to be together for a final shot at the championship). The goal is still to cover all the positions with the least cost.

To ensure that Duncan and Stockton are not signed together, we create the following clause:

$$\overline{D} + \overline{S} \tag{1.6}$$

To ensure Malone and Stockton are signed together, we generate $M \leftrightarrow S$, which is equivalent to the following two clauses:

$$\overline{M} + S \tag{1.7}$$

$$M + \overline{S} \tag{1.8}$$

Clearly, the Duncan and Stockton combination is no longer a feasible solution because it violates clause (1.6) above. Using the same greedy approach, we choose Duncan again to cover the forward and center positions, then we would have to choose Francis to cover the guard position. The cost of this solution is \$9 million. However, if we choose Malone instead of Duncan to play the forward and center positions, then we can choose Stockton for the guard position. The cost of this solution is only \$8 million. Therefore, the greedy approach for *UCP* doesn't work for *MinCostSat* in this case.

All the constraints above can be expressed in the following binate covering matrix (the ith row corresponds to the ith constraint presented above):

$$\begin{array}{ccccc} D & F & M & S & Y \\ \hline 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ -1 & 0 & 0 & -1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 1 & -1 & 0 \end{array}$$

If a variable appears positively (negatively, respectively) in a clause, its corresponding entry in the covering matrix is 1 (-1, respectively). If a variable doesn't appear in a clause, its corresponding entry is 0. *UCP* is a special case of *BCP* : all the entries in a *UCP* covering matrix are non-negative. *BCP* is also a subcase of *MinCostSat.*

1.2.1 MinCostSat and 0-1 Integer Programming

MinCostSat is a special case of 0-1 integer programming problem (*0-1 IP*).

Definition 2 *The objective of 0-1 integer programming problem is to*

$$\begin{array}{ll} \textit{minimize} & \sum_{i=1}^{n} c_i \cdot x_i \\ \textit{subject to} & A \cdot x \geq b, \quad x \in \{0,1\}^n \end{array}$$

where c_i is the nonnegative cost of variable $x_i \in \{0,1\}$ and $i \in [1,n]$; A is a $m \times n$ matrix; and b is a vector of size m. $A \cdot x \geq b$ is the set of m linear constraints.

In *MinCostSat*, the entries in the matrix A are from $\{-1,0,1\}$; therefore, we have the following:

Definition 3 *The objective of MinCostSat problem is to*

$$\begin{array}{ll} \textit{minimize} & \sum_{i=1}^{n} c_i \cdot x_i \\ \textit{subject to} & A \cdot x \geq b, \quad x \in \{0,1\}^n \end{array}$$

where c_i is the nonnegative cost of variable $x_i \in \{0,1\}$ and $i \in [1,n]$; A is a $m \times n$ matrix with entries from $\{-1,0,1\}$; and b is a vector of size m where $b_i = 1 - |\{a_{ij} : a_{ij} = -1\}|$. $A \cdot x \geq b$ is the set of m linear constraints.

To show that the *Sat*-based definition and the *0-1 IP* based definition are equivalent, we observe that a CNF clause $(l_1 + l_2 + \cdots + l_k)$ can be viewed as a linear inequality $l_1 + l_2 + \cdots + l_k \geq 1$. If we use $1 - x_j$ to represent the complement literal $\overline{x}_j$ in a CNF formula, then a CNF clause can be expressed as a linear inequality. For example, consider the CNF clause in 1.6:

$$\overline{D} + \overline{S}$$

can be rewritten as:

$$(1 - D) + (1 - S) \geq 1$$

which turns into:

$$-D - S \geq -1.$$

The six CNF clauses in our *MinCostSat* example can then be rewritten as linear inequalities:

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ -1 & 0 & 0 & -1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 1 & -1 & 0 \end{bmatrix} \times \begin{bmatrix} D \\ F \\ M \\ S \\ Y \end{bmatrix} \geq \begin{bmatrix} 1 \\ 1 \\ 1 \\ -1 \\ 0 \\ 0 \end{bmatrix}$$

Notice that the inequalities comply with the *0-1 IP*-based definition. For example, the number of negative literals in the last row is 1 and $b_6 = 1 - 1 = 0$. The six inequalities can be satisfied iff there is a feasible solution for the six CNF clauses. Since *MinCostSat* is a special case of *0-1 IP*, integer programming tools such as *cplex* [20] can be used to solve *MinCostSat*.

1.3 MinCostSat Problem Classification

Many NP-optimization problems are either subcases of *MinCostSat* (e.g., *UCP* and *BCP*) or can be transformed into *MinCostSat* efficiently by introducing slack variables. We refer to the former as *native MinCostSat* problems and the later as *non-native MinCostSat* problems. We depict our classification of the *MinCostSat* problems in Figure 1.2.

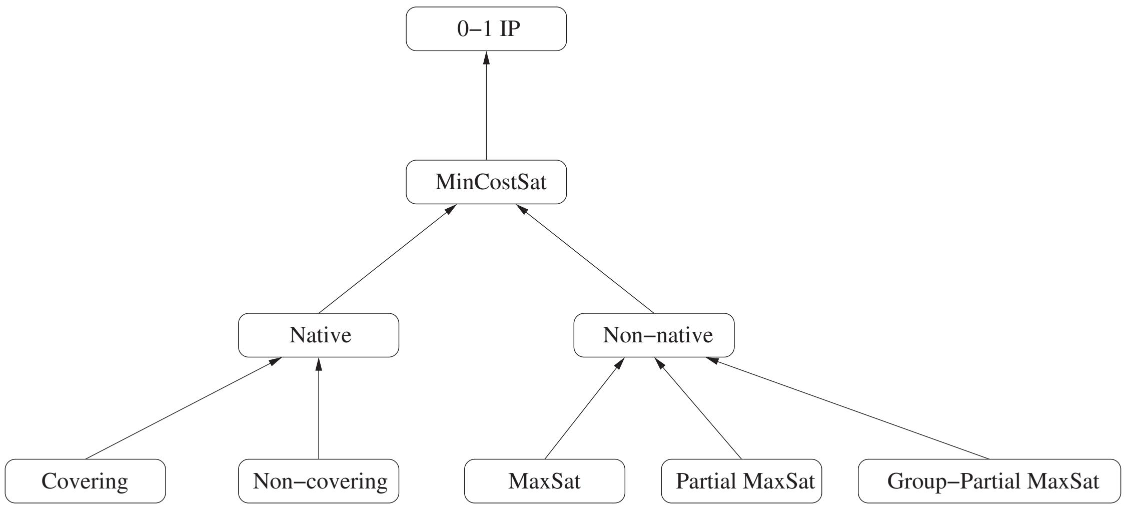


Figure 1.2: Classification of *MinCostSat*.

Native *MinCostSat* problems are further divided into the covering problems and the non-covering problem. The covering problems include unate covering and binate covering problems we presented earlier. The unate covering problems, also known as the set covering problems, have a number of applications in Operations Research (e.g., routing [21] and scheduling [22]) and logic synthesis (e.g., logic minimization [23]). The binate covering problems also have many applications in logic synthesis (e.g., finite state machine minimization [24], technology mapping [19], and Boolean relations [25]). Covering algorithms and solvers based on branch-and-bound [23, 26, 27, 28, 29, 30, 31] and local search [32, 33, 34, 35, 36] have been studied extensively in the past.

The non-covering *MinCostSat* problems have applications in minimum-size test pattern problem [37], minimum-size planning [38], and minimum-cost goal models [39]. Branch-and-bound algorithms and solvers for non-covering *MinCostSat* problems have been investigated in [40, 41, 42]. The differences between the covering problems and the non-covering problems are both syntactic and semantic. Syntactically, most clauses in the covering problems contain all positive literals whereas in the non-covering problems, most clauses contain a mixture of positive and negative literals. In addition, most if not all variables in the covering problems are weighted whereas in non-covering problems, only a fraction of variables may be weighted and the remaining variables have weight of zero. Semantically, the clauses in the covering problems are almost trivial to satisfy whereas in non-covering problems, the clauses can be highly constrained. Furthermore, reduction and lower-bounding techniques that tend to work well for the covering problems are not effective on the non-covering problems.

Non-native *MinCostSat* problems include *Max-Sat*, *partial Max-Sat*, and *group-partial Max-Sat*. The *partial Max-Sat* problem has applications in course assignment [43] and two-level crossing number minimization [44]. The *group-partial Max-Sat* has application in FPGA detailed routing [13]. As we will see in Chapter 2, non-native *MinCostSat* problem can only be transformed into *MinCostSat* by introducing slack variables. Branch-and-bound algorithms for *Max-Sat* are the topics in [45, 46, 47, 48, 49]. Various local-search *Max-Sat* algorithms are presented in [50, 51, 52]. We know of only two algorithms [43, 53] (both based on local search) for *partial Max-Sat* and only one algorithm [13] (based on branch-and-bound) for *group-partial Max-Sat*.

Traditionally, most of these problems have been considered in isolation. In this dissertation, we (1) classify *MinCostSat* problems, (2) study factors affecting the performance of *MinCostSat* solvers, (3) propose algorithms for *MinCostSat* problems, and (4) implement and validate the performance of state-of-the-art solvers for special cases of *MinCostSat*, including *UCP*, *BCP*, *Max-Sat*, and *group-partial Max-Sat*.

1.4 Approaches and Challenges

In general, there are two types of algorithm for solving *MinCostSat*:

1. **Branch-and-bound algorithms.** Given sufficient time, a branch-and-bound solver is guaranteed to find the optimal solution. It systematically explores the search space as follows: at each node of the branching tree, it simplifies the formula by applying several reduction techniques. The effectiveness of the

reductions depends largely on whether the problem represents a covering problem or not: reductions can reduce the size of the formula significantly but are usually useless for non-covering problems. When no more reductions can be applied, a branching variable is chosen and set to either 1 or 0 to generate the two sub search-trees of the current node. The search backtracks when either the upper-bound meets the lower-bound or a clause becomes unsatisfied. In both situations, some *MinCostSat* solvers [41, 42] utilize the *conflict diagnosis* and *non-chronological backtracking* techniques introduced in SAT solvers [15, 54, 16]. The lower-bounding techniques used in these algorithms are mostly based on *maximum independent set of rows* [28, 30, 40, 41] or *linear-programming relaxation* [29]. For upper bounding, some algorithms use local-search methods to find a good upper-bound before doing branch-and-bound. We survey branch-and-bound *MinCostSat* algorithms in Chapter 3.

2. **Local-search algorithms.** Local-search algorithms for *MinCostSat* have largely been ignored in the past. The local-search *partial Max-Sat* solver, *maxwalksat* [53], was used in [40] to solve *MinCostSat* problems after they were transformed to *partial Max-Sat* problems. Local-search algorithms don't guarantee optimality because the searches are greedy and may overlook some regions of the solution space. Experiments in [40] show that *maxwalksat* can sometimes fail to find a solution when one exists; in some other cases, the best solution it finds is not optimal.

The Challenge for Branch-and-bound Algorithms. There are at least two reasons why faster and better branch-and-bound *MinCostSat* solvers are needed. First, even though they may be slow in practice, branch-and-bound solvers are guaranteed to provide an optimal solution. When an optimal solution is valuable enough to justify a large investment in computation time, it is worth running a branch-and-bound solver. Secondly, in order to reliably evaluate the robustness and scaling behavior of local-search *MinCostSat* algorithms, experiments have to be done on benchmarks with a wide range of sizes. This requires the knowledge of the optimal solutions of these benchmarks. However, existing *MinCostSat* solvers time out on many standard benchmarks. In Chapter 4, we present an improved branch-and-bound *MinCostSat* solver that can solve some open problems. It not only incorporates many of the best features[2] in existing *BCP* solvers but also implements new lower bounding, upper bounding, and search-tree traversal algorithms.

The Challenge for Local-search Algorithms. The two main challenges faced by a local-search *MinCostSat* solver are (1) the feasibility issue: finding a solution that satisfies all the constraints if one exists and (2) the quality issue: finding the feasible solution with the optimal quality. In Chapter 5, we propose a new local-search algorithm, *eclipse-stoc*, for solving *MinCostSat*. In general, *eclipse-stoc* solves a *MinCostSat* problem by repeating the following two-stage process: in the construction stage, a feasible solution is constructed by a greedy method using known *Sat* techniques and then in the search stage, a local search is conducted around this solution to find solutions with better quality. For the constructive phase, we show that local-search techniques for *Sat* can be effectively extended to satisfy the constraints in *MinCostSat*.

1.5 Thesis Organization

In Chapter 2, we give an overview of the thesis by introducing the *MinCostSat* problems we study, their benchmarks and solvers, and our contributions. We survey the branch-and-bound algorithms for *MinCostSat* in Chapter 3. In Chapter 4, we introduce a new branch-and-bound *MinCostSat* solver that specializes in the covering problems. Its various components are described in detail and the important performance factors are studied through extensive experimentation. This is followed by a presentation of a new local-search *MinCostSat* solver in Chapter 5. In Chapter 6, we study the algorithms and solvers for five special cases of *MinCostSat*. The first one is *UCP* with application in *two-level logic minimization*. We show how the two-level logic minimizer *espresso* [23] can be improved when we replace its built-in covering procedure with our new covering solver. The second one is *group-partial Max-Sat*

[2]Conflict analysis and non-chronological backtracking are not implemented in our solver.

with application in FPGA detailed routing. We compare specific FPGA detailed routing solvers with our local-search *MinCostSat* solver. The third one is *Max-Sat*, a combinatorial optimization problem that can be formulated as either *MinCostSat* or *0-1 IP*. We introduce and compare a new branch-and-bound *Max-Sat* solver with the more general-purpose *MinCostSat* and integer programming solvers. The fourth one is a non-covering *MinCostSat* problem — the minimum-size test pattern problem. We present a simple and fast branch-and-bound solver that is competitive with the previous leading solver. The last one is *Sat* and we compare two *MinCostSat* solvers with a leading *Sat* solver on a set of common *Sat* benchmarks. Surprisingly, both *MinCostSat* solvers can outperform the *Sat* solver on some instances. Finally in Chapter 7, we conclude the thesis and present some future research directions.

Chapter 2

Thesis Overview

In this chapter, we give an overview of the thesis and problems it studies. We present all the cases of *MinCostSat* shown in the classification scheme depicted in Figure 1.2. For each case, we introduce its formulation, areas of application, the sets of benchmarks we study, and the existing approaches and solvers. At the end of the chapter, we give an overview of our contributions.

2.1 Special Cases of MinCostSat

In this thesis, we study all the special cases of *MinCostSat* shown in Figure 1.2. For the native problems, we study the covering problems that include *UCP* and *BCP*, and the non-covering problems. For the non-native problems, we study *Max-Sat*, *partial Max-Sat*, and *group-partial Max-Sat*. We present the following aspects of these problems, also summarized in Figure 2.1.

1. **Formulation**. Non-native *MinCostSat* problems can only be transformed into *MinCostSat* by introducing slack variables. We present their original formulation as well as the transformations.

Native Problems

a. Covering

Unate Covering

Applications:	OR [21, 22], logic minimization [23]
Benchmarks:	random [30], Steiner [55], industrial [23, 56]
Solvers:	cplex [20], espresso [23], scherzo [28], auraII [30] bsolo [41], **eclipse-lpr**, **eclipse-cp**

Binate Covering

Applications:	logic synthesis [19, 24, 25]
Benchmarks:	industrial [41, 56]
Solvers:	cplex, scherzo, bsolo, **eclipse-lpr**, **eclipse-cp**

b. Non-Covering

Applications:	minimum-size test pattern problem [41]
Benchmarks:	structured [41], subset of *Sat* [57, 58, 59]
Solvers:	cplex, bsolo, mindp [40], **eclipse_bf**

Non-Native Problems

a. MaxSat

Applications:	N/A
Benchmarks:	random [46]
Solvers:	cplex, maxsat [46], LB2+MOMS [47], LB2+JW [47] eclipse_lpr, **qtmax** [60]

b. Partial MaxSat

Applications:	course assignment [61]
Benchmarks:	N/A
Solvers:	cplex, maxwalksat [53], solver from [43]

c. Group-Partial MaxSat

Applications:	FPGA detailed routing [13]
Benchmarks:	structured [13]
Solvers:	cplex, sub_SAT [13], eclipse_lpr, **wpack**

Figure 2.1: The *MinCostSat* problems studied in this thesis.

2. **Applications**. All special cases of *MinCostSat* have practical applications except for *Max-Sat*. We present the application domains for each special case.

3. **Benchmarks**. We introduce the source of the benchmarks and their characteristics.

4. **Solvers**. We give a brief overview of the existing approaches and the solvers.

We now elaborate on cases shown in Figure 2.1 in more detail.

2.1.1 The Covering Problems

The covering problems can be divided into two groups: unate covering and binate covering. The literals in unate covering are restricted to be positive. Unate covering is a special case of binate covering.

Unate Covering Problems. *UCP* is also known as the set covering problem. The the literals in the CNF clauses $\{y_1, y_2 \cdots y_m\}$ for *UCP* are all positive. Given a CNF formula F and a cost assignment that assigns w_i to x_i for integer $i \in [1, n]$, the objective of *UCP* is to minimize:

$$W = \sum_{i=1}^{n} c_i x_i \tag{2.1}$$

$$\text{subject to} \qquad y_1 \wedge y_2 \wedge \cdots \wedge y_m = 1.$$

Solving *UCP* is a necessary step in solving two-level logic minimization [23] and it also has many applications in Operations Research [21, 22]. The three groups of *UCP* benchmarks presented in Table 2.1 include:

1. Two-level logic minimization benchmarks from [23, 56]. The benchmarks post significant challenges to previous *BCP*/*UCP* solvers and have historically served as a standard set of benchmarks for evaluating these solvers.

2. Set-covering benchmarks derived from Steiner triple systems [55].

3. Randomly generated benchmarks from [30].

Table 2.1: The unate covering benchmarks from logic synthesis, set covering and randomly generated instances.

						cplex	
benchmark	cols	rows	# of 1	# of −1	sparsity	opt	time
lin_rom	1076	1030	9955	0	0.0089	120	0.2
exam.pi	4677	509	25694	0	0.0107	63	3.6
bench1.pi	4677	398	9563	0	0.0510	121	4.4
prom2	2611	1924	15507	0	0.0031	278	5.2
prom2.pi	2618	1988	15545	0	0.0030	287	6.0
max1024	1264	1090	7221	0	0.0052	245	18.6
max1024.pi	1278	1087	6974	0	0.0050	259	21.1
ex5.pi	2460	873	40681	0	0.0190	65	25.6
ex5	2428	831	41085	0	0.0204	37	116.5
test4.pi	6139	1437	109318	0	0.0124	≤ 101	3600*
steiner_a0009	9	12	36	0	0.3333	5	0.01
steiner_a0015	15	35	105	0	0.2000	9	0.03
steiner_a0027	27	117	351	0	0.1111	18	0.8
steiner_a0045	45	375	990	0	0.0667	30	38.9
steiner_a0081	81	1080	3240	0	0.0370	≤ 61	3600*
m100_100_10_30	100	100	1968	0	0.1968	11	0.9
m100_100_10_15	100	100	1239	0	0.1239	10	1.8
m100_100_10_10	100	100	1000	0	0.1000	12	4.2
m200_100_10_30	100	200	3949	0	0.1975	11	82.1
m200_100_30_50	100	200	7941	0	0.3971	6	129.1

* cplex times out at 3600 seconds.

This table presents three sets of unate covering benchmarks. From the top, these are subsets from two-level logic minimization [23, 56], set covering derived from the Steiner triple systems [55], and randomly generated benchmark set [30]. Most relevant structural information is provided for each benchmark. Within each group, the benchmarks are sorted by the runtime of *cplex* on each instance. For *test4.pi*, *cplex* times out at 3600 seconds and it reaches a upper-bound is 101. It also times out on *steiner_a0081* but its optimum of 61 is proven in [60].

For each benchmark, we report the number of rows, columns, number of 1's (positive literals) and -1's (negative literals), the sparsity of the covering matrix, the cost of the optimal solution and the runtime of *cplex* to solve each instance. All variables in these benchmarks have unit weight. Each subgroup is sorted according to their *cplex* runtime. The same table format is also used in Tables 2.2–2.5.

Binate Covering Problems. *BCP* has the same formulation as *UCP* except that *BCP* can contain negative literals. It has extensive applications in logic synthesis [24, 19, 25]. The group of benchmarks in Table 2.2 is from the MCNC benchmark suite [56]. All variables in this group have unit weight.

Branch-and-bound algorithms for the covering problems have been studied in [23, 27, 28, 29, 30, 31]. State-of-the-art covering solvers include *scherzo* [28] and *aura*II [30]. As we will show in Chapter 4, *cplex* is also very competitive and can outperform *scherzo* and *aura*II on many industrial covering benchmarks. We will also introduce our covering solvers *eclipse-lpr* and *eclipse-cp* and compare them with other covering solvers in Chapter 4. *MinCostSat* solvers that are not specifically designed for covering problems, such as *bsolo* and *mindp*, are not competitive with the covering solvers above.

2.1.2 Non-covering Problems

The non-covering *MinCostSat* problems have applications in minimum-size test pattern problem [37], minimum-size planning [38], and minimum-cost goal models [39]. We study two sets of non-covering benchmarks shown in Table 2.3:

1. **Exact Minimum-size Test Pattern Problem.** The top group in Table 2.3 includes the five difficult benchmarks from the exact minimum-size test pattern problem set [37, 41]. We will refer to this group as the `ATPG` set. Most clauses in these benchmarks represent the nodes in a multi-level Boolean graph. In these benchmarks, the objective is to assign the minimum number of 1's to the variables corresponding the primary inputs of the Boolean graph. The number of such variables is less than or equal to 100 in each instance and that is only

Table 2.2: The binate covering benchmarks from logic synthesis.

						cplex	
benchmark	cols	Rows	# of 1	# of −1	sparsity	opt	time
count.b	466	694	16704	631	0.0536	24	0.7
clip.b	349	707	13837	668	0.0588	15	0.8
9sym.b	309	963	21675	944	0.0760	5	1.1
jac3	1731	1254	22801	2210	0.0111	15	1.5
f51m.b	406	520	12921	476	0.0634	18	1.9
sao2.b	372	772	12098	722	0.0446	25	3.6
5xp1.b	464	845	29084	805	0.0762	12	5.0
apex4.a	4316	11912	46579	11016	0.0011	776	9.9
rot.b	1451	2932	38126	2629	0.0096	115	12.5
alu4.b	807	1827	34518	1732	0.0246	≤ 50	3600*
e64.b	607	1022	7337	863	0.0132	≤ 48	3600*

* *cplex* times out at 3600 seconds.

This table presents the set of binate covering benchmarks from logic synthesis [56]. For *alu4.b* and *e64.b*, *cplex* times out and only the upper-bounds are reported.

a small fraction of all variables. They have unit weight and the other variables have zero weight.

2. **Sat.** The bottom group in Table 2.3 contains five satisfiable *Sat* benchmarks [57, 58]. Any non-trivial *Sat* benchmark also has the characteristics of a non-covering *MinCostSat* benchmark. We let all variables in these benchmarks have unit weight.

Branch-and-bound algorithms and solvers for non-covering *MinCostSat* problems have been investigated in [40, 41, 42]. Leading solvers for non-covering *MinCostSat* include *bsolo* [41] and *pbs* [42]. They are mainly based on *Sat* techniques and utilize the *conflict diagnosis* and *non-chronological backtracking* techniques introduced in SAT solvers [15, 16, 54]. Our work in the area of non-covering *MinCostSat* is limited and we don't consider *pbs* in this work. Our solver *eclipse-bf* is introduced in Chapter 6 and it is competitive with *bsolo* on the hardest ATPG benchmarks.

Table 2.3: The non-covering benchmarks from ATPG and *Sat*.

						cplex	
benchmark	cols	Rows	# of 1	# of -1	sparsity	opt	time
c432_F37gat@1	964	5054	6656	8235	0.0031	9	179.63
misex3_Fb@1	2346	12574	15953	19725	0.0012	≤ 13	3600*
c1908_F469@0	2824	12735	15352	19155	0.0010	≤ 12	3600*
c6288_F69gat@1	7537	36257	44055	54302	0.0004	NA◇	3600*
c3540_F20@1	7598	41312	52261	65546	0.0004	NA◇	3600*
queen19	361	10735	361	21432	0.0056	19	0.18
hanoi3	249	1512	1099	2615	0.0099	61	1.47
bw_large_b	1087	13772	6306	25461	0.0021	131	42.68
uf_250_1065_027	250	1065	1617	1578	0.0120	NA◇	3600*
sched07s_v1386	1386	25671	41474	49425	0.0026	NA◇	3600*

* *cplex* times out at 3600 seconds.

◇ *cplex* doesn't find a feasible solution.

This table presents two sets of native non-covering benchmarks. The top group is from ATPG [37, 41] and the bottom group is from *Sat* [57, 58, 59]. On the ATPG set, *cplex* times out on four out of five instances. For the last two instances, it fails to find any feasible solutions. On the *Sat* set, *cplex* also fails to find any feasible solutions for *uf250_1065_027* and *sched07s_v1386*.

2.1.3 MaxSat Problems

Max-Sat is the optimization version of *Sat* — the goal is to find the maximum number of satisfiable clauses in a CNF formula by any given assignment. More formally, given a CNF formula F with m clauses, the objective of *Max-Sat* is to maximize:

$$W = \sum_{i=1}^{m} v_i, \quad \text{where } v_i = \begin{cases} 1 & \text{if } y_i \text{ is satisfied} \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

A *Max-Sat* problem M can be easily transformed into an instance of *MinCostSat* B as follows: first, assign all variables in M a cost of 0. Then, for each clause y_i in M, add a new *slack variable* s_i and assign it a cost of 1. This formulation was first introduced in [62] and the slack variables were originally referred by as *weighted variables*. M and B clearly have the same number of clauses. Thus, we can find the maximum number of satisfiable clauses in a CNF formula by transforming it into a *MinCostSat* and finding its minimum cost solution.

To date, we could not identify a set of *Max-Sat* benchmarks that represent problems formulated for a specific application. Rather, the *Max-Sat* benchmarks available in the literature represent instances generated randomly. The benchmarks in Table 2.4 are derived from two series of *Max-Sat* benchmarks, one for *Max-2-Sat* and one for *Max-3-Sat* from the randomly generated set [46]. The slack variables have unit weight and the original variables have zero weight.

Leading branch-and-bound *Max-Sat* solvers include *maxsat* [46], *LB2+MOMS* and *LB2+JW* [47]. The integer linear programming approach is shown to be more effective than the branch-and-bound approach on *Max-2-Sat* benchmarks but less effective on *Max-3-Sat* benchmarks [46, 62]. We introduce a new branch-and-bound *Max-Sat* solver in Chapter 6 and compare it with other state-of-the-art solvers, including *cplex*. Experimental results show that our solver consistently outperforms *cplex* not only on *Max-3-Sat* benchmark but also for *Max-2-Sat* benchmarks.

2.1.4 Partial MaxSat Problems

Partial Max-Sat was first introduced in [43]. *Partial Max-Sat* is composed of two CNF formulas $F_h = \{y_{h1}, y_{h2}, \cdots, y_{hj}\}$ and $F_s = \{y_{s1}, y_{s2}, \cdots, y_{sk}\}$ over the same set of variables. The objective is to find an assignment that satisfies F_h (the hard constraints) and maximizes the number of satisfied clauses in F_s (the soft constraints),

Table 2.4: The non-native benchmarks from random *Max-2-Sat* and *Max-3-Sat*.

						cplex	
benchmark	cols	rows	# of 1	# of -1	sparsity	opt	time
2sat_v050_c200	50	200	218	182	0.0400	–	–
	250	200	418	182	0.0120	16	0.23
2sat_v050_c250	50	250	245	255	0.0400	–	–
	300	250	495	255	0.0100	22	0.57
2sat_v050_c300	50	300	309	291	0.0400	–	–
	350	300	609	291	0.0086	32	1.74
2sat_v050_c350	50	350	357	343	0.0400	–	–
	400	350	707	343	0.0075	41	5.8
2sat_v050_c400	50	400	414	386	0.0400	–	–
	450	400	814	386	0.0067	45	6.15
3sat_v050_c250	50	250	375	375	0.0600	–	–
	300	250	625	375	0.0133	2	3.35
3sat_v050_c300	50	300	481	419	0.0600	–	–
	350	300	781	419	0.0114	3	17.28
3sat_v050_c350	50	350	522	528	0.0600	–	–
	400	350	872	528	0.0100	8	66.01
3sat_v050_c450	50	450	680	670	0.0600	–	–
	500	450	1130	670	0.008	15	295.31
3sat_v050_c400	50	400	657	543	0.0600	–	–
	450	400	1057	543	0.0088	11	432.29

This table presents two sets of benchmarks from random *Max-2-Sat* and *Max-3-Sat* [46]. For each benchmark, the first row describes the original instance and the second row describes its corresponding *MinCostSat* instance. The number of *slack* variables for a *Max-Sat* instance is the same as the original number of clauses. Therefore, in the *MinCostSat* instance, the number of columns is equal to the sum of the rows and columns of the original instance. We don't report *cplex* information on the original *Max-Sat* instances because they cannot be solved by *cplex* directly. Within each group, the benchmarks are sorted according the *cplex* runtime of the *MinCostSat* instances.

in other words to maximize:

$$W = \sum_{i=1}^{k} v_i, \quad \text{where } v_i = \begin{cases} 1 & \text{if } y_{si} \text{ is satisfied} \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$
$$\text{subject to} \quad y_{h1} \wedge y_{h2} \wedge \cdots \wedge y_{hj} = 1$$

We can transform an instance of the *partial Max-Sat* problem into its corresponding *MinCostSat* problem by adding a slack variable for each of its soft constraints. Some applications of *partial Max-Sat* include the course assignment problems [61] as well as the two-level crossing number minimization problem [44]. Local search solvers [43, 53] exist. *Partial Max-Sat* is not studied further in this work.

2.1.5 Group-Partial MaxSat Problems

A variation of the *partial Max-Sat* problem arises from the FPGA detailed routing problem [13]. Here, the soft constraints are in groups $G = \{g_1, g_2, \cdots, g_l\}$. Each group is satisfied if and only if all the soft constraints in the group are satisfied. The objective is to maximize the number satisfied groups:

$$W = \sum_{i=1}^{l} v_i, \quad \text{where } v_i = \begin{cases} 1 & \text{if } g_i \text{ is satisfied} \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$
$$\text{subject to} \quad y_{h1} \wedge y_{h2} \wedge \cdots \wedge y_{hj} = 1.$$

In order to represent the grouping when transforming *group-partial Max-Sat* to *MinCostSat*, we introduce a slack variable for each group (not for each soft constraint).

In Table 2.5, we present ten *group-partial Max-Sat* benchmarks derived from the FPGA detailed routing problem [11, 13]. We explain in Chapter 6 how to formulate the routing problem as a *group-partial Max-Sat* problem. After converting to *MinCostSat*, the slack variables have unit weight and the other variables have zero weight. With a timeout of one hour for each benchmark, most instances from this set of benchmarks are unsolvable for *cplex* as well as other *MinCostSat* solvers. The *sub_SAT* approach [13] is more effective that *cplex* but still fails to solve some large benchmarks. In Chapter 6, we propose a local-search solver that can find known optima quickly and is able to find better than previously-known solutions on benchmarks whose optima remain unknown.

2.2 Our Contributions

Having gathered all *MinCostSat* benchmarks above provides us an opportunity to conduct a comprehensive study of *MinCostSat*. In this work, we study and make contributions in the following areas: improved covering algorithms and solvers, improved branch-and-bound *Max-Sat* solver, and new local-search algorithm for *group-partial Max-Sat* with application in FPGA detailed routing. We briefly highlight each one.

2.2.1 Branch-and-Bound UCP/BCP Algorithms

Many existing *UCP/BCP* benchmarks from logic synthesis are difficult for previous covering solvers and some benchmarks are unsolved prior to this work. In order to solve these open problems, we study seven critical performance factors in any branch-and-bound *MinCostSat* solver that specializes in the covering problems. On three of them, namely, lower-bounding techniques, upper-bounding techniques and search-tree exploration strategies, we make significant improvements by devising new algorithms with carefully designed experiments. Combining the best known techniques with the newly discovered ideas, our branch-and-bound covering solver, *eclipse*, outperforms state-of-the-art covering solvers on most of the logic synthesis benchmarks. This work is presented in Chapter 4.

For two-level logic minimization in particular, we replace the covering procedure in the exact two-level logic minimizer *espresso* [23] with *eclipse*. With the new logic minimizer, we solve some previously unsolvable benchmarks. This work is presented in Chapter 6.

Table 2.5: The non-native benchmarks from *group-partial Max-Sat* (FPGA detailed routing).

						cplex	
benchmark	cols	rows	# of 1	# of -1	sparsity	opt	time
term1_rcs_w3	606	2518	720	4632	0.0035	–	–
	694	2518	808	4632	0.0031	7	0.43
apex7_rcs_w4	1200	9416	1374	18232	0.0016	–	–
	1326	9416	1500	18232	0.0016	2	185.85
9symml_rcs_w5	1295	24309	1475	48100	0.0016	–	–
	1374	24309	1554	48100	0.0015	≤ 2	3600*
c499_gr_rcs_w5	1560	15777	1757	30930	0.0013	–	–
	1675	15777	1872	30930	0.0012	≤ 5	3600*
example2_gr_rcs_w5	2220	23144	2459	45400	0.0009	–	–
	2425	23144	2664	45400	0.0009	≤ 2	3600*
too_large_gr_rcs_w6	3114	43251	3447	85464	0.0006	–	–
	3300	43251	3633	85464	0.0006	na$^\diamond$	3600*
alu2_gr_rcs_w7	3570	73478	3927	145936	0.0006	–	–
	3723	73478	4080	145936	0.0006	na$^\diamond$	3600*
c880_rcs_w6	3936	53018	4358	104724	0.0005	–	–
	4170	53018	4592	104724	0.0005	na$^\diamond$	3600*
vda_rcs_w7	5054	102047	5551	202650	0.0004	–	–
	5279	102047	5776	202650	0.0004	na$^\diamond$	3600*
k2fix_rcs_w9	11313	305160	12166	607806	0.0002	–	–
	11717	305160	12570	607806	0.0002	na$^\diamond$	3600*

* cplex times out at 3600 seconds.

$^\diamond$ *cplex* doesn't find a feasible solution.

This table presents a set of benchmarks from the problem of optimizing FPGA detailed routing. Each benchmark has two rows: the first row describes the original benchmark as *group-partial Max-Sat* instance; the second row describes its BCP counterpart. The number of slack variables in the BCP instance is the same as the number of *groups* in the original instances (each group corresponds to a net).

2.2.2 Local-search Algorithms for FPGA Detailed Routing

Nam et al. [11] presented a *Sat* formulation for deciding the routability of a given FPGA and consequently a solution using *Sat* solvers. However, *Sat* solver cannot be used directly to determine the minimum number of unroutable nets for a unroutable layout. We refer to this problem as the *optimizing FPGA detailed routing* problem. Xu et al. present a novel *sub_SAT* approach by transforming the optimization problem into a series of decision problems. This approach works well for small and medium benchmarks but finds poor solutions for large ones. We show in Chapter 6 that the optimization problem can be viewed as a *group-partial Max-Sat* problem and therefore transformed into *MinCostSat*. However, branch-and-bound *MinCostSat* solvers remain ineffective in solving these benchmarks (shown in Table 2.5, *cplex* times out on all but two small benchmarks).

When optimality is presently not achievable, we resort to local-search algorithms that are not capable of proving optimality but can often find high quality solutions. We study and compare three approaches for optimizing FPGA detailed routing: (1) the *sub_SAT* approach, (2) an local-search approach specializing in optimizing FPGA detailed routing and (3) the generic local-search *MinCostSat* solver we present in Chapter 5.

2.2.3 Branch-and-Bound Algorithms for MaxSat

Branch-and-bound approaches for *Max-Sat* include: (1) branch-and-bound, (2) transform into *0-1 IP* and solve with *IP* solvers, and (3) transform into *MinCostSat* and solve with *MinCostSat* solvers. Comparison studies have been done on the branch-and-bound approach and the *IP* approach [62, 46]. We renew this study by first introducing a new state-of-the-art branch-and-bound *Max-Sat* solver – *qtmax*. We then compare *qtmax* (branch-and-bound approach) with *cplex* (*IP* approach) and *eclipse* (*MinCostSat* approach). The study shows a clear pattern of solver dominance relationships and it challenges the view in the literature that considers *IP* as the most competitive approach for *Max-2-Sat*. We present this study in Chapter 6.

2.2.4 Branch-and-Bound Algorithm for Minimum-Size Test Pattern Problem

In the field of automatic test pattern generation (`ATPG`), the *MinCostSat* formulation is the first formal non-heuristic model towards computing minimum-size test patterns [37]. Even though the `ATPG` benchmarks in Table 2.3 can contain thousands of variables, only a small fraction of them corresponds to the primary inputs of the underlying multi-level Boolean graph. In their *MinCostSat* formulation, the primary-input variables are very important: (1) only these variables contribute to the cost function because they have unit weight while the rest of the variables have zero weight, (2) the assignments for these variables largely determine the values of the rest of the variables. Traditional *MinCostSat* solvers perform poorly on these `ATPG` benchmarks mainly because they don't exploit the unique properties of the primary-input variables. In addition, Section 4.4.1 shows that the lower-bounding techniques in current *MinCostSat* solvers perform poorly on these benchmarks. A recent *MinCostSat* solver *bsolo* [41] leads all solvers on these benchmarks by utilizing SAT-based learning and non-chronological backtracking techniques.

We take a different approach on the `ATPG` benchmarks and do not use any expensive lower-bounding or SAT-based learning. Our solver explores the search space as fast as possible with only basic lower-bounding. As a result, our approach explores many more nodes than *bsolo* but can be significantly faster on some benchmarks. This work is presented in Chapter 6.

2.3 Summary

In this chapter, we presented various cases of *MinCostSat* along with their benchmarks and solvers. For the native ones, we presented both the covering problems and non-covering problems, which require different treatments when designing an efficient solver. Solvers that are designed for non-covering problems are not efficient on covering problems and vice versa. For the non-native ones, we gave the formulation of *Max-Sat*, *partial Max-Sat*, and *group-partial Max-Sat* and showed how to transform them into *MinCostSat*. Finally, we gave an overview of the contributions of this work. In the next chapter, we start the main body of this thesis by surveying branch-and-bound algorithms for *MinCostSat*.

Chapter 3

Survey of Branch-and-Bound Algorithms

There are mainly two types of branch-and-bound algorithms for solving *MinCostSat*. For the covering problems, the classical branch-and-bound algorithms work the best; for the non-covering problems, the SAT-based algorithms have been shown to be more effective [41]. In this chapter, we survey these two approaches with an emphasis on the first approach. This lays a foundation for our branch-and-bound *MinCostSat* algorithm to be introduced in Chapter 4, which specializes in the covering problems.

3.1 Classical Branch-and-Bound Algorithm

A branch-and-bound algorithm for *MinCostSat* can be captured in a search tree. The root of the search tree is initialized with the original covering instance. If applicable, reductions are applied to simplify the covering matrix. We define *upper bound* to be the objective value of the best solution found and *upper-bounding* to be the process that finds a upper bound. We also define *lower bound* to be the sum of the minimum cost to satisfy the current covering matrix and the cost of the path from the root leading to the current node. *Lower-bounding* is the process of estimating the lower bound. The search can backtrack when the lower bound is no less than the upper bound. Otherwise, a branching variable is chosen to generate two children of the parent, after setting the branching variable to either 0 or 1. Branch-and-bound algorithms avoid exhaustively enumerating all the possible solutions by backtracking early. The classical branch-and-bound algorithm has four main ingredients: (1) polynomial time reductions, (2) lower bounding, (3) search-tree pruning, and (4) branching variable selection. The first two tend to be effective on the covering problems but have little effect on non-covering problems. We introduce each of the ingredients in turn.

3.1.1 Reduction Techniques

Three polynomial-time reduction techniques can be used to simplify the covering matrix[1] for *MinCostSat*, namely, essentiality, row dominance and column dominance [1].

3.1.1.1 Essentiality

Definition 4 *An essential row of a covering matrix F is a row with one literal.*

The literals in the essential rows are called essential literals and their corresponding variables are called essential variables. In order to cover the essential rows, the essential literals have to be set true. For each essential literal, F can be reduced (simplified) by removing all rows containing the essential literal and removing the complements of the essential literal from the other rows. Consider this example,

[1]An example of the covering matrix can be found on page 5.

$$\begin{array}{c} \begin{array}{ccccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \end{array} \\ \begin{bmatrix} 1 & -1 & 1 & -1 & 1 & 1 & -1 \\ 1 & -1 & 0 & -1 & 1 & 1 & -1 \\ 1 & 1 & -1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & -1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Row 6 is an essential row and x_1 is the essential variable that must have the value false. The covering matrix can be simplified as:

$$\begin{array}{c} \begin{array}{cccccc} x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \end{array} \\ \begin{bmatrix} -1 & 1 & -1 & 1 & 1 & -1 \\ -1 & 0 & -1 & 1 & 1 & -1 \\ 1 & -1 & 1 & 0 & 1 & 0 \\ 0 & 1 & -1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

After the reduction using essentiality, new essential variables can arise and the reduction process can be repeated. The idea of essentiality is the same as the *unit propagation* in the *Sat* literature. Assume a literal can be located in the covering matrix in constant time, the time complexity for essentiality reductions is $O(l)$, where l is the number of literals (non-zeros) removed.

3.1.1.2 Row Dominance

Definition 5 *Let row_i and row_j be two rows in the covering matrix F, then row_i dominates row_j if for every $column_k$, one of the following situations occurs:*

1. $F_{ik} = 1$ *and* $F_{jk} = 1$

2. $F_{ik} = -1$ *and* $F_{jk} = -1$

3. $F_{jk} = 0$

In other words, row_i *dominates* row_j *if* row_i *is satisfied whenever* row_j *is satisfied.*

A dominating row can be removed from the formula without affecting the optimal solution because any solution satisfying the dominated row also satisfies the dominating row. In our running example, after removing the essential variable, row_1 dominates row_2. We can remove row_1 and get:

$$\begin{array}{cccccc} x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \\ \end{array}$$
$$\begin{bmatrix} -1 & 0 & -1 & 1 & 1 & -1 \\ 1 & -1 & 1 & 0 & 1 & 0 \\ 0 & 1 & -1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Reduction using row dominance can be implemented using the algorithm presented in Figure 3.1.

In the algorithm in Figure 3.1, for each row_i, the loop in line 2–18 finds all its dominating rows and removes them from the covering matrix F. We say a row_i and a column $column_j$ intersect when $F_{ij} \neq 0$. On line 2, we find the shortest column intersecting with row_i in order to minimize the rows we have to consider as possible dominating rows of row_i. The function `row-dominates` reflects the row dominance definition. We define the length of a row (column) as the number of nonzero entries on that row (column). The algorithm runs in $O(m \cdot c_{max} \cdot r_{max})$ where m is the number of rows and c_{max} (r_{max}) is the maximum length of any column (row). The idea of row dominance is equivalent to the *subsumption* operation used in some *Sat* solvers: if clause A is satisfied whenever clause B is satisfied (A is subsumed by B), then neither

```
Algorithm: Row-Dominance-Reduction( $F$ )
input: a covering matrix $F$
output: $F$ after applying row dominance reduction
method:
 1 did-reduction = false
 2 for each $row_i$ do
 3     Let $column_k$ be the shortest column intersecting with $row_i$
 4     for each $row_j$ intersecting with $column_k$ where $j \neq i$ do
 5         if $length(row_j) > length(row_i)$, then
 6             if row-dominates($row_j, row_i$)
 7             then $F = F - row_j$
 8                     did-reduction = true
 9             endif
10         endif
11         if $length(row_j) = length(row_i)$ and $j > i$
12             if row-dominates($row_j, row_i$)
13             then $F = F - row_j$
14                     did-reduction = true
15             endif
16         endif
17     end do
18 end do
19 return did-reduction

20 function row-dominates($row_m$, $row_n$)
21     for each $column_k$ in $row_n$ do
22         if $F_{nk} \neq 0$ and $F_{nk} \neq F_{mk}$
23         then return false
24         endif
25     end do
26     return true
```

Figure 3.1: The row dominance reduction algorithm.

the satisfiability nor the solutions of a formula are affected by removing clause A from the formula.

3.1.1.3 Column Dominance

Definition 6 *Let $column_j$ and $column_k$ be two columns in the covering matrix F, then $column_j$ dominates $column_k$ if, for each row_i, one of the following situations occurs:*

1. $F_{ij} = 1$
2. $F_{ij} = 0$ *and* $F_{ik} \neq 1$
3. $F_{ij} = -1$ *and* $F_{ik} = -1$

Column dominance is defined in such a way that when $x_k = 0$, the rows containing $\overline{x}_k$ can be satisfied by $x_k = 0$ and the rows containing x_k can be satisfied by $x_j = 1$. This leads to the following theorem [1]:

Theorem 1 *Let F be satisfiable. If $column_j$ dominates $column_k$ and $cost(column_j) \leq cost(column_k)$, then there is at least one minimum solution with $x_k = 0$.*

Going back to our running example (after applying essentiality and row dominance), x_5 is dominated by x_4. Therefore, the covering matrix:

$$\begin{array}{cccccc} x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \\ \end{array}$$
$$\begin{bmatrix} -1 & 0 & -1 & 1 & 1 & -1 \\ 1 & -1 & 1 & 0 & 1 & 0 \\ 0 & 1 & -1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

can be further reduced by setting $x_5 = 0$ to:

$$\begin{array}{ccccc} x_2 & x_3 & x_4 & x_6 & x_7 \\ \end{array}$$
$$\left[\begin{array}{ccccc} -1 & 0 & -1 & 1 & -1 \\ 1 & -1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{array}\right]$$

Notice that now x_6 dominates x_7 and the covering matrix can be further reduced.

The algorithm for column dominance reduction is very similar to that for row dominance reduction. In the algorithm in Figure 3.2, for each $column_i$, if it doesn't contain any 1's, then we can assign $x_i = 0$ without increasing the cost function (line 3 through line 6). Line 7 through line 15 attempts to find one dominating column of $column_i$ and if one is found, we assign x_i to 0. On line 2, we find the row with the minimum number of $1's$ that intersects with $column_i$ on a 1. We choose this row in order to minimize the columns we have to consider as possible dominating columns of $column_i$ (if $column_i$ has a 1 on row_k, then its dominating column must have a 1 on row_k too.) Lines 20 and 22 in the function `column-dominates` enumerate the cases by which we can determine that $columm_m$ doesn't dominate $column_n$. These cases are basically the complement of the cases listed in Definition 3.1.1.3. The column dominance reduction algorithm runs in $O(n \cdot r_{max} \cdot c_{max})$ where n is the number of columns.

We are not aware of an equivalent operation that detects column dominance in *Sat* solvers. Column dominance is definitely applicable in the context of *Sat* because *Sat* is a special case of *MinCostSat* in which all variables have zero costs. Assume that $column_k$ is dominated in a *Sat* instance F. If F is satisfiable, then there is a solution with $x_k = 0$; therefore, we can eliminate the branch where $x_k = 1$. If F is unsatisfiable, then its unsatisfiability is not changed by setting $x_k = 0$ and we don't have to explore the branch where $x_k = 1$. In both cases, it is an open question whether the gain from eliminating such a branch is sufficient to overcome the overhead of detecting column dominance relationships.

```
Algorithm: Column-Dominance-Reduction( $F$ )
input: a covering matrix $F$
output: $F$ after applying column dominance reduction
method:
 1 did-reduction = false
 2 for each $column_i$ do
 3     if $column_i$ doesn't contain any $1's$
 4     then $F = F(x_i \rightarrow 0)$
 5             did-reduction = true
 6     endif
 7     Let $row_k$ be the row with the minimum number
 8     of $1's$ that intersects with $column_i$ on a 1
 9     for each $column_j$ intersecting with $row_k$ where $j \neq i$ do
10         if column-dominates($column_j, column_i$)
11         then $F = F(x_i \rightarrow 0)$
12                 did-reduction = true
13                 break
14         endif
15     end do
16 end do
17 return did-reduction

function column-dominates($column_m$, $column_n$)
18 if cost($column_m$) > cost($column_n$), then return false endif
19 for each $row_k$ in $column_m$ do
20     if $F_{km} = 0$  and $F_{kn} = 1$
21     then return false
22     else if $F_{km} = -1$  and $F_{kn} \neq -1$
23     then return false
24     endif
25 end do
26 return true
```

Figure 3.2: The column dominance reduction algorithm.

The three reduction techniques: essentiality, row dominance and column dominance all can be done in polynomial time. However, when no more reduction techniques can be applied, the covering matrix becomes *cyclic* and branching is needed. For example, the following covering is cyclic:

$$\begin{array}{c} \begin{array}{ccc} x_1 & x_2 & x_3 \end{array} \\ \begin{bmatrix} -1 & 0 & 1 \\ 1 & -1 & 0 \\ -1 & 1 & 0 \end{bmatrix} \end{array}$$

3.1.1.4 Order of Applying Reductions

Given the essentiality and the dominance reduction techniques, does the order matter when several reductions can be applied at the same time? To answer this question, we do a case-by-case analysis to see whether the order between any two reductions makes a difference.

Row dominance and column dominance. Reduction by row dominance and reduction by column dominance are two orthogonal operations; therefore, the order between these two reductions is irrelevant.

Essentiality and column dominance. The order here does make a difference. Consider the following example:

$$\begin{array}{c} \begin{array}{cccc} x_1 & x_2 & x_3 & x_4 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & -1 & 1 & -1 \\ 0 & 0 & -1 & -1 \\ -1 & -1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Notice that x_1 is the essential variable and $column_1$ dominates $column_2$. If we apply reduction by essentiality first by setting $x_1 = 1$, we have:

$$\begin{array}{ccc} x_2 & x_3 & x_4 \\ \left[\begin{array}{ccc} -1 & 1 & -1 \\ 0 & -1 & -1 \\ -1 & 0 & 1 \end{array}\right] \end{array}$$

The covering is cyclic at this point (the original *column*$_1$ is eliminated and *column*$_2$ is no longer dominated). However, if we apply column dominance first, we can let $x_2 = 0$ and have:

$$\begin{array}{ccc} x_1 & x_3 & x_4 \\ \left[\begin{array}{ccc} 1 & 1 & 1 \\ 0 & -1 & -1 \\ 1 & 0 & 0 \end{array}\right] \end{array}$$

We can then apply reduction by essentiality and set $x_1 = 1$ to get:

$$\begin{array}{cc} x_3 & x_4 \\ \left[\begin{array}{cc} -1 & -1 \end{array}\right] \end{array}$$

The column x_4 now dominates x_3 (or vice versa) and we can set $x_3 = 0$, $x_4 = 1$ to get the optimal solution. This example shows that by doing reduction with essentiality first, we may lose some reduction power. Is it possible to lose reduction power when we do column dominance first? The answer is no. The only case an essential variable x_i can be eliminated by column dominance is when x_i appears negatively in the essential row and the column for variable x_i is dominated. In this case, column dominance reductions set $x_i = 0$ and reduce the covering matrix, the exact same way as the reduction by essentiality would have done. Therefore, column dominance doesn't affect essentiality when doing reduction. As a result, *we always want to do column dominance reduction prior to doing reduction with essential variables.*

```
Algorithm: Reduction ( $F$ )
 **input:** a covering matrix $F$
 **output:** the cyclic core of $F$
 **method:**
 reduced = *true*
 **while** reduced = *true*
     reduced = *false*
     **if** column-dominance-reduction($F$), **then** reduction = *true*
     **if** essentiality-reduction($F$), **then** reduction = *true*
     **if** row-dominance-reduction($F$), **then** reduction = *true*
 **end do**
 **return**
```

Figure 3.3: The overall reduction algorithm.

Essentiality and row dominance. If we do reduction by essentiality first, then, for two rows with a dominance relationship before the reduction, the relationship still exists after the reduction because reduction by essentiality removes the same column elements from both rows. If we do reduction with row dominance first, then the essential variable won't be affected because the essential row can't be the dominating row and won't be eliminated.

Summary When doing reduction, we should always carry out reduction with column dominance before we apply essentiality. Row dominance can be applied in any order with respect to column dominance and essentiality. The overall reduction is presented in Figure 3.3. The main while loop repeats as long as one of the three reduction techniques has been applied (indicated by reduction = *true*). The end product of this algorithm is the cyclic core of the covering matrix. In branch-and-bound *MinCostSat* solvers, such as *scherzo* [28], reduction is applied at each node before branching. However, at any given node, it is possible that no reduction can be applied.

For any branch-and-bound algorithm, good lower-bounding and search pruning techniques are essential in reducing the amount of search. Next, we present these techniques as they are used for *MinCostSat*.

3.1.2 Lower-Bounding Technique

Given any node n_r and the search-tree rooted at this node T_r, let $Cost_r$ be the lower bound of the best-cost solution in T_r. Then $Cost_r = Cost_{path} + Cost_{lower}$, where $Cost_{path}$ is the cost of the path from the root to n_r and $Cost_{lower}$ is the lower-bound on the cost of covering T_r. Let UB be the global upper-bound of the optimal solution, given by the best solution found so far. The search can backtrack once $Cost_{path} + Cost_{lower} \geq UB$. Both $Cost_{path}$ and UB are fixed for any given node. This means that much effort should go into estimating $Cost_{lower}$. There are two approaches for doing lower-bound estimation. The first one is based the maximum independent set of rows and the second one is based on linear programming relaxation. We present them next.

3.1.2.1 MIS-Based Lower Bounding

Maximum Independent Set of Rows. A lower-bound on the cost of covering a matrix is provided by the cost for covering its maximum independent set of rows (MIS). Two rows are independent if it is not possible to cover both by setting at most one variable to true. In the case of *UCP* two rows are independent of each other if they don't have any intersecting columns. In the case of *MinCostSat*, however, any row_i containing a -1 is dependent on any other row because row_i can be satisfied by setting the complemented variable to 0. Because the rows in an independent set don't intersect, at least one variable from each row in the independent set has be true in order to cover the whole matrix. Therefore, the cost of covering the MIS provides a lower-bound on the best solution that covers the whole matrix. However, MIS-based lower-bound can be arbitrarily far from the minimum solution cost even in the *UCP* case. Courdert provides a covering matrix made of n rows and $n(n-1)/2$ columns such that any pair of rows is covered by only one column and each column covers only two rows [28]. The following figure shows such a matrix with $n = 6$:

$$\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1
\end{bmatrix}$$

Assuming unit cost for all variables, the MIS has a size of 1 and provides a lower-bound of cost 1. However, the minimum cost solution consists of setting $\lceil n/2 \rceil$ variables to 1.

Basic MIS Construction. Not only can the size of MIS be far from the minimum cost of the solutions, finding the size of the MIS is also NP-hard [28]. For this reason, we have to construct a *maximal* independent set of rows using greedy heuristics. We present the one from [28] in Figure 3.4.

```
Algorithm: Greedy-MIS-Construction
input: a covering matrix $F$
output: the size of a set of independent rows of $F$
method:
Let $Y$ be the rows in $F$
$MIS = \emptyset$
while $Y \neq \emptyset$ do
    choose a random row $y$ from $Y$
    remove all rows intersecting with $y$ from $Y$
    $MIS = MIS \cup y$
end do
return $|MIS|$
```

Figure 3.4: The greedy algorithm for constructing the MIS and calculating the lower-bound.

Intuitively, two rows are intersecting when at least one variable occurs in both rows. The approximation ratio of the best known algorithm for selecting a maximal independent set is $O(m/(\log(m^2))$ [63].

Coudert [28] presented a log-approximation algorithm for the lower bound that is not based on MIS. It can provably obtain a tighter bound than the algorithm presented above in the worst case; however, the former doesn't perform well in practice because (1) many standard benchmarks are sparse, for which the bounds obtained from the two methods have small differences, (2) the log-approximation algorithm, not based on MIS, cannot take advantage of the *limit lower-bound* pruning technique, which we will present in Section 3.1.3 when we discuss the search pruning techniques.

Heuristic MIS Construction. As shown in Figure 3.4, once a row is selected, all its intersecting rows are removed from the covering matrix. Let $IR(x)$ denote the intersecting rows of row x. The larger $|IR(x)|$ is, the fewer rows remain to build a large MIS. Therefore, a simple heuristic that chooses the row x with minimum $|IR(x)|$ has been used in [23].

It was claimed in [28] that it helps to look-ahead one more level. Removing $I(y)$ from Y decreases the size of $I(y')$ for all y' such that $I(y) \cap I(y') \neq \emptyset$. The heuristic we describe below breaks ties by selecting the row y that maximizes $\sum_{y' \in I(y)} |I(y')|$. Let $IC(y)$ be the columns that intersect with y. We define for a row y,

$$weight(y) \quad = \quad \text{Min cost(x) where x} \in IC(y). \tag{3.1}$$

It was proposed that the row to select should minimize:

$$\frac{1}{weight(y)} \sum_{y' \in I(y)} \frac{weight(y')}{|I(y')|}. \tag{3.2}$$

This is a costly heuristic because every time when a row y is removed, each $weight(y')$ and $|I(y')|$ have to be updated. Consequently, the cost function for many other column variables have to be updated before adding the next row to the MIS. However, we observe that the high quality lower bounds obtained with this heuristic help to speed up the search significantly compared to the MIS construction method in Figure 3.4. We refer to the new heuristic as *Coudert's heuristic* and the resulting lower-bounding technique as *MIS1*.

Further Improvement. In Figure 3.4, when a row y is selected, all rows in $I(y)$ are removed from the covering matrix. On first sight, this is necessary because for the purpose of constructing the MIS, no rows in $I(y)$ are allowed to contribute to the remaining MIS construction because these rows can be potentially covered by a variable in y. However, when no variable exists that can cover y and all rows in $I(y)$, the removal of $I(y)$ is not necessary and the lower-bound can be bumped up further. For example, consider the following matrix:

$$\begin{array}{cccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \\ \end{array}$$
$$\left[\begin{array}{cccccc} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{array}\right]$$

If the first row is chosen to be in the MIS, then the second and third rows are removed because they intersect with the first row. The last row is then added the MIS and the size of the MIS is 2. Let V_{MIS} be all the variables that occur in the rows of the MIS. In this example, $V_{MIS} = \{x_1, x_2, x_3, x_4, x_6\}$. Notice that no variable in V_{MIS}, if set to 1, covers the first three rows at the same time. Therefore, at least one more variable has be set to 1 and we can increase the lower-bound by 1 in addition to the size of the MIS. In the general case, for each row y in the MIS and its intersecting rows, we can check whether the situation above occurs and increment the lower-bound when appropriate.

The new lower-bounding algorithm (not previously known in the literature) is described in Figure 3.5. On line 1, we initialize the variable *bonus* that will record how much we can increment from the size of the MIS when doing lower bounding. Lines 5–10 construct the MIS and store in V_{MIS} all the variables that occur in the MIS. Bear in mind that, if a row contains -1, then it won't be put into the MIS because it is dependent on all other rows. On lines $11 - 16$, for each row y we check whether there is a variable in V_{MIS} that can cover y and $I(y)$. If not, we can increment

```
Algorithm: Improved-Lower-Bound-Calculation
 input: a covering matrix $F$
 output: a lower-bound of the covering matrix
 method:
  1 bonus = 0
  2 Let $X$ be the rows in the covering matrix
  3 $MIS = \emptyset$
  4 $V_{MIS} = \emptyset$
  5 while $Y \neq \emptyset$ do
  6     choose a row $y$ from $Y$ using Coudert's heuristic
  7     remove $I(y)$ from $Y$
  8     $MIS = MIS \cup y$
  9     $V_{MIS} = V_{MIS} \cup \{x_i | x_i \in y\}$
 10 end do
 11 for each $y \in MIS$ do
 12     if no variable in $V_{MIS}$ covers all rows in $y$ and $I(y)$
 13     then bonus = bonus + 1
 14             $V_{MIS} = V_{MIS} \cup \{x_i | x_i \; occurs \; in \; I(y)\}$
 15     endif
 16 end do
 17 return $|MIS| + bonus$
```

Figure 3.5: The improved algorithm for constructing MIS and calculating the lower-bound.

the lower-bound by 1 and update V_{MIS}. In practice, we observe that this strategy can often return a lower-bound that is slightly greater (normally 1–3) than the size of the MIS itself. We refer to this improved lower-bounding technique as *MIS2*. A comparison of MIS1 and MIS2 will be done in Section 4.4.1.

3.1.2.2 LPR-based Lower-Bound

We have shown in Section 1.2.1 that *MinCostSat* can be formulated as *0-1 IP*. When the integral constraints are removed, the problem becomes linear programming (*LP*). Many polynomial algorithms such as the *interior point* method have been developed to solve *LP*. Liao and Devadas [29] proposed an algorithm that is similar to the classical branch-and-bound algorithm but with a new lower-bounding technique using linear-programming relaxation (LPR). At each node of the search-tree, an LP instance is constructed from the current covering matrix. A general purpose LP solver is then used to solve the LP instance. If C_{opt} is returned as the optimal objective value, then it is safe to take $\lceil C_{opt} \rceil$ as the lower-bound of the IP instance. The experiments of Liao et al. show that the LPR-based lower-bounding method, even though more expensive, can almost always find better lower-bounds than MIS-based method. For example, the MIS-based lower-bounding technique gives a lower-bound of cost 1 for the covering matrix on page 42 whereas LPR gives a lower-bound of 3.

3.1.3 Search Pruning Technique

The standard search procedure backtracks as soon as $Cost_{path} + Cost_{lower} \geq UB$. Two improvements were presented in [28]. The first improvement is based on the following theorem:

Theorem 2 *(C_l-Lower-Bound [28]) Let C be a binate covering problem and x be an unassigned unate variable. Let $Cost.l_{lower}$ and $Cost.r_{lower}$ be the lower-bound on the cost of C_l (the branch after setting x to 1) and C_r (the branch after setting x to 0), respectively. If $Cost_{path} + Cost.l_{lower} \geq UB$, then both C_l and C_r can be pruned.*

Proof. For any clause y in C, if it contains x, then y is eliminated from C_l when x is set 1; however, the clause $y - x$ is still in C_r. If c doesn't contain x, then c is in both C_l and C_r. Therefore, $Cost.r$, the cost of covering C_r is no less than the cost of covering C_l because C_r contains all clauses in C_l and some extra clauses. Hence, if $Cost_{path} + Cost.l_{lower} \geq UB$, then clearly $Cost_{path} + Cost.r \geq UB$. Therefore, both C_l and C_r can be pruned. $\square$

It was claimed in [28] that *C_l-Lower-Bound* can reduce the number of branchings by about 5%. The cost of utilizing *C_l-Lower-Bound* pruning rule is minimal: at each branching point, the condition is checked, which takes constant time.

Theorem 3 *(Limit Lower-Bound) [28] Let Y_{MIS} be an independent set of rows and $C_{lower} = |Y_{MIS}|$ be the lower-bound. If there exists a variable x_i not intersecting with Y_{MIS} such that $Cost_{path} + Cost_{lower} + Cost(x_i) \geq UB$, then the branch with $x_i = 1$ can be eliminated and the covering matrix can be simplified by letting $x_i = 0$.*

Clearly, if $x_i = 1$, then the lower-bound of the current branch becomes $Cost_{path} + Cost_{lower} + Cost(x_i)$ and since it is greater than UB, the search has to backtrack. Branch-and-bound algorithms using MIS-based lower-bounds can take advantages of the limit lower-bound pruning technique by checking all unassigned variables not in the MIS and see whether the theorem above applies. This is a relatively cheap operation: the complexity of checking for *limited lower-bound* is $O(m \cdot |MIS|)$. In practice, the lower-bound of the newly created covering problem almost always exceeds the global upper-bound and this causes the search to backtrack right away. Experiments in [28] show that the *limit lower-bound* pruning technique is extremely effective and can speed up the solver for three orders of magnitude on some benchmarks. We also observe similar speedup with limit lower-bound in our experiments.

3.1.4 Branching Variable Selection

The choice of branching variable can also have a big impact on the efficiency of a branch-and-bound solver. Heuristics for choosing branching variable in *UCP* and *BCP* solvers are discussed in [23, 28].

The heuristics tend to favor column variables with the following properties:

1. A column that covers many rows. It is assumed that such variables are more likely in the optimal solution.
2. A column that covers many *short* rows. It is assumed that the short rows are harder to cover and worth exploring first.

For a row y, define

$$length(y) = |\{x \in X | x \in IC(y)\}|$$

Then the heuristic in [28] chooses the column that maximizes:

$$\frac{1}{Cost(x)} \sum_{y \in IC(x)} \frac{weight(y)}{length(y)}. \tag{3.3}$$

This heuristic is also used in our covering solver *eclipse*, to be introduced in Chapter 4.

3.1.5 The Classical Branch-and-Bound Algorithm

Now, we present a recursive branch-and-bound algorithm for *MinCostSat* introduced in [1]. The `Reduce`, `Lower-Bound`, and `Choose-Branching-Variable` functions in Figure 3.6 can be implemented using the ideas we presented earlier in this chapter. Note that the function `Lower-Bound` returns the sum of $Cost_{path}$ and $Cost_{lower}$.

```
Algorithm: BCP( $F$, $UB$, $solution$ )
$(F, solution)$ = Reduce$(F, solution)$
if all rows are covered
    if $F \neq 0$ and $Cost(solution) < UB$ then
        $UB = Cost(solution)$
        return $solution$
    else
        return “no solution”
    endif
endif
$lower\text{-}bound$ = Lower-Bound$(F, solution)$
if $lower\text{-}bound \geq UB$
    return “no solution”
endif
$x_i$ = Choose-Branching-Variable$(F)$
S1 = BCP$(F(x_i)$, $upper\text{-}bound$, $solution \cup \{x_i\})$
S2 = BCP$(F(-x_i)$, $upper\text{-}bound$, $solution)$
return Best-Solution$(S_1, S_2)$
```

Figure 3.6: The branch-and-bound algorithm for *MinCostSat* in [1].

3.2 SAT-based Algorithms

SAT-based branch-and-bound algorithms, unlike the classical branch-and-bound algorithms, work better on non-covering problems. Reductions and lower-bounding techniques play a much smaller role in these algorithms: (1) most of the time, the reduction techniques do not apply, (2) the lower-bound found is often too far from the upper-bound to help with search pruning. In this section, we briefly introduce two SAT-based branch-and-bound algorithms for solving *MinCostSat*. The first SAT approach does a linear search for the optimum by generating a sequence of intermediate decision problems and solving them with *Sat* solvers. The second one is more sophisticated and it does not create intermediate problems.

3.2.1 SAT-based Linear Search Algorithm

The SAT-based linear search algorithm [64, 65] solves *MinCostSat* by doing a linear search on the cost function. In addition to the feasibility constraints, constraints ensuring a solution with cost lower than *UB* are also incorporated into the CNF formula. This is done by encoding a multilevel adder and the bound of the sum with a CNF formula [64]. The value of *UB* is initialized to be just above the highest possible value:

$$UB = 1 + \sum_{j=1}^{n} cost(x_j) \tag{3.4}$$

If the *Sat* formula is satisfiable, then *UB* is replaced by the cost value of the solution found:

$$UB = \sum_{j=1}^{n} cost(x_j)x_j \tag{3.5}$$

The *Sat* formula is then reformulated with the new *UB*. The process continues until the formula is no longer satisfiable. The solution with a cost of the last *UB* becomes the optimal solution.

The advantage of the SAT-based linear search approach is that it can directly use general-purpose *Sat* solvers that are extremely efficient. The shortcoming is that many intermediate problem instances could be generated and each of them can be difficult to solve. Experiments in [41] show that the solvers built on this type of algorithms are not as competitive as the other state-of-the-art *MinCostSat* solvers.

3.2.2 SAT-based Branch-and-Bound Algorithm

Unlike the linear search approach, this branch-and-bound approach [41, 42] works directly on the original formula without generating intermediate instances. It incorporates the search pruning techniques from *Sat* by doing non-chronological backtracking [66] on both logical conflicts and bounding conflicts. Logical conflicts arise when a constraint is violated. A bounding conflict arises when the cost of the current (partial) solution has a cost no less than the *UB*. It was shown in [41] that on the covering problems, *Sat*-based solvers are not competitive with traditional covering solvers; however, on the non-covering benchmarks from `ATPG`, *Sat*-solvers are much more efficient.

3.3 Summary

In this chapter, we have surveyed branch-and-bound algorithms for *MinCostSat* in the literature. For the classical branch-and-bound algorithms, we presented the reduction, lower-bounding, search pruning, as well as branching variable selection techniques. We also briefly discussed two SAT-based algorithm for *MinCostSat*. Classical branch-and-bound algorithms are mainly designed to solve the covering problems, on which both reduction and lower-bounding techniques can be very effective. However, for non-covering problems, *Sat*-based algorithms are more efficient by utilizing the learning techniques used *Sat* solvers. In the next chapter, we concentrate on improving the classical branch-and-bound method and present a new branch-and-bound *MinCostSat* solver that specializes in solving the covering problems.

Chapter 4

Engineering an Efficient Branch-and-Bound MinCostSat Solver

Despite the advances in branch-and-bound covering solvers in recent years, many current benchmarks remain unsolvable by the leading solvers. In this chapter, we introduce *eclipse*, a new branch-and-bound solver that improves the state of the art for solving the covering problems. After presenting the outline of the *eclipse* algorithm, we describe the three key features in this new solver. For each of the three features and their corresponding components in *eclipse*, we explain the rationale behind our implementation by comparing them with other alternatives. In the experimental section, we compare two variations of *eclipse* with state-of-the-art *MinCostSat* solvers, especially its nearest competitor *cplex*, on a wide range of unate and binate benchmarks. Last, we conclude with some future plans.

4.1 Introduction to the Eclipse Algorithm

Unlike the recursive branch-and-bound algorithm presented in Figure 3.6, the *eclipse* algorithm is iterative. The concept of a *node* in *eclipse* is similar to the concept of a node in any search tree. In the context of *eclipse*, a node contains the current covering matrix and its lower-bound, which can be calculated from the covering matrix itself. The outline of the *eclipse* algorithm is as follows.

Step 1. Initialize the root of the search-tree with the original covering matrix and set its lower-bound to 0. Put the root node on a priority queue.

Step 2. If the priority queue is empty, then return the global upper-bound and terminate. Otherwise, the node in the front of the priority queue is removed. If the lower-bound of this node is no less than the current global upper-bound, then repeat step 2.

Step 3. A branching variable is chosen and the two children of the node are generated by setting the branching variable to 0 and 1. For each node, (1) apply reduction techniques to its covering matrix, (2) apply lower-bounding techniques to calculate the lower-bound of the node — if the lower-bound is no less than the global upper-bound, then discard the node; and (3) do a local search at the node; if a solution cost less than the global upper-bound is found, update the upper-bound with the new lowest cost. Otherwise, put the node in the priority queue. Go back to step 2.

4.2 Key Features

Besides implementing many of the best features in the literature, such as a *sparse matrix* data structure and the reduction algorithms (presented in Chapter 3), *eclipse* also introduces three new features to optimize the search:

1. **Advanced Lower-bounding Techniques:** The lower-bounding technique is one of the most important factors in designing *any* branch-and-bound solver.

Besides the MIS-based lower-bounding techniques, *eclipse* also implements two advanced lower-bounding methods: LPR (linear programming relaxation, first introduced in [29]) and CP (cutting planes, first used for lower-bounding in this work). Our experiments show that on the same benchmark, the lower-bounds given by LPR and CP are almost always better and never worse than the lower-bounds given by the MIS-based methods. In addition, on some benchmarks whose optimal solutions were previously unknown, CP returns better lower-bounds than LPR and allows *eclipse* to solve them to optimality. Our experiments demonstrate that the LPR and CP lower-bounding methods are essential contributors to the success of *eclipse*.

2. **Upper-bounding Techniques:** Branch-and-bound solvers backtrack when the lower-bound meets the upper-bound. Having the two powerful techniques for lower bounding above, it is still critical to have a good upper-bound in order to force backtracks as soon as possible. To achieve this, *eclipse* deploys a local search at each node of the search-tree to find a high-quality upper bound. Our local-search procedure, when properly initialized, is very fast and often very effective in finding the optimal solution for a given covering matrix. Of course, the local search doesn't provide any guarantee that the solution is optimal. However, for the purpose of finding a good upper-bound, proven optimality is not necessary. The amount of local search to do at each node also plays an important role in the overall performance of *eclipse*: too little search can overlook a good upper-bound but too much search can slow down *eclipse* significantly. *Eclipse* accommodates these conflicting demands by parameterizing the amount of search at each node. Combining this upper-bound search strategy with the search-tree exploration method below, *eclipse* is often able to find the global optimal solution early in the search and therefore, achieve the best possible upper-bound to maximize its search pruning.

3. **Heuristic Search-Tree Exploration Method:** In most state-of-the-art *Min-CostSat* solvers, the search-tree is explored in a depth-first fashion. This can be problematic if the search wanders into a "bad" branch that doesn't con-

tain an optimal solution and yet allows no effective search pruning. In *eclipse*, the search-tree is explored in a “best”-first fashion. Unexplored tree nodes are stored in a priority queue. The priority can be defined in various ways. In our study, the most effective one is to give the highest priority to the node currently having the *lowest* lower-bound. Intuitively speaking, the optimal solution is more likely to reside in the search-tree rooted at the node with the lowest lower-bound; therefore, it is desirable to explore that node first and use the local-search procedure to possibly improve the upper-bound. Our experiments show that this node selection strategy complements the local search well in practice and is very effective in guiding the search towards the optimal solution early.

4.3 Experimental Setup

The above features could not have been devised without doing experimental comparative study. We have presented the benchmark sets in Section 2.1.1. In order to increase the reliability of the performance evaluation of components of *eclipse*, we follow the principles first articulated in [58]:

1. We first do experiments on the *reference* benchmarks, either downloaded from the Internet or shared with us by their creators. When the results from the experiments are not conclusive, we proceed with the following.

2. For each reference benchmark, we randomly select 32 instances from its *equivalence P-class*. Each instance in the P-class is generated from the reference by randomly permuting the column variables, the rows, and the order of column variables in each row. The cost as well as the number of optimal solutions are preserved in the P-class.

3. For each solver (or its variations) and all instances in the equivalence class, we observe and record the mean and standard deviation for two random variables: runtime and number of nodes explored. Runtime is used to measure the overall

efficiency; the number of nodes explored is a machine/implementation independent combinatorial measure that can provide valuable insights not revealed by looking at the runtime measure alone.

Unless specified otherwise, all our experiments in this work are done on a Pentium IV@2.0Ghz with 1GB of RAM under Linux.

4.4 Performance Factors

Within *eclipse*, various "engines" have been built to optimize the search process for time and space. There are many factors that can influence the performance of *eclipse*. These include (1) lower-bounding techniques, (2) upper-bound techniques, (3) search-tree exploration strategy, (4) branching variable selection, (5) search pruning techniques, (6) reduction methods, and (7) data structures. The last four factors have been well studied in the literature and the best choice for implementation is easy to determine. However, for each of the first three factors, many variations exist and making the right choice requires the implementation of many alternatives and comparative studies of their performances. We list all seven performance factors in Figure 4.1. We now start our discussion with what we believe is the most important factor, the lower-bounding techniques.

4.4.1 Factor 1: Lower-Bounding Techniques

The lower-bounding technique is one of the most important factors in designing any branch-and-bound solver. Clearly, the best lower-bounding algorithm will allow the branch-and-bound to backtrack as early as possible and prevent wasting time on unfruitful search branches. *Eclipse* implements four lower-bounding techniques.

1. MIS1: This lower-bounding technique was introduced in Section 3.1.2.1 and it is based on the idea of a maximal independent set of rows. Let's assume all variables have unit cost. Since no two rows in the independent set can be

Factor 1: Lower bounding Techniques
- *a.* MIS1 (maximum independent set)
- *b.* MIS2 (maximum independent set improved)
- *c.* LPR (linear programming relaxation)
- *d.* CP (cutting planes)

Factor 2: Upper bounding Techniques
- *a.* No local search
- *b.* Local search only at the root
- *c.* Local search at each node
    - Initialization
        - *1.* Random
        - *2.* Solution from lower bounding
    - Amount of Search
        - *1.* Fixed
        - *2.* Parameterized
- *d.* Ask the Oracle

Factor 3: Search-Tree Exploration Strategy
- *a.* Breadth-First Search
- *b.* Depth-First Search
- *c.* Best-First Search

Factor 4: Branching Variable Selection Heuristics
Factor 5: Search Pruning
Factor 6: Reduction Techniques
Factor 7: Data Structures

Figure 4.1: The seven performance factors in *eclipse*.

covered by the same variable, at least one variable has to be chosen from each row to cover the rows in the maximum independent set. Since all the chosen variables are distinct, the lower-bound is the size of the maximal independent set.

2. MIS2: We have shown that MIS1 can underestimate the lower-bound in Section 3.1.2.1. MIS2 is the improved version we introduced in the same section.

3. LPR: This method is based on linear programming relaxation and we first intro-

duced it in Section 3.1.2.2. With this method, at each node of the search-tree, we generate a linear programming (LP) instance from the current covering matrix after relaxing the integral constraints. We then find the optimal objective value of the LP problem using *cplex* [20], a state-of-the-art *LP/IP* tool. The optimal objective value from the LP problem may be fractional but we can take its ceiling as the lower-bound for the current node.

4. CP: This one is similar to LPR but it utilizes *cutting planes*. We describe this method in more detail next.

Lower Bounding with Cutting Planes The *cutting planes* method is a well-known method for solving *IP*. With this method, the *IP* problem is first solved as a LP problem with the *simplex* method. If the optimal solution of the LP problem contains all integer values, then an optimal solution for the *IP* problem has been found; otherwise, the search for an optimal (integral) solution starts. The cutting planes method systematically generates new constraints, called Gomory cuts [67, 68, 69], and adds them to the optimal tableau of the original problem. These new constraints cut away part of the feasible solution space from the original problem, but they never exclude any integer solutions from the original solution space. Therefore, the new *IP* problem created is guaranteed to contain the same optimal solution as the original one. However, when the new *IP* problem is solved as an *LP* problem, its optimal objective value is no better than that of the first LP problem.

In the context of a *minimization* problem like *MinCostSat*, this means that the lower-bound we get after adding the Gomory cuts to the original *IP* may be *greater* than the one we get from simply doing LPR on the original *IP*.

Lower bounding with the cutting planes method goes through the following three steps:

Step 1. Solve the original LP problem with the simplex method. If the optimal solution contains fractional values, go to step 2; otherwise, return the optimal solution of the LP problem as the optimal solution of the *IP* problem.

Step 2. Generate the Gomory cuts and add the Gomory cuts to the optimal tableau found in step 1.

Step 3. Solve the new LP problem and return the ceiling of its optimal objective value as the lower-bound.

Clearly, CP is more expensive than LPR. The extra work comes from generating the Gomory cuts and solving the new LP problem. But CP sometimes provides much better lower-bounds than the other three methods. For example, at the root of the search-tree for the binate instance *apex4.a*, the four lower-bounding methods provide a lower-bound of 525 (MIS1), 525 (MIS2), 756 (LPR) and 773 (CP), respectively. Next, we compare the effectiveness of the four lower-bounding techniques in detail.

Comparisons of the Four Lower-Bounding Methods. In this study, we apply all four methods on the *reference instances.* As we will see, the performance trend for the four methods is very clear and convincing. For this reason, we *do not* generate the P-class for each benchmark. Table 4.1 shows the results on a group of unate covering benchmarks. The second column shows the size of the minimum cost cover. The next two columns show the lower-bounds obtained from MIS1 and MIS2. Comparison between the two columns shows that MIS2 can return slightly better lower-bounds than MIS1, e.g., for *exam.pi*, *max1024* and *steiner_a0081.*

The last two columns show the lower-bounds given by LPR and CP. Clearly, either method is able to obtain lower-bounds that are no worse (often better) than MIS1 and MIS2 on *all* the benchmarks. CP never returns worse lower-bounds than LPR and obtains slightly better lower-bounds than LPR on *exam.pi* and all the *Steiner* benchmarks.

A similar comparison study is shown in Table 4.2 for the binate benchmarks. The performance pattern is identical to that in Table 4.2. The quality of the lower-bounds is significant better for LPR and CP on most benchmarks. CP outperforms LPR on benchmarks such as *apex4.a* and *rot.b.* As we will see in Section 4.5, these two instances remain unsolvable (within the timeout limit) for *eclipse* with LPR lower bounding but are solved when using CP.

Table 4.1: Comparison of four lower bounding techniques on unate covering benchmarks.

benchmark	opt	MIS1	MIS2	LPR	CP
lin_rom	120	115	115	120	120
exam.pi	63	52	55	60	62
bench1.pi	121	116	116	120	121
prom2	278	264	264	278	278
prom2.pi	287	273	273	287	287
max1024	245	236	238	244	244
max1024.pi	259	250	252	258	258
ex5.pi	65	60	60	64	64
ex5	37	32	32	36	36
test4.pi	$\leq$ 101	56	56	80	80
steiner_a0009	5	3	3	3	4
steiner_a0015	9	5	5	5	6
steiner_a0027	18	9	9	9	11
steiner_a0045	30	15	15	15	17
steiner_a0081	61	25	26	27	30
m100_100_10_30	11	4	4	7	7
m100_100_10_15	10	4	4	8	8
m100_100_10_10	12	5	5	10	10
m200_100_10_30	11	3	4	8	8
m200_100_30_50	6	1	2	3	3

For each benchmark, the cost of the minimum cost cover is reported in the second column. The last four columns compare the two MIS-based lower bounding techniques with linear programming relaxation (LPR) and cutting planes (CP) based techniques. LPR and CP never performs worse than MIS1 and MIS2. CP can outperform LPR for benchmarks such as *exam.pi* and *steiner_a0081*. In most cases, a gradual increase in the quality of the lower-bounds can be observed across the four methods from left to right.

Table 4.2: Comparison of four lower-bounding techniques on binate covering benchmarks.

benchmark	opt	MIS1	MIS2	LPR	CP
count.b	24	17	17	24	24
clip.b	15	10	10	14	14
9sym.b	5	4	4	5	5
jac3	15	12	12	15	15
f51m.b	18	14	15	16	17
sao2.b	25	24	25	25	25
5xp1.b	12	9	9	11	11
apex4.a	776	525	525	756	773
rot.b	115	95	98	111	114
alu4.b	50	38	39	47	47
e64.b	≤ 48	32	32	37	40
c432_F37gat@1	9	1	1	3	3
misex3_Fb@1	8	1	1	2	2
c1908_F469@0	11	1	1	4	4
c6288_F69gat@1	6	1	1	2	2
c3540_F20@1	6	1	1	3	3

The format of this table is identical to Table 4.1. Again, LPR and CP never perform worse than MIS1 and MIS2. In most cases, a gradual increase in the quality of the lower-bounds can be observed across the four methods from left to right. In particular, on *count.b*, *apex4.a*, *rot.b*, and *e64.b*, the differences are significant. Notice that for the ATPG group, the lower-bounds at the root are relatively far from the optimum, especially for MIS1 and MIS2. This is because all but few rows contain -1's and these rows cannot be added to the independent set.

LPR and CP are in general much more expensive than MIS1 and MIS2 for two reasons: (1) the overhead of creating the LP instance and set up a procedure call to *cplex*; (2) the dual simplex method used by *cplex* to solve the LP takes more time than an MIS heuristic. However, since a small increase in the lower-bound value can potentially cut off search branches of exponential size, an improvement is still possible even with this big overhead. Indeed, our experiments in Section 4.5 show that *eclipse* with LPR or CP can outperform state-of-the-art *MinCostSat* solvers on many benchmarks. From this point on, we refer to the version of *eclipse* using LPR as *eclipse-lpr* and the version using CP as *eclipse-cp*. Table 4.1 and table 4.2 clearly demonstrate that LPR/CP dominate MIS1/MIS2; therefore, we will not present any results of *eclipse* with MIS1/MIS2.

Implementation Details. For the two MIS-based lower-bounding methods, *eclipse* implements the algorithms we presented in Section 3.1.2. *Eclipse-lpr* uses the idea presented in Section 1.2.1 to generate an *IP* instance from the covering matrix at each node. It then relaxes the integral constraints and feeds the corresponding LP problem to *cplex* via a system call. Upon *cplex*'s completion, *eclipse-lpr* extracts the optimal objective value v and uses $\lceil v \rceil$ as the lower-bound of the current node.

Eclipse-cp generates the *IP* instance the same way as *eclipse-lpr*; however, instead of feeding the relaxed LP instance, *eclipse-cp* feeds the *IP* instance itself to *cplex*. *Cplex* adds the Gomory cuts to the *IP* problem and starts solving the new *IP* instance. However, before it solves the new *IP* instance, *cplex* will calculate the lower-bound of the objective function, which is the optimal objective value of the LP instance corresponding to the new *IP* instance. This is exactly the value *eclipse-cp* needs to extract as the lower-bound. The system call to *cplex* is terminated once it finds the first integer solution to the new *IP* problem. It is worth emphasizing that the Gomory cuts are generated internally by *cplex*, not by *eclipse-cp*.

4.4.2 Factor 2: Upper-bounding Techniques

Branch-and-bound search is a tale of the two bounds: lower-bound and upper-bound. The solver backtracks whenever they meet and the branch of the search-tree below the current node is pruned. We have spent a lot of effort doing lower bounding, but it is equally important to find an effective upper-bounding strategy. By definition, the upper-bound is the cost of the best solution found and the best possible upper-bound is the cost of the optimal solution. Clearly, the goal of any upper-bounding strategy is to find an optimal solution as soon as possible. We list four possible ways of finding high quality upper-bounds, each with increasing level of sophistication:

1. **No local search:** This the most naive method in which there is no explicit search for a good upper-bound. The upper-bound is initialized to be ∞ (or a sufficiently large number). The first time the upper-bound is updated is when the first feasible solution is found in the branch-and-bound search-tree. The upper-bound is updated only when a feasible solution with better quality is found. The performance of a solver with this approach depends highly on where the optimal solutions may reside in the search-tree. In the worst case, the optimal solutions may be found only at the end of the search and therefore cannot help the search pruning effectively.

2. **Local search at the root:** This is a method used in many branch-and-bound solvers. These solvers have two phases. The purpose of the first phase is to find a good upper-bound at the root of the search-tree. After that, no more explicit search for a good upper-bound is carried out. In general, a branch-and-bound solver with this method is more effective than one with the first method because a good upper-bound can often be found quickly in very little time. However, since the local search may get stuck in local minima, it may never find the best possible upper-bound. Shown in Section 5.6, this situation does happen in practice.

3. **Local search at each node:** This method does local search not only at the root but also at each node of the search-tree. The local search at these nodes tend to be more diversified than the one at the root. This is because at each node below the root, a portion of the variables have been assigned. The local search at each node fixes the same set of assignments and only the unassigned variables are allowed to be changed during the local search. It is easy to see that the set of fixed variables and their assignments is unique for each distinct node. By doing a local search at each node, we are essentially forcing the local search to cover all regions of the search space (even though these regions may

overlap). Therefore, the search is less likely to get stuck in a local minima and overlook a high quality upper-bound.

4. **Ask the Oracle:** This is the ideal situation where the cost of the optimal solution is given a priori. With this cost as the upper-bound, the search should achieve the best performance (everything else being equal). We use this method as a reference for evaluating the previous three methods.

At each node, besides calculating the lower-bound, *eclipse* uses the third method above to search for a high quality upper-bound. The local-search procedure is based on the algorithms we will present in Chapter 5. As we will see, the local-search procedure is very fast and often very effective in finding an optimal solution. Of course, the local search doesn't provide any guarantee that the solution it finds is optimal. However, for the purpose of finding a good upper-bound, this is not necessary.

The local-search procedure starts by initializing all the variables to either 0 or 1. At each step of the search, a variable is chosen according to some heuristics (discussed in Chapter 5) and its value is flipped. The search terminates after a predefined number of steps. Various strategies can be used to initialize the variables, among them the easiest is to randomly initialize the variables. Next, we present a novel initialization strategy implemented in *eclipse*.

Local-Search Initialization. Both *eclipse-lpr* and *eclipse-cp* take advantage of the solution found in the their lower-bounding procedures. In *eclipse-lpr*, the variables are initialized with the optimal LP solution from the LPR procedure, with fractional values in $(0, 0.5]$ rounding down to 0 and fractional values in $(0.5, 1)$ rounding up to 1. The rationale behind this initialization strategy is that we project the optimal integer solution for the *IP* problem to be "close" to that for the corresponding LP problem; therefore, we keep the variables with integer values intact and round the non-integer variables to their nearest integer values.

In *eclipse-cp*, the solution it extracts from *cplex* during the lower-bounding procedure is already a feasible solution to the *IP* problem. We use this solution to initialize the local search.

We choose six representative benchmarks from the unate and binate covering benchmark set in order to compare the two initialization strategies: random initialization and heuristic initialization using the solutions from lower bounding. The same set of six benchmarks will be used throughout the study of the second and the third performance factor. For each reference benchmark, we run two variations of *eclipse-cp* on the reference and 32 instances from its P-class (with a timeout of 300 seconds on each instance). We then report the mean and standard deviation of the runtime and the number of nodes explored for each initialization strategy. The results are shown in Table 4.3.

Table 4.3: Comparison of two local-search initialization strategies in *eclipse*: random initialization vs. heuristic initialization.

	random init		heuristic init	
benchmark	time	nodes	time	nodes
exam.pi	20.8/50.0	117/474	5.4/5.0	12/22
bench1.pi	300/0*	597/5	4.7/1.4	7/4
max1024	143.9/86.6	303/333	27.5/18.7	27/20
ex5	16.3/5.3	4/1	15.1/7.7	5/3
5xp1.b	3.0/1.5	10/7	3.1/1.5	10/7
jac3	8.7/1.5	3/1	8.1/1.0	3/0

* all 33 instances time out at 300 seconds.

This table compares two initialization strategies for the local search: random initialization vs. heuristic initialization with the solution from the lower-bounding procedure. The mean and standard deviation for runtime and nodes are reported. The heuristic initialization method performs significantly better on *exam.pi*, *bench1.pi* and *max1024*. In both approaches, *exam.pi* and *max1024* each have large variances in time and nodes.

For three of the six benchmarks, the initialization scheme makes a big difference. For example, on *bench.1.pi*, all instances under random initialization time out at 300 seconds; under heuristic initialization, the mean runtime for *eclipse-cp* is only 4.7 seconds with a standard deviation of 1.4. For *max1024*, *eclipse-cp* with random initialization is on average about five times slower than *eclipse-cp* with heuristic initialization. The former explores 303 nodes on average whereas the latter only explores 27. A similar situation can also be observed on *exam.pi*. For the other benchmarks, the initialization scheme doesn't make a noticeable difference. This is primarily because a global optimal solution is easy to find early and cannot be improved by the later local search, regardless of which initialization method is used.

This experiment shows that the heuristic initialization method can make the local search much more effective in finding good quality upper-bounds than the random initialization method. The cost of the heuristic initialization method is minimal because the solutions we use are byproducts of the lower-bounding procedure.

After the initialization, the local search runs until the search termination criterion is met. This termination criterion determines the amount of local search to do at each node.

4.4.2.1 Search Termination Criteria

How much local search to do at each node can have a big impact on the performance of *eclipse*:

1. Too much search will increase the processing time of each node and slow down *eclipse* significantly.
2. Too little search may overlook the chance of improving the global upper-bound at a node.

In addition, the amount of local search at each node shouldn't be the same. Since the search space below each node decreases exponentially as the depth of the search tree grows, the amount of local search at each node along a path from the root should decrease accordingly. To address these issues in *eclipse*, we define the number of search steps to do at a given node to be:

$$(1/2)^d * c_f * S \tag{4.1}$$

where d is the depth of this node, c_f is a constant (set to 5 in our experiments), and S is the size (the number of non-zero entries) of the covering matrix at the node. Under such a strategy, the maximum amount of search at any given node is done at the root and the amount of search decreases exponentially along any path of the search-tree.

4.4.2.2 Comparisons of the Four Upper-Bounding Methods

In order to present the essential role the local search plays in *eclipse*, we compare the four different upper-bounding methods in Table 4.4 on the same six representative benchmarks we studied in Table 4.3. We make the following observations:

1. From left to right, the decreasing trend for both the runtime and the number of nodes is very clear. With no explicit upper bounding (the first method), *eclipse* times out on three benchmarks. Simply by doing a local search at the root, the performance of *eclipse* is already greatly enhanced and the timeout situation is no longer observed. By doing parameterized amount of local search at each node with heuristic initialization, *eclipse* is further improved, especially on *bench1.pi* and *max1024*.

2. The difference between the third method (implemented in *eclipse*) and the last method with the oracle are not dramatic. This indicates that the local search is effective in finding an optimal solution *early*.

Shown in Section 4.5, the amount of time *eclipse* spent on doing local search at each node is not overwhelming in most cases.

4.4.3 Factor 3: Heuristic Search-tree Exploration

There are many ways to explore the search-tree and the node selection strategy determines the order the search-tree nodes are visited. Common tree traversal strategies include:

Table 4.4: Comparison of four different upper-bounding methods with *eclipse*.

	none		root		each node		oracle	
benchmark	time	nodes	time	nodes	time	nodes	time	nodes
exam.pi	300/0*	2355/89	7.74/5.53	15/24	5.4/5.0	12/22	4.4/3.9	10/19
bench1.pi	300/0*	609/15	211.0/136.2	447/293	4.7/1.4	7/4	3.8/1.8	3/2
max1024	251.4/15.6	263/45	148.4/80.7	148/81	27.5/18.7	27/20	17.8/1.7	11/1
ex5	300/0*	97/3	25.3/33.4	6/11	15.1/7.7	5/3	3.7/0.2	1/0
5xp1.b	26.6/3.5	594/92	2.84/1.26	9/5	3.1/1.5	10/7	2.5/1.8	7/6
jac3	10.8/1.6	130/70	8.3/0.6	3/0	8.1/1.0	3/0	2.4/0.3	1/0

* times out at 300 seconds.

From left to right, this table presents four upper-bounding strategies with respect to doing local search: (1) no search at all, (2) only search at the root, (3) search at all nodes and (4) get the best possible upper-bound from an oracle. The mean and standard deviation for runtime and nodes are reported. Clearly, the third method performs much better than the first two methods in both runtime and the number of nodes explored. In addition, the performance of the third method is not much worse than the fourth method (the ideal situation). This indicates that the local search is effective in finding the best possible upper-bound *early*.

1. Breadth-First Search: the node with the least depth is explored first. Breadth-first search has the advantage of being able to search uniformly into the branches of the search-tree but as the depths of the search increases, there are exponential number of nodes to store in the memory. For this reason, we don't consider breadth-first search any further in our study.

2. Depth-First Search: the node with the greatest depth is explored first. The advantage of depth-first search is that the number of nodes stored in memory is proportional to the depth of the search-tree, which is often a small fraction of the total number of variables. The disadvantage of depth-first search is that once the search goes into a "bad" branch, it may take a long time for the search to backtrack to a more fruitful branch.

3. Best-First Search: the node with the best heuristic measure is explored first. Both the depth-first search and the breadth-first search are special cases of the best-first search. The main advantage of the best-first search is that it may, on average, find an optimal solution faster than the other two search strategies.

We deploy a type of best-first search in *eclipse*. All the unexplored nodes are maintained on a priority queue. The highest priority is given to the node currently having the *lowest* lower-bound. Intuitively speaking, an optimal solution is more likely to reside in the search-tree below that node and therefore, it is desirable to explore that node first. We compare two search-tree exploration strategies, depth-first and best-first, in Table 4.5, one is depth-first search and the other one is best-first search. On the first three benchmarks, best-first search explores less nodes and has shorter runtime on average. For example, on *max1024*, *eclipse* with depth-first search explores 68 nodes on average but it only explores 27 nodes on average with best-first search. On the last three benchmarks, the two strategies have practically the same performance.

Table 4.5: Comparison of two search-tree exploration strategies: depth-first search vs. best-first search.

	depth FS		best FS	
benchmark	time	nodes	time	nodes
exam.pi	7.0/5.4	15/23	5.4/5.0	12/22
bench1.pi	7.91/12.0	12/21	4.7/1.4	7/4
max1024	74.1/27.0	68/43	27.5/18.7	27/20
ex5	16.2/6.7	4/2	15.1/7.7	5/3
5xp1.b	3.0/1.4	10/7	3.1/1.5	10/7
jac3	8.6/1.5	3/1	8.1/1.0	3/0

This table compares two search-tree exploration strategies: depth-first search and best-first search. The mean and standard deviation for runtime and nodes are reported. Best-first search explores the node with the least lower-bound first. Best-first search outperforms depth-first search on the first tree benchmarks. For the last three benchmarks, the two approaches are almost identical.

4.4.4 Factor 4: Branching Variable Selection

After applying the reduction techniques at each node of the search-tree, we obtain the cyclic core of the current covering matrix. To explore the search-tree further, a branching variable has to be chosen to generate two children of the current node. In *eclipse-lpr*, we complement the branching variable selection heuristic [28] (we also presented it in Section 3.1.4) with the linear programming solution we obtained from the LPR lower bounding. *Eclipse-lpr* extracts the optimal *LP* solution for the variables. For each variable x_i, it calculates the distance between the value assigned to x_i and its nearest integer:

$$d_i = min(\lceil v_i \rceil - v_i, v_i - \lfloor v_i \rfloor) \tag{4.2}$$

Let $IR(x_i)$ be the rows containing either x_i or $\overline{x}_i$. For the branching variable, we choose the x_i that maximizes the expression

$$c \cdot d_i + \frac{1}{Cost(x_i)} \sum_{y \in IR(x_i)} \frac{weight(y)}{length(y)}. \tag{4.3}$$

where $weight(y)$ is the weight of the variable in y with the smallest weight and $length(y)$ is number of non-zero literals in y. The purpose of the constant c in the function above is to give more weights to d_i. In all our experiments, we set c to 5 because it gives us the best overall results.

In *eclipse-cp*, since *cplex* is given an *IP* instance, we don't have the capability of extracting the optimal solution for the LP instance (even though we can extract the optimal objective value of the LP instance to serve as the lower-bound). Instead, the solution *eclipse-cp* extracts is simply the first integral solution *cplex* finds. Therefore, the distance between any variable's assignment and its nearest integer is 0 and consequently, the first term of the cost function in (4.3) is always 0. Then the variable selection heuristic becomes identical to the one we introduced in Section 3.1.4. In *eclipse*, the time spent on branching variable selection is negligible.

4.4.5 Factor 5: Search Pruning

Besides the usual search pruning when $Cost_{part} + Cost_{lower} \geq UB$, *eclipse* also uses the C_l*-Lower-Bound* pruning technique we introduced in Section 3.1.3. Recall that the C_l*-Lower-Bound* pruning technique states: if $Cost_{path} + Cost.l_{lower} \geq UB$, then both C_l (the branch after setting the branching variable to 1) and C_r (the branch after setting the variable to 0) can be pruned. The applicability of this pruning rule can be checked at no extra cost since $Cost_{path}$ is available when the root of C_l is generated and $Cost.l_{lower}$ is just the result from the lower-bounding procedure, a necessary step *eclipse* goes through regardless what search pruning technique it uses.

4.4.6 Factor 6: Reduction Techniques:

At each node, the current covering matrix is reduced to its cyclic core by using row dominance, column dominance and essentiality. These reduction techniques are implemented in *eclipse* following the algorithms described in Section 3.1.1. Runtime analysis in Section 4.5 shows that during an execution of *eclipse*, the time spent on reduction is often a small fraction of the total runtime.

4.4.7 Factor 7: Data Structures

Most of the standard *MinCostSat* benchmarks are very sparse: less than 10% of entries (some are much less) in the covering matrix are occupied by 1 or -1 (see Tables 2.1–2.5). To represent such matrices efficiently, we use a *sparse matrix* data structure to store the covering matrix:

1. Each row is represented by a doubly-linked list of the non-zero entries in the row.
2. Each column is represented by a doubly-linked list of the non-zero entries in the column.
3. Each matrix entry has pointers to the row and columns it resides in and knows, implicitly or explicitly, its positions in the appropriate lists.

The sparse matrix data structure compactly represents the covering matrix. It also allows efficient implementation of the reduction algorithms by allowing constant time removal of entries.

We have finished discussing the seven performance factors in *eclipse*. To summarize, *eclipse* implements two advanced lower-bounding techniques: LPR and CP. For upper bounding, *eclipse* does local search at each node of the search tree to find high quality upper-bounds. The local search is initialized with the solutions from the lower-bounding procedure and the amount of local search at each node is parameterized by its depth in the tree. *Eclipse* uses the branching heuristic in (4.3). It explores the search-tree in a best-first fashion in order to find the best possible upper bound early. For search pruning, *eclipse* uses C_l-*Lower-Bound* pruning technique. *Eclipse* is built on top of the sparse matrix data structure and it implements efficient reduction algorithms.

4.5 Experimental Results

Even though *eclipse* is capable of solve any *MinCostSat* problems, it has been designed to specialize in the covering problems. Therefore, in this experimental study, we concentrate on the native covering benchmarks first introduced in Table 2.1 and Table 2.2. We start by introducing the solvers involved in our study.

4.5.1 The Solvers

The six solvers we consider in this experimental study include:

1. *scherzo* [28]: This solver introduced many of the state-of-the-art techniques for solving the covering problems that include new heuristics for MIS-based lower bounding, branching variable selection strategies, and two search pruning rules.

2. *aura*II [30]: This solver is built on top of *scherzo*. By doing "negative thinking", *aura*II is able to outperform *scherzo* on some unate instances. However, *aura*II has not been implemented to solve the binate covering problems.

3. *bsolo* [41]: This is the latest *MinCostSat* solver in the literature. It is a *scherzo*-like solver but it also incorporates *Sat* techniques such as non-chronological backtracking. It is not efficient on covering problems but performs well on a set of non-covering benchmarks from `ATPG` [41].

4. *cplex* [20]: This is a commercial state-of-the-art *LP*/*IP* solver. Since *MinCostSat* can be formulated as *IP*, *cplex* can be used to solve any *MinCostSat* problem. It is quite competitive with solvers that specialize in solving covering problems. *Eclipse* uses *cplex* to do LPR and CP lower bounding.

5. *eclipse-lpr*: This is our branch-and-bound *MinCostSat* solver that implements all the main features we have discussed in this chapter. For lower bounding, *eclipse-lpr* uses linear programming relaxation.

6. *eclipse-cp*: This is the same as *eclipse-lpr* except that for when doing lower bounding, the cutting-planes method is used.

We chose not to include the *MinCostSat* solver *mindp* [40] in this comparison study because it is not as competitive as the other solvers: it times out at 600 seconds on *lin_rom* and *count.b*, the smallest industrial *UCP* and *BCP* benchmarks we consider in this work.

4.5.2 Unate Covering Comparisons

We start by comparing *scherzo*, *aura*II, *cplex*, *eclipse-lpr* and *eclipse-cp* on the *reference benchmarks* from two-level logic minimization. We don't include *bsolo* because it is not competitive with the other five solvers. We then study the performances of *cplex*, *eclipse-lpr* and *eclipse-cp* more carefully by conducting statistically significant experiments on the P-classes of the reference benchmarks. For the set covering and randomly generated benchmarks, we follow the same setup as above by first experimenting with the reference benchmarks only, and then with the P-class instances.

4.5.2.1 Comparison on logic minimization benchmarks

Comparison of five solvers on the reference instances. In Table 4.6, we compare five solvers' performance on the two-level logic minimization reference benchmarks that were first introduced in Table 2.1. We observe dramatic performance differences between the first two solvers and the last three solvers. *Scherzo* and *aura*II perform poorly compared to *cplex*, *eclipse-lpr* and *eclipse-cp*. For example, on *bench1.pi*, *scherzo* takes 1349.8 seconds; *aura*II times out at 3600 seconds; *cplex*, *eclipse-lpr* and *eclipse-cp* take only 4.4, 3.8 and 2.2 seconds, respectively. The differences between the number of nodes explored are equally striking: *scherzo* explores over two million nodes before it solves the instance whereas *cplex*, *eclipse-lpr* and *eclipse-cp* explore fewer than 50 nodes each.

Recall that the lower-bounding techniques used in *scherzo* and *aura*II are based on MIS. We have shown in Section 4.4.1 that MIS-based lower bounding is inferior to both LPR and CP. We believe that the lack of ability to find good lower-bounds is the key reason why *scherzo* and *aura*II don't perform as well as the other three

Table 4.6: Comparison of *scherzo*, *aura*II, *cplex*, *eclipse-lpr* and *eclipse-cp* on unate covering benchmarks (reference instances only) from logic minimization.

benchmark	measure	*scherzo*	*aura*II	*cplex*	*eclipse-lpr*	*eclipse*-cp
lin_rom	time	0.6	1.6	0.2	0.3	0.3
	nodes	236	54	1	1	1
exam.pi	time	3600*	3600*	3.6	162.2	22.4
	nodes	–	–	30	2768	109
bench1.pi	time	1349.8	3600*	4.4	3.8	2.2
	nodes	2001438	–	10	32	4
prom2	time	376.9	3600*	5.2	2.2	8.0
	nodes	25865	–	1	1	1
prom2.pi	time	650.5	3600*	6.0	2.4	6.1
	nodes	23585	–	1	1	1
max1024	time	224.3	3590.9	18.6	39.3	15.9
	nodes	533635	1616993	118	224	11
max1024.pi	time	1749.3	3600*	21.1	53.0	18.0
	nodes	414030	–	119	282	16
ex5.pi	time	3600*	488.8	25.6	7.7	9.0
	nodes	–	223542	104	9	3
ex5	time	3157.9	400.4	116.5	5.9	43.2
	nodes	615187	194663	1149	5	18
test4.pi	time	3600*	3600*	3600*	3600*	3600*
	nodes	–	–	–	–	–

* times out at 3600 seconds.

This table compares five covering solvers on the set of two-level logic minimization benchmarks. For each reference benchmark, we report the runtime and the number of nodes explored by each solver with a one-hour timeout. Clearly, traditional covering solvers *scherzo* and *aura*II are not competitive with *cplex*, *eclipse-lpr* and *eclipse-cp* on either runtime or nodes. The performance relationships among *cplex*, *eclipse-lpr* and *eclipse-cp* are less obvious. We compare these three solvers more thoroughly in Table 4.7.

solvers.

The performance relationships among *cplex*, *eclipse-lpr* and *eclipse-cp* are less obvious. In order to evaluate their performances more reliably, we conduct a set of experiments with the three solvers on P-classes of size 32 on the same set of logic minimization benchmarks. We report the results in Table 4.7.

Comparison between eclipse-lpr and eclipse-cp on P-classes. We first compare the two variants of *eclipse*. In terms of runtime, *eclipse-lpr* generally runs much faster on the easier instances such as *prom2*, *prom2.pi*, and *bench1*. However, on the more difficult instances such as *exam.pi*, *max1024*, and *max1024.pi*, *eclipse-cp* is more efficient. On *exam.pi* in particular, *eclipse-cp* has a mean runtime of 5.4 seconds whereas the mean for *eclipse-lpr* is 160.6 seconds. In terms of the number of nodes explored, the pattern is much more consistent. *Eclipse-cp always* explores fewer nodes than *eclipse-lpr* on average. However, fewer nodes doesn't always imply less runtime, e.g. on *bench1*. This indicates that *eclipse-cp* spends more time processing each node than *eclipse-lpr* (mainly doing lower bounding) .

Comparison between cplex and eclipse-cp on P-classes. In terms of runtime, *cplex* and *eclipse-cp* are very similar on all but three benchmarks: *exam.pi*, *ex5* and *ex5.pi*. On these three benchmarks, *eclipse-cp* outperforms *cplex* by a big margin. For example, on *ex5*, *eclipse-cp* has a mean runtime of 15.1 seconds and *cplex* runs more than four times as slow at 66.2 seconds. In terms of the number of nodes explored, *eclipse-cp* always explores fewer nodes on average. On instances such as *exam.pi* and *ex5*, the nodes explored by *eclipse-cp* are two orders of magnitude smaller than that of *cplex*. Again, the savings on the number of nodes explored don't always transfer to the same amount of savings on runtime. For example, on *max1024*, the average number of nodes explored by *cplex* (298) is 10 times as much as that for *eclipse-cp* (27); however, *cplex* still runs slightly faster than *eclipse-cp*.

Overall comparison of cplex, eclipse-lpr and eclipse-cp on P-classes. On all benchmarks, *eclipse-cp* explores the least number of nodes on average. *Eclipse-cp* also has the best overall runtime performance among three solvers. It is interesting to observe that all three solvers explore one node on *lin_rom*, *prom2* and *prom2.pi*.

Table 4.7: Comparison of *cplex*, *eclipse-lpr* and *eclipse-cp* on P-classes of size 32 for the unate covering benchmarks from two-level logic minimization.

benchmark	measure	*cplex*	*eclipse-lpr*	*eclipse-cp*
lin_rom	time	0.19/0.02	0.34/0.02	0.19/0.02
	nodes	1/0	1/0	1/0
exam.pi	time	26.1/44.7	160.6/31.3	5.4/5.0
	nodes	1917/4248	2523/448	12/22
bench1.pi	time	7.3/2.3	2.4/1.5	4.7/1.4
	nodes	68/50	17/15	7/4
prom2	time	5.1/0.8	2.8/0.13	8.0/0.7
	nodes	1/0	1/0	1/0
prom2.pi	time	5.2/0.7	2.5/0.2	7.8/0.8
	nodes	1/0	1/0	1/0
max1024	time	26.6/21.3	52.7/20.3	27.5/18.7
	nodes	298/578	254/69	27/20
max1024.pi	time	20.3/7.9	58.9/18.1	23.0/12.3
	nodes	148/217	257/71	22/13
ex5.pi	time	65.7/37.6	12.8/14.1	16.7/15.0
	nodes	524/419	18/34	5/4
ex5	time	66.2/28.3	13.9/15.7	15.1/7.7
	nodes	505/310	22/40	5/3
test4.pi	time	3600*	3600*	3600*
	nodes	–	–	–

* times out at 3600 seconds.

This table compares *cplex*, *eclipse-lpr* and *eclipse-cp* on the P-classes of size 32 for the two-level logic minimization benchmarks. For each solver on each benchmark, we report the mean and standard deviation of the runtime and the number of nodes explored. On the easier instances, *eclipse-cp* behaves similar to *cplex*. But on harder instances such as *exam.pi*, *ex5* and *ex5.pi*, *eclipse-cp* performs significantly better. *Eclipse-lpr* runs much slower than the other two solvers on *exam.pi*, *max1024* and *max1024.pi*. All solvers time out on *test4.pi*.

This means that there exists an all-integer optimal solution to the *LP* relaxation. Therefore, there is no need to explore the search-tree beyond the root. The benchmark *test4.pi* remains unsolvable by all three solvers.

4.5.2.2 Comparison on Set Covering and Random Benchmarks

Following the same setup for studying the logic minimization benchmarks, we first run all solvers on the reference benchmarks and report the results in Table 4.8. We then run them on P-classes of the reference benchmarks.

Comparison of Five Solvers on the Reference Instances. For the set covering group, the runtime information for *scherzo* and *aura*II are almost identical. Both solvers are much faster than *cplex*, *eclipse-lpr* and *eclipse-cp*. For example, on *steiner_a0045*, *scherzo* and *aura*II run for 6.8 and 8.6 seconds, respectively; *cplex* runs for 38.9 seconds and both *eclipse-lpr* and *eclipse-cp* run for more than 200 seconds. However, the number of nodes explored tells a different story. On *steiner_a0045*, *scherzo* explores 42334 nodes whereas *eclipse-cp* only explores 5818 nodes. This discrepancy between the runtime and the number of nodes is mainly due to the difference between the overhead of the lower-bounding procedures. In this group, *eclipse-lpr* and *eclipse-cp* are similar in runtime but are both slower than *cplex*. For the randomly generated group, *aura*II is the most dominant solver in speed. Like before, *eclipse* explores the least number of nodes.

Comparison of *cplex*, *eclipse-lpr* and *eclipse-cp* on P-classes. For the set covering group, *cplex* is the most efficient among the three solvers. It is worth pointing out that the standard deviation of runtime and nodes is much smaller than their mean for all three solvers (except for *cplex* on *steiner_a0045*). This indicates that the permutations applied on the P-class instances have limited impact on the landscape of the search space. Given the fact that the set covering benchmarks are “handmade” and contain numerous symmetries, this is expected.

For the randomly generated benchmarks, *cplex* and *eclipse-lpr* are faster than *eclipse-cp* but no clear winner can be declared. The tradeoff between quality and

Table 4.8: Comparison of *scherzo*, *aura*II, *cplex*, *eclipse-lpr* and *eclipse-cp* on unate covering benchmarks (reference instances only) from set covering and randomly generated set.

benchmark	measure	*scherzo*	*aura*II	*cplex*	*eclipse-lpr*	*eclipse-cp*
steiner_a0009	time	0.01	0.01	0.01	0.04	0.11
	nodes	14	6	5	5	12
steiner_a0015	time	0.01	0.01	0.03	0.36	0.32
	nodes	60	86	91	39	29
steiner_a0027	time	0.09	0.14	0.79	11.87	13.29
	nodes	2054	4145	3707	1287	1040
steiner_a0045	time	6.81	8.58	38.94	268.17	214.11
	nodes	42334	93815	72135	15852	5818
steiner_a0081	time	3600	3600	3600	3600	3600
	nodes	–	–	–	–	–
m100_100_10_15	time	2.57	1.01	0.93	0.96	2.28
	nodes	10335	11067	116	33	19
m100_100_10_30	time	1.70	0.33	1.78	0.64	0.67
	nodes	4618	2724	510	25	5
m100_100_10_10	time	15.29	8.99	4.19	4.59	8.39
	nodes	95086	127919	1040	312	81
m200_100_10_30	time	245.5	55.9	82.06	106.01	258.76
	nodes	564302	371160	25409	2847	1592
m200_100_30_50	time	71.4	14.1	129.07	65.29	200.68
	nodes	123621	51616	30529	1511	1178

This table compares five covering solvers on the set covering and randomly generated benchmarks. For each reference benchmark, we report the runtime and the number of nodes explored by each solver with a one-hour timeout. On the set covering benchmarks, *scherzo* and *aura*II behave similarly and they both outperform the other three solvers. On the randomly generated benchmarks, *aura*II is the clear winner. Even though *eclipse-lpr* and *eclipse-cp* explores significant fewer nodes, they are still much slower than *scherzo* and *aura*II on most benchmarks. All solvers time out on *steiner_a0081*.

Table 4.9: Comparison of *cplex*, *eclipse-lpr* and *eclipse-cp* on P-classes of size 32 for the unate covering benchmarks from set covering and randomly generated set.

benchmark	measure	*cplex*	*eclipse-lpr*	*eclipse-cp*
steiner_a0009	time	0.01/0	0.04/0.01	0.11/0.03
	nodes	2.2/2.6	5/0	12/0
steiner_a0015	time	0.03/0	0.32/0.06	0.37/0.16
	nodes	82/5	37/2	31/14
steiner_a0027	time	0.81/0.04	11.88/0.54	14.1/1.24
	nodes	3635/188	1278/10	1060/101
steiner_a0045	time	39.3/49.5	307.1/26.7	229.8/6.2
	nodes	74547/9390	20648/4228	5998/141
steiner_a0081	time	3600*	3600*	3600*
	nodes	–	–	–
m100_100_10_15	time	0.9/0.1	1.07/0.57	2.45/0.41
	nodes	116/1	50/44	22/4
m100_100_10_30	time	1.0/0.4	0.53/0.23	0.65/0.01
	nodes	222/163	19/18	5/0
m100_100_10_10	time	6.7/3.1	10.75/10.25	17.1/19.7
	nodes	2403/1595	665/643	200/270
m200_100_10_30	time	78.1/3.4	107.7/1.6	258.9/1.06
	nodes	22603/1398	2846/2	1592/0
m200_100_30_50	time	123.0/4.7	66.0/1.2	202.0/0.77
	nodes	31051/293	1511/0	1180/2

This table compares *cplex* with *eclipse-lpr* and *eclipse-cp* on P-classes of size 32 for the set covering and randomly generated benchmarks. The mean and standard deviation for runtime and nodes are reported. On the set covering benchmarks, *cplex* is much more efficient than *eclipse-lpr* and *eclipse-cp* even though the latter two explore fewer nodes on average. All three solvers show very small variability in both runtime and nodes. On the randomly generated benchmarks, *eclipse-cp* generally runs slower than the other two solvers but no clear dominance relationship exists between *cplex* and *eclipse-lpr*. Again, *eclipse-cp* explores the least number of nodes on average.

efficiency of the lower-bounding techniques manifests itself again here: for the number of nodes explored, *eclipse-cp* consistently has the least mean value for all benchmarks; however, *eclipse-cp* still runs slower than the other two solvers.

4.5.3 Binate Covering Comparisons

We first consider five solvers' performance on the reference instances from logic synthesis benchmarks. The covering solver *aura*II can only solve unate covering problems so we replace it with *bsolo* in the binate covering comparisons.

Comparison of five solvers on the reference instances. The results are reported in Table 4.10. It is clear that *bsolo* is not competitive so we exclude it from the following discussions. On the easy benchmarks, *scherzo* remains competitive with the other three solvers. For the more difficult instances, *apex4.a* and *rot.b*, both *cplex* and *eclipse-cp* outperform *scherzo*. *Eclipse-lpr* times out on these two instances mainly because its lower-bounding procedure doesn't provide large enough lower-bounds to prune the search early. Recall that in Table 4.2, *eclipse-lpr* returns a lower-bound of 756 for *apex4.a*, but *eclipse-cp* returns a lower-bound of 773 (only 4 away from the global optimum). For *rot.b*, *eclipse-lpr* returns a lower-bound of 111 whereas *eclipse-cp* returns 114 (only 1 away from the global optimum). For the number of nodes, *eclipse-cp* explores the least amount on all benchmark except for *rot.b*. We observe that when *eclipse-lpr* times out on *apex4.a* and *rot.b*, thousands of unexplored nodes still reside in its priority queue. All solvers time out on the two most difficult instances: *alu4.b* and *e64.b*.

Comparisons of *cplex*, *eclipse-lpr*, and *eclipse-cp* on P-classes. In Table 4.11, we observe that *cplex* is the fastest solver in most cases, especially for the two harder instances *apex4.b* and *rot.b*. For these two benchmarks, *eclipse-lpr* times out mainly because its relatively weak lower-bounding technique. Using cutting planes for lower bounding, *eclipse-cp* is able to solve the two instances. On the two instances, it also explores the least number of nodes on average among the three solvers; however, its runtime performance is not as competitive as *cplex*.

Table 4.10: Comparison of *scherzo*, *bsolo*, *cplex*, *eclipse-lpr* and *eclipse-cp* on binate covering benchmarks (reference instances only) from logic synthesis.

benchmark	measure	*scherzo*	*bsolo*	*cplex*	*eclipse-lpr*	*eclipse-cp*
count.b	time	333.7	2275.5	0.7	0.7	1.8
	nodes	1429	31002	20	1	1
clip.b	time	0.1	7.7	0.8	3.4	1.9
	nodes	85	535	60	127	17
9sym.b	time	1.9	0.5	1.1	1.5	4.2
	nodes	318	43	31	1	1
jac3	time	2.6	3600*	1.5	2.6	7.8
	nodes	294	–	7	3	3
f51m.b	time	0.9	2897	1.9	2.3	5.0
	nodes	1562	45271	199	128	81
sao2.b	time	0.4	3600*	3.6	2.1	1.6
	nodes	285	–	602	35	3
5xp1.b	time	2.1	3600*	5.0	2.3	4.1
	nodes	2661	–	258	128	36
apex4.a	time	87.4	3600*	9.9	3600*	29.5
	nodes	33185	–	587	–	77
rot.b	time	3600*	3600*	12.5	3600*	507.0
	nodes	–	–	308	–	716
alu4.b	time	3600*	3600*	3600*	3600*	3600*
	nodes	–	–	–	–	–
e64.b	time	3600*	3600*	3600*	3600*	3600*
	nodes	–	–	–	–	–

This table compares *scherzo*, *bsolo*, *cplex*, *eclipse-lpr* and *eclipse-cp* on the binate covering reference benchmarks from logic synthesis. *Bsolo* is clearly not competitive with other solvers. *Scherzo* is competitive on easier benchmarks but runs much slower than *cplex* and *eclipse-cp* on two harder instances *apex4.a* and *rot.b*. *Eclipse-lpr* times out on these two instances because of its relatively weak lower-bounding technique. All solvers time out on the two most difficult instances *alu4.b* and *e64.b*.

Table 4.11: Comparison of *cplex*, *eclipse-lpr*, and *eclipse-cp* on P-classes of size 32 for the binate covering benchmarks from logic synthesis.

benchmark	measure	*cplex*	*eclipse-lpr*	*eclipse-cp*
count.b	time	0.7/0.4	0.7/0.02	2.0/0.1
	nodes	21/18	1/0	1/0
clip.b	time	0.4/0.3	2.8/0.8	1.1/0.7
	nodes	18/23	88/32	8/8
9sym.b	time	1.2/0.9	1.7/0.04	4.6/0.1
	nodes	48/65	1/0	1/0
jac3	time	2.5/3.8	3.3/0.3	8.1/1.0
	nodes	13/45	3/0	3/0
f51m.b	time	2.2/0.9	3.2/1.7	2.8/2.5
	nodes	325/314	168/94	68/86
sao2.b	time	1.6/0.7	3.3/2.6	3.1/4.8
	nodes	216/168	63/5	101/340
5xp1.b	time	6.2/5.0	3.2/1.7	3.1/1.5
	nodes	499/856	168/94	10/7
apex4.a	time	11.3/5.2	3600*	52.0/10.5
	nodes	588/494	–	211/73
rot.b	time	70.7/52.9	3000*	333.4/153.2
	nodes	3967/3706	–	455/208
alu4.b	time	3600*	3600*	3600*
	nodes	–	–	–
e64.b	time	3600*	3600*	3600*
	nodes	–	–	–

This table compares *cplex*, *eclipse-lpr* and *eclipse-cp* on P-classes of size 32 for the binate benchmarks from logic synthesis. The mean and standard deviation for runtime and nodes are reported. Overall, *cplex* has the best runtime behavior whereas *eclipse-cp* explores the least number of nodes on average. *Eclipse-lpr* times out on *apex4.a* and *rot.b*. With lower bounding using cutting planes, *eclipse-cp* is able to solve *apex4.a* and *rot.b*, but it is still much slower than *cplex*. All solvers time out on *alu.b* and *e64.b*.

4.5.4 Runtime Analysis

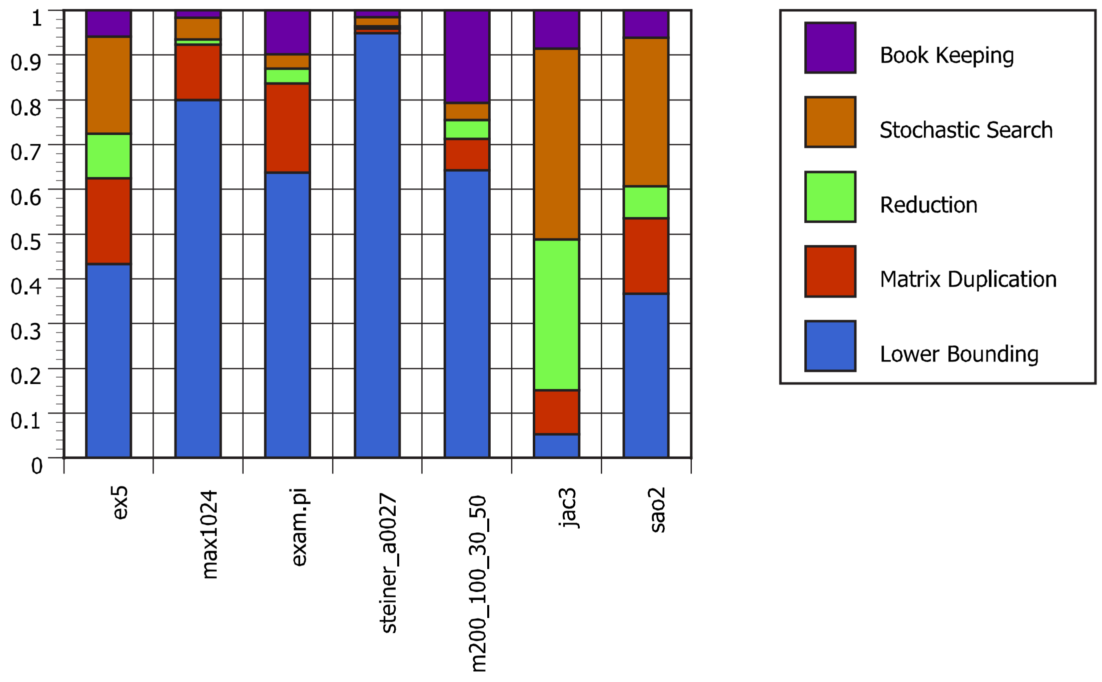
During an execution of *eclipse*, the CPU cycles are spent doing the following: (1) lower bounding, (2) matrix copying when generating new nodes, (3) reduction, (4) local search at each node, and (5) bookkeeping. Figure 4.2 takes a closer look at the breakdown of the execution time for *eclipse-lpr* and *eclipse-cp*. Shown in the top figure in Figure 4.2, the lower-bounding procedure is the most time consuming component of *eclipse-lpr* for the five unate benchmarks: over 40% of the total time is dedicated to doing lower bounding. For the two relatively easy binate benchmarks, *jac3* and *sao2*, the time spent on local search is most dominant. Unfortunately, we are unable to do a similar analysis for harder benchmarks such as *apex4* and *rot.b* because they remain unsolved by *eclipse-lpr*.

In the bottom figure in Figure 4.2, we include all the benchmarks in the top figure as well as two benchmarks that are unsolved by *eclipse-lpr*, namely, *apex4* and *rot.b*. For the unate benchmarks, the lower-bounding time in *eclipse-cp* is even more dominant: lower bounding consumes over 65% of the total execution time. For the binate benchmarks, the profiles for *jac3* and *sao2* are similar to that for *eclipse-lpr*. However, for *apex4* and *rot.b*, over 80% of the runtime are spent on doing lower bounding.

Overall, Figure 4.2 indicates that lower bounding is the dominant time-consuming factor in both *eclipse-lpr* and *eclipse-cp* in most cases. Any future improvement on *eclipse-lpr* and *eclipse-cp* should first focus on speeding up the lower-bounding procedures without loss of quality.

4.6 Summary

In this chapter, we introduced a new state-of-the-art branch-and-bound covering solver, *eclipse*. *Eclipse* implements well-known reduction techniques, branching variable selection strategies and search prune methods on top of efficient data structures. Three new features in *eclipse* are advanced lower-bounding techniques (Section 4.4.1),



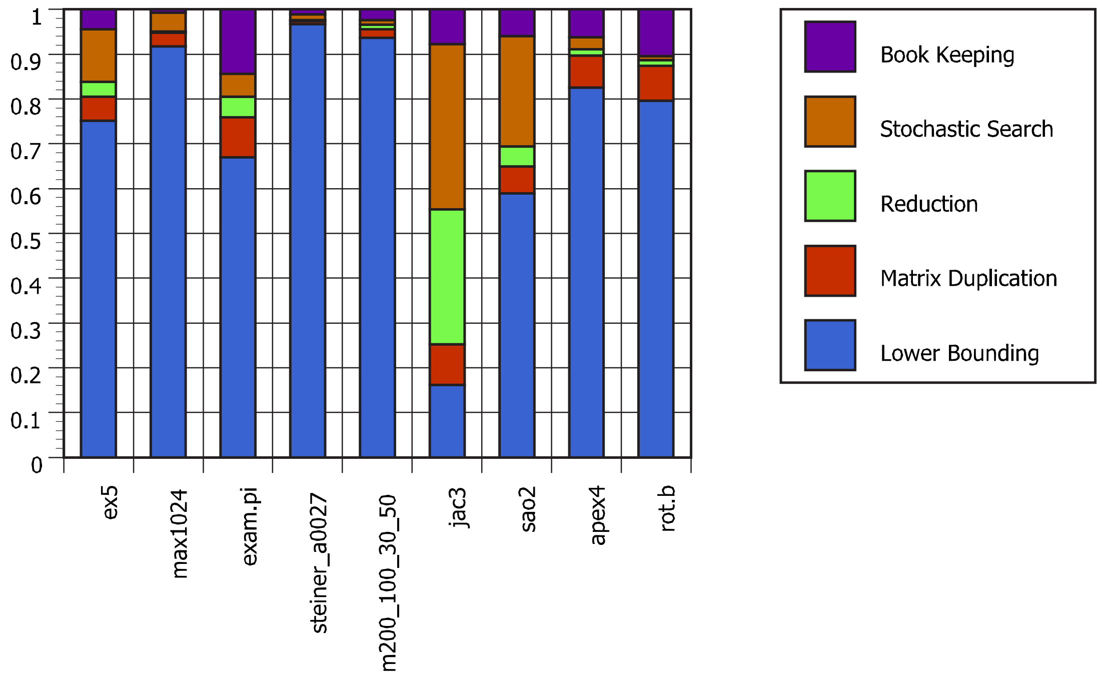


Figure 4.2: The breakdown of execution time for various components of *eclipse-lpr* (top) and *eclipse-cp* (bottom).

upper bounding with local search (Section 4.4.2), and best-first search-tree exploration strategy (Section 4.4.3).

For lower bounding, *eclipse* implements two advanced techniques: linear programming relaxation (LPR) and cutting planes (CP). We showed that these two lower-bounding techniques are able to consistently provide better or equal lower-bounds than MIS-based methods. For upper bounding, *eclipse* conducts a local search at each node of the search tree in order to find the best possible upper-bound. Through carefully designed search initialization method and termination criterion, the local search is able to find the best possible upper-bound early and yet without degrading the overall performance of *eclipse.* For search-tree exploration, *eclipse* conducts a best-first search where the first node to explore has the lowest lower-bound among all unexplored nodes. Best-first search enables *eclipse* to find the best possible upper-bound earlier than depth-first search.

Combining all the features above, we created two variations of *eclipse*: *eclipse-lpr* and *eclipse-cp.* Through statistically significant experiments, we showed that both *eclipse-lpr* and *eclipse-cp* can outperform traditional covering solvers by a big margin on the unate benchmarks from logic minimization. For the set covering and random benchmarks, *scherzo* and *aura*II are still the leading solvers. *Cplex* has the best performance on binate benchmarks from logic synthesis. *Eclipse-cp* stays competitive with *cplex* but is slower on some benchmarks. For all the benchmarks above, *eclipse-cp* consistently explores the least number of nodes among all solvers; however, its expensive lower-bounding procedure can sometimes slow it down significantly.

In order to further enhance the performance of *eclipse*, the following areas are worth pursuing:

1. **The Generation of Gomory Cuts.** We have observed throughout our experiments that even though *eclipse-cp* is able to explore very few nodes compared to other solvers, it doesn't consistently have the fastest runtime. Recall that in the lower-bounding method with cutting planes (Section 4.4.1), doing lower bounding with CP involves (1) solving the original *LP* problem, (2) generating the Gomory cuts and creating a new *LP* problem, and (3) solving the new

LP problem. We observe that generating the Gomory cuts are the most time consuming part of this process. Currently, we don't have control over how the Gomory cuts are generated because it is done internally by *cplex*. It is worth investigating whether it pays to generate the Gomory cuts directly. By generating the cuts ourselves, we may be able to decide the number of Gomory cuts to generate as well as which ones to generate. Good decisions here may lead to a better balance between the quality of the lower-bounds returned and the time consumed by the lower-bounding procedure.7:49 am

2. **Incremental Lower Bounding.** The lower-bounding procedure using either LPR or CP is implemented in *eclipse* in two ways: incremental and non-incremental. When implemented *incrementally*, an *LP* model is created at program initialization according to the covering matrix. When a variable is assigned, e.g., $x_i = 1$, either due to branching or reduction, the model is updated with the *constraint* $x_i = 1$. The lower bounding is then carried out based on the updated model. When *eclipse* backtracks, the model is updated accordingly. For example, if *eclipse* backtracks to level t, then all the constraints corresponding to the assignments made below level t are removed from the model. When LPR and CP are implemented *non-incrementally*, a new *LP* model is created at each node of the search-tree from the current covering matrix. Implementing the lower-bounding procedure incrementally avoids creating a new model at each node and was expected to be more efficient than the non-incremental approach. However, preliminary experiments show that the two approaches perform very similarly in runtime. It is likely that we haven't found the most efficient way of updating the *LP* model in *cplex*.

3. **Dynamic Local-Search Termination Criterion.** As Figure 4.2 indicates, the local-search procedure can sometimes take up a significant portion of the runtime in both *eclipse-lpr* and *eclipse-cp*. According to our parameterized local-search termination criterion, the nodes on the same level of the search-tree are allocated the same number of flips. Instead of running the local search for such a fixed number of flips, more dynamic search termination criterion may

be used to reduce the amount of search. For example, the search may still run up to its allocated number of flips but may terminate early if no improvements have been made for a certain period of time.

4. **Choosing the best lower-bounding technique.** On the set covering and randomly generated benchmarks, both *eclipse-lpr* and *eclipse-cp* are slower than *scherzo* and *aura*II mainly because the overhead of the expensive lower-bounding procedures. We observe that *eclipse* can be made competitive simply by switching to the MIS-based lower bounding. This raises the question of how we decide which lower-bounding technique to use for a given benchmark. One possible indicator is the distance between the lower-bound found at the root and the cost of the optimal solution. Recall that *eclipse-cp* outperforms *scherzo* and *aura*II on unate benchmarks from logic minimization and binate benchmarks from logic synthesis. We observe that for these benchmarks, the distances between the lower-bound found at the root and the cost of the optimal solution are very small (mostly 0 or 1 but can be up to 3; please refer to Table 2.1 and Table 2.2). However, for the set covering and randomly generated benchmarks, such distances are normally larger (ranging from 1 to 29 but mostly 3 and up; please refer to Table 2.1). Even though we don't know the optimum up front, a local search for the upper-bound at the root of the search-tree can normally provide a very good estimate. Therefore, we may choose to measure the distance between the lower-bound and the upper-bound at the root in order to determine which lower-bounding heuristic to use.

Chapter 5

Local Search Algorithms

Branch-and-bound *MinCostSat* algorithms explore the search space systematically and they reduce the potentially exponential search space by using reduction and search pruning techniques. However, branch-and-bound algorithms don't scale well. We have seen that large benchmarks such as *test4.pi* and *alu4.b* remain unsolved by any branch-and-bound covering solver. For such problems, we have to rely on local-search algorithms to find optimal or near optimal solutions.

5.1 Introduce the Eclipse-stoc Algorithm

First stated in Section 1.4, the two main challenges faced by local-search *MinCostSat* algorithms are (1) the feasibility issue: finding a solution that satisfies all the constraints and (2) the quality issue: finding a feasible solution with the minimum cost. In this chapter, we present a new SAT-based local-search algorithm for *MinCostSat* called *eclipse-stoc*. *Eclipse-stoc* attempts to address the two challenges of *MinCostSat* as follows.

To address the first challenge, we rely on local-search *Sat* algorithms to find feasible solutions (and hopefully, good quality solutions). We consider two prominent local-search *Sat* algorithms: *walksat* [70] and *unitwalk* [71]. According to our experiences with *Sat* benchmarks and solvers [72], no single solver dominates the other on all benchmarks. In most cases, *walksat* works better for randomly generated benchmarks with little structure while *unitwalk* works better for more structured instances. However, facing an unknown benchmark, how do we decide its "structuredness"? In this work, we present two measures, one static and one dynamic. The static measure is simply the percentage difference between the positive and negative literals in the benchmark. The dynamic measure is *variable immunity*, a parameter we introduced in [72] that gauges the structuredness of the benchmarks. Based on either measure, we can effectively choose the better solver between *unitwalk* and *walksat* to solve a given benchmark.

Once a feasible solution is found, we address the second challenge by conducting a local search around the solution in the pursuit of an optimal solution. The local search uses ideas from *walksat*, *gsat* [73] and *tabu* search [74]. The solver *gsat* was the first successful local-search algorithm for *Sat* and *tabu* search increases the robustness of our algorithm by ensuring that the value of a variable cannot be changed if the variable is in the *tabu* list. The *tabu* list contains the variables changed most recently.

In the rest of this chapter, we first present the two measures of benchmark structuredness. We then introduce modified versions of *unitwalk* and *walksat* that are used to find feasible solutions for *MinCostSat*. After that, we present the local-search techniques for finding good quality solutions. We then describe the *eclipse-stoc* algorithm, which combines all these concepts along with the reduction techniques from branch-and-bound solvers. In the experimental section, we study the performance of *eclipse-stoc* on *native MinCostSat* benchmarks from two-level logic minimization, logic synthesis and `ATPG`.

5.2 Benchmark Profiling

Benchmark profiling is the idea of putting benchmarks in different categories so that each benchmark is solved by a solver that is most suitable for the benchmark's category. In this chapter, we consider the *native MinCostSat* benchmarks that include all the *UCP* and *BCP* benchmarks (see Table 2.1 and Table 2.2) and non-covering benchmarks from `ATPG` (see Table 2.3). We consider two local search based *Sat* solvers *unitwalk* and *walksat* and we categorize the benchmarks as either *structured* or *unstructured*. We introduce two measures of benchmark structuredness next.

5.2.1 The Static Measure

The static measure is the *percentage difference* between the positive literals m and negative literals n in the formula:

$$\text{percentage difference} = \frac{|n-m|}{n+m}. \tag{5.1}$$

We calculate the percentage differences for the benchmarks from Table 2.1– 2.3 and report them in Table 5.1 and Table 5.2. Not surprisingly, for all the covering benchmarks, the percentage differences are above 60% with most of them over 90%. For the non-covering benchmarks from `ATPG`, the percentage differences are around 10%. This means in these benchmarks, the positive and negative literals are fairly balanced.

5.2.2 The Dynamic Measure

The dynamic measure is called *variable immunity*. Before we show how variable immunity is calculated, we present some additional *Sat* concepts:

1. A *Sat* formula is empty if all its clauses are either satisfied or conflicted. A clause is satisfied when one of its literals is true. A clause is conflicted if all of its literals are false.

2. When a unit clauses $\{x\}$ exists in a *Sat* formula, in order to satisfy the formula, the unit literal x has to be set true. This unit literal is then propagated: all clauses containing x are then satisfied and all $\bar{x}$'s are removed from its residing

clauses. New unit clauses can arise as a result and they are propagated similarly. This process stops when no unit clauses exist and we call this process *unit propagation*.

```
Algorithm: calculate-variable-immunity
input: a Boolean formula $F$ in CNF (n = # of variables)
output: the value of variable-immunity
method:
 1 variable-immunity = 0
 2 random-assignments = 0
 3 for $i := 1$ to MAX-ITERATIONS do
 4     while $F \neq \emptyset$ do
 5         assign a random value to an unassigned variable chosen at random
 6         random-assignments++
 7         do unit propagation until no unit clauses exist
 8     end do
 9     restore $F$
10 end do
11 variable-immunity = random-assignments / (n $\times$ MAX-ITERATIONS)
12 return variable-immunity
```

Figure 5.1: The algorithm for calculating the variable immunity

We show how to calculate variable immunity in Figure 5.1. Our experiments with existing SAT benchmarks show that structured instances have low variable immunity (normally below 0.1 or 10%). Intuitively, for these benchmarks, the variables have intricate and tight relationships, and one variable's assignment may cause a lot of other variable to take on certain values. This leads to a low variable immunity because only a few random assignments are needed to make the formula empty. On the other hand, unstructured instances have a higher variable immunity because many variable assignments have little impact on other variables and a large portion of the variables have to be assigned randomly instead of assigned by unit propagation.

In Table 5.1, the variable immunity is above 90% for logic minimization and randomly generated benchmarks; for the Steiner set-covering benchmarks, the vari-

Table 5.1: The static and dynamic measures of structuredness on unate covering benchmarks.

	static	dynamic
benchmark	% diff	immunity
lin_rom	100	95.7
exam.pi	100	99.9
bench100.pi	100	99.9
prom2	100	96.3
prom2.pi	100	95.9
max1024	100	95.5
max1024.pi	100	92.4
ex5.pi	100	99.9
ex5	100	98.8
test4.pi	100	99.9
steiner_a0009	100	75.4
steiner_a0015	100	65.9
steiner_a0027	100	54.3
steiner_a0045	100	46.1
steiner_a0081	100	40.1
m100_100_10_30	100	92.9
m100_100_10_15	100	99.9
m100_100_10_10	100	99.7
m200_100_10_30	100	99.9
m200_100_30_50	100	99.9

The table presents the static and dynamic structuredness measures of the *UCP* benchmarks. Since all benchmarks are unate, the percentage differences (static measure) is 100%. For variable immunity (dynamic measure), all benchmarks except the Steiner set-covering instances have a value above 90%. Compared to the `ATPG` benchmarks in Table 5.2, both measures here are much higher.

Table 5.2: The static and dynamic measures of structuredness on binate covering benchmarks.

	static	dynamic
benchmark	% diff	immunity
count.b	92.7	98.1
clip.b	90.8	99.1
9sym.b	91.7	99.9
jac3	85.8	99.2
f51m.b	92.9	98.4
sao2.b	88.7	98.0
5xp1.b	94.6	99.6
apex4.a	61.7	92.7
rot.b	87.1	97.3
alu4.b	90.4	97.7
e64.b	79.0	99.8
c432_F37gat@1	10.6	7.5
misex3_Fb@1	10.6	1.2
c1908_F469@0	11.0	3.0
c6288_F69gat@1	10.4	1.7
c3540_F20@1	11.3	1.5

This table presents the static and dynamic structuredness measures for the binate covering and the non-covering benchmarks. Both measures for the `ATPG` set are significantly lower than those for the logic synthesis set.

able immunity decreases as the number of columns grows but it is still at 40% for *steiner_a0081*, the largest such benchmark we consider. In Table 5.2, the variable immunity is above 90% for all logic synthesis benchmarks but below 10% for all the `ATPG` benchmarks. We observe very strong correlation between the static and dynamic measures. Both measures are much lower for the `ATPG` benchmarks.

Since there exists a sharp threshold for both measures to categorize the *Min-CostSat* benchmarks into the *same* two groups, either measure can be used in the *eclipse-stoc* algorithm. We choose to use variable immunity in the *eclipse-stoc* algorithm because we believe it is the more robust measure.

5.3 Addressing the Feasibility Issue

After calculating the variable immunity, we can decide which local-search solver to use to find feasible solutions for *MinCostSat*. We use *unitwalk* for benchmarks with low variable immunity and *walksat* for high variable immunity (both algorithms are adapted to *MinCostSat* after some minor changes). We set the threshold at 8% because it is effective in separating structured and unstructured benchmarks. Both *unitwalk* and *walksat* generate initial assignments for the variables and these assignments are modified by complementing (flipping) the value of some variable in each step. We first present the modified *unitwalk* algorithm and then present the modified *walksat* algorithm.

5.3.1 The unitwalk Algorithm Adapted for BCP

The *unitwalk* algorithm takes advantage of unit propagation whenever possible, delaying variable assignment until unit clauses are resolved. Initially, the variables are randomly assigned to *true* or *false*. An iteration of the outer loop of *unitwalk* is called a *period*. At the beginning of a period, a permutation of the variables is randomly chosen. The algorithm will start doing unit propagation using the assignment for the first variable in the permutation. The unit propagation process modifies the current assignment and will continue as long as unit clauses exist. When there are no unit clauses left and some variables remain unassigned, the first unassigned variable in the permutation along with its current assignment is chosen to continue the unit propagation process. At least one variable is flipped during each period, thus ensuring progress. If at the end of a period, the formula becomes empty, then the current assignment is returned as the satisfying solution.

We modify the *unitwalk* algorithm slightly: instead of generating a random assignment at the beginning of a period, we set all non-zero cost variables to false and set all zero cost variables to true or false randomly. This way, we generate an initial assignment with a zero cost. We present the algorithm in detail in Figure 5.2.

```
Algorithm: unitwalk-adapted
 input: a Boolean formula $F$ in CNF containing $n$ variables $x_1, \cdots, x_n$
        the cost function of the variables $Cost$
 output: a satisfying assignment $A$ or "No solution found"
 method:
  1 for each variable $x_i$
  2     if $cost(x_i) > 0$, then $A[i] = 0$
  3     else assign $A[i]$ randomly
  4 end do
  5 for $p := 1$ to MAX-PERIODS do
  6     $\pi :=$ random permutation of $1, \cdots, n$
  7     $G := F$; $flipped :=$ false
  8     for $i := 1$ to $n$ do
  9         while $G$ contains a unit clause do
 10             pick a unit clause $\{x_j\}$ or $\{\overline{x}_j\}$
 11             if this clause is not satisfied by $A$ and $G$ doesn't contain the
 12                opposite unit clause then flip $A[j]$ and set $flipped := true$
 13             $G := G[v_j \leftarrow A[j]]$
 14         end do
 15         if variable $v_{\pi[i]}$ still appears in $G$ then $G := G[v_{\pi[i]} \leftarrow A[\pi[i]]]$
 16     end do
 17     if $G$ contains no clauses, then return $A$
 18     if $flipped$ = false, then choose $j$ randomly from $1, \cdots, n$ and flip $A[j]$
 19 end do
 20 Output "No solution found"
```

Figure 5.2: The *unitwalk* algorithm adapted for *MinCostSat*.

As lines 8-13 show, the algorithm will continue doing unit propagation as long as unit clauses exist. During the search in *unitwalk*, a variable v with non-zero cost is assigned true under only two conditions: (1) when this assignment is part of the unit propagation process in which v becomes a unit literal, v has to be flipped from false

```
Algorithm: walksat-adapted
input: a Boolean formula $F$ in CNF containing $n$ variables $v_1, \cdots, v_n$
          the cost function of the variables $Cost$
output: a satisfying assignment $A$ or "No solution found"
method:
 1  for $t := 1$ to MAX-TRIES do
 2      for each variable $v_i$ do
 3          if $cost(v_i) > 0$, then $A[i] = 0$
 4          else assign $A[i]$ randomly
 5      end do
 6      for $i := 1$ to MAX-FLIPS do
 7          choose an unsatisfied clause $C$ at random
 8          with probability $p$
 9              choose a variable $x$ in $C$ at random
10          with probability $1 - p$
11              choose a variable $x$ in $C$ that minimize the
12              number of unsatisfied clauses when flipped
13          modify A by flipping the value of $x$ in $A$
14          if $A$ satisfies $F$, return $A$
15      end do
16  end do
17  Output "No solution found"
```

Figure 5.3: The walksat algorithm adapted for *MinCostSat*.

to true (line 12). (2) when no flip is made for the period, v is randomly chosen and flipped from false to true (line 17). The first condition attempts to keep the formula satisfied at the expense of increasing the cost of the current assignment. The second condition can happen because we need to make sure that at least one flip is made during each period, but this is a rare event.

5.3.2 The walksat Algorithm Adapted for BCP

We present the adapted *walksat* algorithm in Figure 5.3. In *walksat*, an initial assignment is chosen the same way as in *unitwalk*. The variable to be flipped is chosen from a randomly picked unsatisfied clause (line 7). *walksat* chooses a variable to flip with the following heuristic: with probability p (the noise parameter), randomly choose a variable in the clause (line 9); with probability $1 - p$, pick a variable that minimizes the number of unsatisfied clauses when flipped (lines 11–12). The chosen variable is then flipped on line 13. If a solution is not reached after a specified number of flips, *walksat* tries again with a new random assignment as its starting point.

5.4 Addressing the Quality Issue

Once a solution is found by using either *unitwalk* or *walksat*, we can attempt to find an optimal solution by doing a local search around the solution found. The details of the local-search algorithm are presented in Figure 5.4.

The neighborhood-search algorithm combines ideas from random walk, *walksat* and *gsat*. In addition, to increase the robustness of the search, *tabu* search is also added: we maintain a tabu list that contains the most recently flipped variables and no variable in the tabu list can be flipped. During each iteration of the main loop, with probability p_1, a variable is randomly chosen (lines 3–4). With probability p_2, if unsatisfied clauses exist, choose one at random and choose the variable in the clause that minimizes the number of unsatisfied clauses when flipped; otherwise, choose a *true* variable with non-zero weight at random (this is an attempt to reduce the cost of the solution) (lines 5–10). With probability $1 - p_1 - p_2$, choose the variable that minimizes the number of unsatisfied clauses when flipped (lines 11–13). If the chosen variable is not in the tabu list, it is flipped and added to the tabu list (lines 14–16). The length of the tabu list has little impact on *eclipse-stoc* and any small positive integer suffices. In our experiments, the default is 3. If the resulting assignment is feasible and has a cost less than the best known solution, then the best solution is

```
Algorithm: neighborhood-search
input: a Boolean formula $F$ in CNF
       the cost function of the variables $Cost$
       a solution A
output: the best cost found
method:
 1 best-solution = $A$
 2 for $i := 1$ to MAX-FLIPS do
 3     with probability $p_1$
 4         choose a variable $x$ at random
 5     with probability $p_2$
 6         if unsatisfied clauses exist
 7         then choose an unsatisfied clause $C$ at random
 8                 choose a variable $x$ in $C$ that minimizes the
 9                 number of unsatisfied clauses when flipped
10         else choose a true variable $x$ with non-zero weight at random
11     with probability $1 - p_1 - p_2$
12             choose the variable $x$ that minimizes the number of
13             unsatisfied clauses when flipped
14     if $x$ is not in the tabu list
15     then modify current-solution by flipping the value of $x$ in $A$
16             add $x$ to the tabu list
17     if $A$ is feasible and cost-of( $A$ ) $<$ cost-of( best-solution )
18     then best-solution = $A$
19 end do
20 return best-solution
```

Figure 5.4: The neighborhood-search algorithm that searches for optimal solutions.

updated with the current solution (lines 17–18). The main loop runs for MAX-FLIPS number of iterations, which is determined by the size of the formula.

```
Algorithm: Eclipse-stoc
input: a Boolean formula $F$ in CNF
       the cost function of the variables $Cost$
output: the best cost found
method:
 1 best = INT-MAX
 2 current-solution = NULL
 3 variable-immunity = calculate-variable-immunity (F)
 4 for $i$ := 1 to MAX-ITERATIONS do
 5     if variable-immunity < THRESHOLD
 6         current-solution = unitwalk-adapted( $F$, $Cost$ )
 7     else
 8         current-solution = walksat-adapted( $F$, $Cost$ )
 9     current-solution = neighborhood-search( $F$, $Cost$, current-solution )
10     if cost-of( current-solution ) < best
11     then best = cost-of( current-solution )
12 end do
13 return best
```

Figure 5.5: The eclipse-stoc algorithm for *MinCostSat*.

5.5 The Eclipse-stoc Algorithm

The *eclipse-stoc* algorithm in Figure 5.5 combines all the algorithms introduced in this chapter. On line 3, the *eclipse-stoc* algorithm calculates the variable immunity and decides which one of *unitwalk* and *walksat* to use to find feasible solutions. On lines 5–8, the chosen solver finds a feasible solution for *MinCostSat*; the neighborhood-search is done on line 9; and the best solution is updated on lines 10–11.

5.6 Experimental Results

Eclipse-stoc, like many other local-search solvers, doesn't have a natural termination criterion: it can run for a fixed number of iterations, a fixed amount of time, or terminate when a solution with a desired quality is found. In our experiments with *eclipse*-stoc, we use one of two termination criteria:

1. Termination Criterion A: each *eclipse-stoc* run is terminated when the best known value is found for the first time or when a timeout has been reached.
2. Termination Criterion B: for large benchmarks whose optimum is unknown, *eclipse-stoc* is allowed to run for the same duration the branch-and-bound solvers are allowed to run.

Termination criterion A allows us to gauge the ability of *eclipse-stoc* in finding an optimal solution. It was also used in the early design stage to tune the local search.

5.6.1 Unate Covering Comparisons Under Termination A

Table 5.3 presents the *eclipse-stoc* results under *termination criterion A* on the P-classes of the unate covering benchmarks. We study two variations of *eclipse-stoc*, one applies an initial reduction to the covering matrix and the other one does not. We terminate each experiment run when either the best known solution is found or when the timeout of 300 seconds is exceeded.

Comparisons between with and without reduction. On the logic minimization benchmarks, *eclipse-stoc* is generally faster in finding the best known solutions with an initial reduction than without an initial reduction (e.g. *exam.pi* and *bench1.pi*). On *bench1.pi*, without an initial reduction, *eclipse-stoc* times out at 300 seconds 19 out of 33 runs on the P-class instances; with the reduction, it finds the optimum in only 4.9 seconds on average. On *max1024* and *max1024.pi*, *eclipse-stoc* times out with or without doing initial reduction. This indicates that *eclipse-stoc* can sometimes get stuck in local minima and it echoes a point we made in Section 4.4.2: upper bounding with local search at the root of the search tree may not find the best possible upper-bound. The Steiner set-covering and randomly generated benchmarks are not susceptible to reduction; therefore, we observe no significant differences between the two variations of *eclipse-stoc* on these benchmarks.

Table 5.3: The *eclipse-stoc* results on unate covering benchmarks under termination criterion A.

	No Reduction		With Reduction	
benchmark	runtime	flip	runtime	flip
lin_rom	0.7/0.9	45890/77418	0.4/0.2	30842/23370
exam.pi	26.0/22.2	925481/836855	2.9/1.7	121631/160188
bench1.pi	279.8/74.1*	1.03e7/3.07e6	4.9/3.8	492360/433963
prom2	1.5/0.5	45673/22314	1.5/0.6	49369/40488
prom2.pi	1.7/0.6	60363/27839	1.3/0.4	42021/22611
max1024	300.0/0$^\diamond$	4.75e6/16890	300.0/0$^\diamond$	5.63e6/617978
max1024.pi	300.0/0$^\diamond$	4.58e6/198374	300.0/0$^\diamond$	5.52e6/896550
ex5.pi	7.0/5.7	356154/350382	2.8/1.3	93156/96295
ex5	6.4/8.0	230983/209635	2.9/1.0	99890/77463
test4.pi	12.8/1.4	22636/6976	10.4/1.3	25302/9917
steiner_a0009	0.01/0	3/2	0.01/0	3/2
steiner_a0015	0.01/0	3/2	0.01/0	3/2
steiner_a0027	0.01/0	27/27	0.01/0	27/27
steiner_a0045	6.8/5.6	1.82e6/1.49e6	7.0/5.7	1.82e6/1.49e6
steiner_a0081	0.02/0.02	1813/3162	0.02/0.02	1813/3162
m100_100_10_30	0.01/0.01	80/18	0.01/0.01	80/18
m100_100_10_15	0.05/0.05	11533/12177	0.05/0.05	11533/12177
m100_100_10_10	0.08/0.07	18620/18241	0.1/0.1	18620/18241
m200_100_10_30	0.02/0.01	1930/1702	0.02/0.01	1930/1702
m200_100_30_50	0.02/0.01	201/115	0.02/0.01	201/115

* *eclipse-stoc* times out at 300 seconds on 19 out of 33 instances.

$^\diamond$ *eclipse-stoc* times out at 300 seconds.

This table presents the *eclipse-stoc* results on the P-classes of size 32 of the unate covering benchmarks. We study two variations of *eclipse-stoc*, with and without initial reduction. We terminate each experiment when either the best known solution is found or the timeout is exceeded. On the logic minimization benchmarks, *eclipse-stoc* is generally faster in finding the best known solution with reduction. No significant differences between the two variations of *eclipse-stoc* can be observed on set-covering and randomly generated benchmarks.

Comparisons between *eclipse-stoc* and *eclipse-cp*. Even though it is hardly fair to compare a local-search solver with a branch-and-bound solver, we would like to make a few interesting observations by comparing the *eclipse-stoc* data in Table 5.3 with the *eclipse-cp* data in Table 4.7. On *bench1.pi*, *eclipse-stoc* takes 4.9 seconds on average to find an optimal solution; however, *eclipse-cp* only takes 4.7 seconds on average to find and prove the optimality of such a solution. More surprisingly, on *max1024* and *max1024.pi*, *eclipse-stoc* times out at 300 seconds but *eclipse-cp* can find the optimal solutions and prove their optimality in 27.5 and 23.0 seconds, respectively. In both cases, it means that *eclipse-cp* can sometimes be more effective in finding an optimal solution than *eclipse-stoc*. We believe this is due to the fact that *eclipse-cp*'s branch-and-bound search style allows more diversified local search at each node of the search-tree and avoids getting stuck in local minima more effectively than *eclipse-stoc*.

5.6.2 Binate Covering Comparisons Under Termination A

Table 5.4 presents the *eclipse-stoc* results on the P-classes of size 32 of the binate covering benchmarks. The setup is the same as in Table 5.3. With reduction, *eclipse-stoc* does significantly better on *sao2.b*, *apex4.b*, and *rot.b*. In particular, *eclipse-stoc* times out on *apex4.b* without reduction, but with reduction, *eclipse-stoc* solves it in 3.9 seconds on average. We observe that reduction doesn't play a significant role on the `ATPG` benchmarks.

5.6.3 Experiments Under Termination Criterion B

In the benchmarks we considered in this chapter, three of them remain unsolved by any branch-and-bound *MinCostSat* solvers. We run *eclipse-stoc* under termination criteria B in which *eclipse-stoc* is allowed to run for an hour on each instance. In Table 5.5, we report the previous best known solution found by *cplex* in the second column and the best solution found by *eclipse-stoc* in the third column. *Eclipse-stoc* is able to find a better solution than *cplex* with one hour execution time.

Table 5.4: The *eclipse-stoc* results on binate covering benchmarks under termination criterion A.

	No Reduction		With Reduction	
benchmark	runtime	flip	runtime	flip
count.b	0.1/0.02	1856/877	0.28/0.02	1926/957
clip.b	0.1/0.02	3266/2186	0.1/0.01	1374/1318
9sym.b	0.1/0.01	563/316	0.6/0.03	803/674
jac3	0.7/0.2	10686/9406	1.7/0.2	4786/4063
f51m.b	0.08/0.01	1140/819	0.08/0.01	408/289
sao2.b	53.3/15.8	5.49e6/1.63e6	1.95/1.34	287600/206849
5xp1.b	0.2/0.1	3403/6451	0.3/0.1	4305/4299
apex4.a	300/0*	1.179e6/14719	3.9/0.9	83118/79266
rot.b	57.2/12.7	2.91e6/600918	7.09/5.45	602138/527877
alu4.b	0.5/0.1	7894/6608	0.6/0.03	2347/1307
e64.b	0.2/0.1	13644/11370	0.1/0.01	2934/2154
c432_F37gat@1	52.8/83.5	928565/1.35e6	49.3/78.2	890872/1.12e6
misex3_Fb@1	1.4/0.7	2908/1418	1.3/0.6	2876/1298
c1908_F469@0	157.4/123.5	30988/26011	154.4/113.3	29876/25691
c6288_F69gat@1	33.4/83.82	18493/46499	32.0/78.24	17654/43309
c3540_F20@1	300/0*	231512/97490	300/0*	222987/84334

* *eclipse-stoc* times out at 300 seconds.

This table presents the *eclipse-stoc* results on the P-classes of size 32 of the binate covering and non-covering benchmarks. The setup is the same as in Table 5.3. Each experiment is terminated when either the best known solution is found or the timeout of 300 seconds is exceeded. With reduction, *eclipse-stoc* does significantly better on benchmarks such as *sao2.b*, *apex4.b*, and *rot.b*. In particular, *eclipse-stoc* times out on *apex4.b* without reduction, but with reduction, *eclipse-stoc* solves it in 3.9 seconds on average. Reduction does not play a significant role on the `ATPG` benchmarks. On *c3540_F20@1*, both variations time out.

Table 5.5: Find new best solutions with *eclipse-stoc*.

	previous best	current best
benchmark	cplex	eclipse-stoc
test4.pi	101	94
alu.b	50	50
e64.b	48	48

On *test4.pi*, one of the three benchmarks whose optimum has not been proven, *eclipse-stoc* is able to find a better solution within an hour.

5.7 Summary

In this chapter, we presented a new local-search *MinCostSat* solver. We introduced two measures, on static and one dynamic, to gauge the structuredness of the benchmarks. *Eclipse-stoc* chooses either *unitwalk* or *walksat* to find a feasible solution based on the variable immunity measure. It then utilizes a local-search to look for an optimal solution near the solution just found. Such a process repeats under a termination criterion is met. *Eclipse-stoc* is effective in finding an optimal solution quickly on most benchmarks. However, it may get stuck in local minima and fails to find an optimal solution within a given time frame. Our experiments also show that applying an initial reduction to the covering matrix can often enhance the performance of *eclipse-stoc*.

Chapter 6

Applications and Case Studies

In this chapter, we first present two applications of *MinCostSat*. The first one is unate covering problem with application in two-level logic minimization. The second one is *group-partial Max-Sat* with application in FPGA detailed routing. We then present three case studies on *MinCostSat*. The first case study involves the introduction of a new branch-and-bound *Max-Sat* solver, *qtmax*, and its performance comparisons with *cplex* and *eclipse-lpr*. The second case study introduces a simple non-covering *MinCostSat* solver that utilizes neither reduction nor advanced lower bounding. We compare its performance with *bsolo*, a leading branch-and-bound solver for non-covering *MinCostSat* problems, on benchmarks from the minimum-size test pattern problem. The last case study compares *cplex*, *eclipse-cp* with a leading *Sat* solver on some typical *Sat* instances and reveals some surprising results.

6.1 UCP and Two-Level Logic Minimization

Solving the unate covering problem is an integral part of doing two-level logic minimization. We first provide some background information on two-level logic minimization and show how to minimize it. We then present experimental results demonstrating that the two-level logic minimizer *espresso* can be improved by incorporating *eclipse-cp*.

6.1.1 Background

Two levels of logic are the minimum required to implement an arbitrary Boolean function. Compared to multilevel logic, two-level logic has two key advantages: speed and simplicity. It speed, or the lack of delay, is due to fact that the electronic signals propagate through only two levels. It is also simpler to design and analyze than multi-level logic. In fact, properties of multi-level logic can often be simplified and modelled by two-level logic. Next, we briefly discuss the representation of two-level logic, some of its properties and its cost measures.

Sums of Products and Products of Sums. A *literal* represents an uncomplemented or a complemented Boolean variable, e.g. x_i or $\overline{x}_i$. A two-level n-variable Boolean formula can be expressed in a number of canonical forms. Relevant to this thesis are the *sum of product* (SOP) of n literals called *minterms* and the *product of sum* (POS) of n literals called *maxterms*. Here, a product represents a conjunction and a sum represents a disjunction. The objective of two-level logic minimization is to re-write the formula with an equivalent formula that contains fewer terms and fewer literals in each term. For example, a SOP formula for the 3-variable *majority function* (true iff at least two of the three variables are true) can be written by inspection as:

$$f_{maj} = \overline{x}_1 x_2 x_3 + x_1 \overline{x}_2 x_3 + x_1 x_2 \overline{x}_3 + x_1 x_2 x_3,$$

which minimizes to

$$f_{maj} = x_1 x_2 + x_1 x_3 + x_2 x_3.$$

Similarly, a POS formula for the 3-variable *threshold function* (true iff at most one of the three variables is true) can be written by inspection as

$$f_{thr} = (\overline{x}_1 + \overline{x}_2 + \overline{x}_3)(x_1 + \overline{x}_2 + \overline{x}_3)(\overline{x}_1 + x_2 + \overline{x}_3)(\overline{x}_1 + \overline{x}_2 + x_3),$$

and minimized to

$$f_{thr} = (\overline{x}_1 + \overline{x}_2)(\overline{x}_1 + \overline{x}_3)(\overline{x}_2 + \overline{x}_3).$$

Interchangeably, SOP and POS forms are also known as the disjunctive and conjunctive normal forms. In general, we refer to terms contained in the SOP form as *product terms* which are, as illustrated earlier, the primary objective of logic minimization. In contrast, terms contained in the POS form are typically called *clauses*.

Implicants and Prime Implicants. An implicant of a function f is a product term that is included in f. For example, if $f = x\bar{y} + xz + \bar{y}z$, two of its implicants include $x\bar{y}$ and $x\bar{y}z$. A prime implicant of f is an implicant that doesn't include any other implicant. In the previous example, $x\bar{y}$ is a prime implicant whereas $x\bar{y}z$ is not because it contains the implicant $x\bar{y}$. We will see next that the prime implicant $x\bar{y}$ has less cost than $x\bar{y}z$ because it has one fewer literal.

Cost Function for Two-Level Logic. Programming logic arrays (PLAs) are well-known structures to implement two-level logic functions [19, 1]. A PLA is a rectangular macro-cell consisting of an array of transistors aligned to form rows in correspondence with product terms and columns in correspondence with inputs and outputs. The input and output columns partition the array into two subarrays, called *input* and *output planes*, respectively. Each row of a PLA is in one-to-one correspondence with a product term of the SOP representation. Each transistor in the output plane relates to the dependence of a scalar output on a product term. Therefore, the primary goal of logic minimization is the reduction of product terms and a secondary one the reduction of literals. For example, $f = xy + \bar{x}y + x\bar{z}$ has a minimal representation $f' = y + x\bar{z}$ with one fewer product term and three fewer literals than the original.

6.1.2 Minimizing the Two-Level Logic

Two-level logic minimization is based on the following theorem due to Quine [75]:

Theorem 4 *A minimal SOP must always consist of a sum of prime implicants if* *any definition of cost is used in which the addition of a single literals to any formula increases the cost of the formula.*

A well-known two-level logic minimization procedure is called the Quine-McCluskey procedure. It first generates all the prime implicants of the SOP formula and then chooses a subset of them to cover all the minterms. As an example, let's consider the following SOP representing a two-level Boolean formula with three variables:

$$f(x) = xy + xy\bar{z} + \bar{y}z + \bar{x}\bar{z} + xy\bar{z} \tag{6.1}$$

The minterms for $f(x)$ are $m_1 = \bar{x}\bar{y}\bar{z}$, $m_2 = \bar{x}\bar{y}z$, $m_3 = \bar{x}y\bar{z}$, $m_4 = x\bar{y}z$, $m_5 = xy\bar{z}$ and $m_6 = xyz$.

Compute the Prime Implicants. The prime implicants of a two-level formula can be computed using the *tabular method* or the *iterated consensus method* [19, 1]. We will not present these techniques here since our main interest is in the covering procedure. Using either technique, we find that the prime implicants for $f(x)$ in (6.1) are: $p_1 = \bar{x}\bar{y}, p_2 = \bar{x}\bar{z}, p_3 = \bar{y}z, p4 = y\bar{z}, p_5 = xz$ and $p_6 = xy$.

Constructing the Covering Matrix. To construct the covering matrix, we list the minterms ($m's$) as the rows and the prime implicants ($p's$) as the columns. The covering matrix for f is:

$$\begin{array}{ccccccc}
 & p_1 & p_2 & p_3 & p_4 & p_5 & p_6 \\
m_1 & 1 & 1 & 0 & 0 & 0 & 0 \\
m_2 & 1 & 0 & 1 & 0 & 0 & 0 \\
m_3 & 0 & 1 & 0 & 1 & 0 & 0 \\
m_4 & 0 & 0 & 1 & 0 & 1 & 0 \\
m_5 & 0 & 0 & 0 & 1 & 0 & 1 \\
m_6 & 0 & 0 & 0 & 0 & 1 & 1
\end{array}$$

The matrix element A_{ij} is 1 if p_j covers m_i, and 0 otherwise. For example, A_{11} and A_{21} are 1's because $p_1 = \bar{x}\bar{y}$ covers $m_1 = \bar{x}\bar{y}\bar{z}$ and $m_2 = \bar{x}\bar{y}z$. The goal of the two-level logic minimization is to cover all the minterms (the rows) with the least number of prime implicants (the columns). This is the familiar unate covering problem.

6.1.3 Two-Level Logic Minimizer – Espresso

Espresso [23] is a well-known two-level logic minimizer and represents a very efficient implementation of the Quine-McCluskey procedure. *Espresso* runs in two modes: heuristic and exact. In the heuristic mode, *espresso* does not generate the full covering table and only uses greedy heuristics to find a subset of the prime implicants to cover all the minterms. In this mode, *espresso* runs very fast but there is no guarantee how close the quality of the solution found is from the optimal value. In the exact mode, *espresso* is guaranteed to return the optimal representation of two-level logic but it may run for a long time in order to exhaust the large search space. Thus, it is often impractical to run *espresso* in the exact mode on large benchmark problems. From this point on, we refer to *espresso* running in exact mode as *espresso-exact* and *espresso* running in heuristic mode as *espresso-heur*.

The value of the exact mode is that it allows us to obtain optimal solutions to many small to medium size benchmarks. Only when we have knowledge of these optimal solutions, can we reliably assess the effectiveness of various heuristics for two-logic minimization. In both the heuristic and the exact modes, *espresso* uses a procedure called *mincov* to solve the unate covering problem. Next, we study how we can improve the performance of *espresso-exact* by replacing *mincov* with *eclipse-cp*.

6.1.4 Experimental Results

The covering procedure in *espresso-exact* is called *mincov*. We incorporate *eclipse-cp* into *espresso-exact* by swapping out *mincov* for the covering procedure. We call the new two-level logic minimizer *latte*. We ran both *espresso-exact* and *latte* on 134 standard two-level logic minimization benchmarks that are distributed with *espresso*.

The 134 benchmarks were categorized as *easy* and *hard* according to whether it is solvable by *espresso*. For the 114 easy instances, the covering procedure is not the bottleneck and both solvers can solve the instances in approximately the same time. For the hard instances, 14 of them were solved by *espresso-signature* in [76]. However, 6 hard instances remains unsolved: 3 of them are due to the fact that the covering problem couldn't be generated because the enormous number of prime implicants involved; the other 3 are unsolved because of the complexity of the covering procedure. We successfully solve the latter 3 benchmarks with *latte* and we report the results in Figure 6.1.

Table 6.1: Comparison of *espresso*-exact and *latte* on three previous unsolved benchmarks.

	espresso		*latte*	
benchmark	cover	total	cover	total
ex5	–	12hrs*	129.58	139.24
max1024	–	12hrs*	329.11	329.5
prom2	–	12hrs*	12.97	14.75

* times out at 12 hours.

This table compares *espresso*-exact with *latte* on three two-level logic minimization benchmarks. For each benchmark, the timeout value is 12 hours and the table reports the covering time (if the benchmark is solved) and the total execution time. On all three benchmarks, *espresso* times out after 12 hours of execution. However, *latte* can solve each benchmark within a few hundred seconds. By comparing the covering time and the total time, it is clear that the covering procedure is the bottleneck for *latte* to solve these benchmarks.

In Figure 6.1, for each benchmark, we report the covering time and the total time for *latte*. The solver *espresso-exact* times out on all the benchmarks after 12 hours of execution. However, *latte* is able to solve each of them within a few hundred seconds. By comparing the covering time and the total time, it is clear that the covering procedure is the bottleneck for *latte* to solve these benchmarks.

In Table 6.2, we compare the solution quality between *espresso-heur* and *latte*. We mentioned earlier in Section 6.1.1 that, in two-level logic minimization, the primary cost is the number of product terms and the secondary cost is the number of literals. *Latte*, like *espresso*-exact, guarantees the optimality of the number of product terms but not the number of literals. On *ex5* and *max1024*, we observe a gradual decrease in both product terms and literals from the original instance to the solution returned by *latte*. For example, on *max1024*, the original instance has 1024 product terms. The solution returned by *espresso-heur* has 274 product terms and the solution returned by *latte* only has 259 product terms. On *prom2*, the original formula already has the minimum number of product terms. Even though *latte* is able to reduce the number of literals, it gives a solution with more literals than *espresso-heur*.

Table 6.2: Comparison of solution quality between local-search and branch-and-bound methods on two-level logic minimization benchmarks

	orig		*espresso-heur*		*latte*	
benchmark	p. terms	literals	p. terms	literals	p. terms	literals
ex5	256	9668	74	1903	65	1193
max1024	1024	13472	274	2266	259	2207
prom2	287	5610	287	5526	287	5528

This table compares the solution quality between *espresso-heur* and *latte*. The table reports the original number of product terms and literals for each benchmark. It also reports the number of product terms and literals after doing logic minimization with *espresso-heur* and *latte*. The solver *latte* guarantees the optimality on the number of product terms, the primary cost, but not the optimality on the number of literals, the secondary cost. On *ex5* and *max1024*, we observe a gradual decrease in both product terms and literals. On *prom2*, *latte* actually gives a solution with more literals than *espresso-heur*.

6.1.5 Summary

We improved the state-of-the-art two-level logic minimizer, *espresso-exact*, by replacing its covering procedure with *eclipse-cp*. The new logic minimizer, *latte*, is able to solve some open problems by having a much more efficient implementation of the covering procedure. We also compared the solution quality between *latte* and *espresso-heur*. For benchmarks whose optima are previously unknown, *latte* found better solutions than *espresso-heur* in two of the three cases.

6.2 Group-Partial MaxSat and FPGA Routing

Advances in *Sat* solvers [15, 16, 17] have motivated researchers in physical design to recast FPGA detailed routing problems as *Sat* problems [9, 10, 11, 12]. Wood and Rutenbar represented the routing constraints as a single large Boolean equation using a binary decision diagram (BDD) [9]. This formulation considers all nets simultaneously: any assignment to the input variables that satisfies the Boolean equation also specifies a complete detailed routing. However, BDDs suffer from memory explosion when the FPGA detailed routing instances become large. To alleviate this problem, Nam *et al.* [10] presented a *Sat* formulation that is capable of handling very large FPGA instances. This formulation uses a *track-based* routing constraint model. Nam *et al.* [11] made further improvements by introducing the RCS formulation using a *route-based* routing constraint model. They showed that the RCS formulation yields an easier-to-solve and more scalable routability Boolean function. Their SAT-based router obtains second best results next to VPR [77] but outperforms all other conventional routers such as SEGA [78] and SROUTE [79]. It is worth noting that VPR has the advantage of doing both global and detailed routing while the SAT-based router only does detailed routing.

The SAT-based approaches above ask the question introduced in [10]: "Given an FPGA placement, is this layout routable?" The answer is yes or no: either a complete detailed routing solution is found or we get a proof that a routing does not exist.[1] This situation arises when for a layout with N nets, we cannot tell whether half of the nets cannot be routed or only one net cannot be routed – there is no middle ground. Xu et al. argue for a *partial solution* and ask a different question [13]: "... can we route this layout with not more than k nets unconnected?" In response to the question, they formulate a "subset satisfiable" Boolean *Sat* which clearly supports the concept of high quality partial solutions [13].

In this work, we ask the question that is equivalent to the question above: "What is the maximum number of nets that can be routed in this layout?" We answer this question directly by formulating it as a *group-partial Max-Sat* problem. In this section, we first present the RCS routing formulation and connect it with *group-partial Max-Sat*. We then present our local-search solver, *wpack*, for optimizing FPGA detailed routing. Experimental results that compare and contrast *sub_SAT*, *wpack* and *eclipse-lpr* on a set of FPGA routing benchmarks are then presented.

6.2.1 The RCS Formulation

The RCS formulation for the FPGA detailed routing problems distinguishes the routing constraints as either "liveness" constraints or "exclusivity" constraints [11]: the first ensures the routability of each net and the second ensures the exclusive usage of the tracks. For any complete or partial solution to make sense, *all* the exclusivity constraints have to be satisfied. Since a net can have multiple connections, each of which corresponds one-to-one to the liveness constraints, we group all such constraints for the same net together to represent its routability. A net is routed only when *all* of its liveness constraints are satisfied. The goal of maximizing the number of routable nets then requires us to *satisfy* all exclusivity constraints and *maximize* the number of nets whose group of liveness constraints are all satisfied. When using the RCS formulation of FPGA detailed routing, a multi-pin net is decomposed into a set of 2-pin connections prior to the transformation.

[1]In practice, we may get a third answer, due to the *Sat* solver timeout: 'unresolved'. The real possibility of such a scenario re-enforces the case for partial solutions.

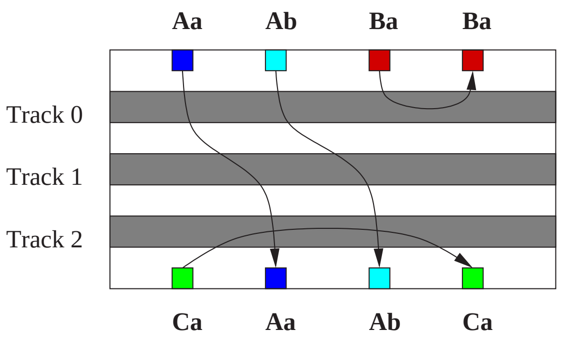
We show an example of RCS formulation in Figure 6.1. This is an unroutable example with 3 nets and 3 tracks. Net A has two 2-pin connections Aa and Ab; Net B and Net C each has one 2-pin connection. There are two types of clauses (constraints) in Figure 6.1:

1. **Liveness Clauses.** The four liveness clauses guarantee that each 2-pin connection has at least one route in the routing solution. A typical liveness clause consists of an OR of all route variables for a 2-pin connection. The number of route variables for a 2-pin connection is the same as the number of tracks available to the connection. In Figure 6.1, the first liveness clause $Aa0 \vee Aa1 \vee Aa2$ means that the 2-pin connection Aa is routed either via track 0 or track 1 or track 2. For a net with n 2-pin connections, n liveness clauses are required to ensure all such connections are routed. Net A in Figure 6.1 has two 2-pin connections: Aa and Ab. A net is not considered routed if any such connection is not routed. Therefore, the four liveness clauses in this example are in three *groups*: the first two are the group of liveness clauses for net A and the last two are in their individual groups for B and C.

2. **Exclusivity Clauses.** The exclusivity clauses guarantee that no two 2-pin connections are assigned to the same track. An exclusivity clause consists of two negative literals. For example, in Figure 6.1, the exclusivity clause $\overline{Aa0} \vee \overline{Ab0}$ means that Aa and Ab cannot be routed on track 0 at the same time. The group of six exclusivity clauses ensure that no two nets are routed on track 0. Similar clauses can be constructed for track 1 and 2.

This example is not routable due to the pigeon hole principle. In fact, at most two nets can be routed.

The problem of optimizing FPGA detailed routing can be viewed as a *group-partial Max-Sat* problem (first introduced in Section 2.1.5): the exclusivity clauses are the hard constraints; the groups of liveness clauses are the software constraints and each group is satisfied if and only if all the clauses in the group are satisfied. The traditional *Sat* approach is unable to solve this problem easily because it can



$$\left.\begin{array}{l} Aa0 + Aa1 + Aa2 \\ Ab0 + Ab1 + Ab2 \\ Ba0 + Ba1 + Ba2 \\ Ca0 + Ca1 + Ca2 \end{array}\right\} \begin{array}{c}\text{Liveness}\\ \text{clauses}\end{array}$$

$$\left.\begin{array}{l} \overline{Aa0} + \overline{Ab0} \\ \overline{Aa0} + \overline{Ba0} \\ \overline{Aa0} + \overline{Ca0} \\ \overline{Ab0} + \overline{Ba0} \\ \overline{Ab0} + \overline{Ca0} \\ \overline{Ba0} + \overline{Ca0} \end{array}\right\} \begin{array}{c}\text{Exclusivity clauses}\\ \text{for track 0}\end{array}$$

$$\left.\begin{array}{c} \cdot \\ \cdot \\ \cdot \end{array}\right\} \begin{array}{c}\text{Exclusivity clauses}\\ \text{for track 1, 2}\end{array}$$

Figure 6.1: A small unsatisfiable routing example (3 nets and 3 tracks) and its RCS formulation.

only tell whether *all* the nets can be routed or not. Xu et al. [13] took a novel *subset-satisfiability* approach to solve this problem.

6.2.2 The sub_SAT Approach

The subset-satisfiability approach, or *sub_SAT* for short, transforms the optimization problem into a sequence of *Sat* problems with a relaxation parameter k. Starting from $k = 1$, the original unsatisfiable *Sat* instance is transformed into a new *relaxed Sat* instance, which is satisfiable if and only if k nets cannot be routed in the original *Sat* formulation. If the relaxed *Sat* instance is still not satisfiable, then a new instance with $k = 2$ is generated from the original instance. The process continues until the relaxed *Sat* instance is satisfiable and then the current $k = opt$ becomes the minimum number of unroutable nets. The satisfiability checking part is done using one of the fastest *Sat* solvers - *zchaff* [16]. One obvious advantage of this approach is that it can leverage the highly optimized *Sat* solvers and their continuous advances. The *sub_SAT* idea was successfully applied to several standard FPGA routing benchmarks and found the minimum number of unroutable nets [13] (this is equivalent to finding the maximum number of routable nets).

However, the *sub_SAT* approach has a few possible drawbacks:

1. The *sub_SAT* approach relies on state-of-the-art branch-and-bound *Sat* solvers. Despite the tremendous improvements made recently, these solvers still don't scale well. However, the CNF formulae from today's FPGA routing benchmarks can contain tens of thousands of variables and clauses.

2. Instances generated with $k = 1, 2, ..., opt$ can be difficult themselves. The cost of finding the optimum has to take into account the cost of solving all the intermediate problems.

3. The formulation may suffer from significant size explosion when the relaxation parameter k is large. So the *sub_SAT* approach is suitable best for cases where the minimum number of unsatisfied constraints is small. One of the reasons why *sub_SAT* works well for the FPGA detailed routing benchmarks is that they are "almost" routable, which means the optimum number of unroutable nets is small.

To avoid these drawbacks, we decide to tackle the routing problem directly with a local-search algorithm.

6.2.3 The Local-Search Approach

Our local-search algorithm *wpack* works on the original RCS formulation and doesn't generated any intermediate problems. We present *wpack* in Figure 6.2.

In the *wpack* algorithm, on lines 3–4, a choice is made *globally* to flip a variable to increase the number of routed nets the most. On lines 5–8, a choice is made *locally* within an unrouted net with the least number of unsatisfied liveness clauses. This is an attempt to route an "almost routable" net. Line 9 changes the value of the chosen variable and lines 10–12 updates the best solution if appropriate.

```
Algorithm: wpack
input: a cnf formual F representing a FPGA detailed routing instance
output: the minimum number of nets that cannot be routed
method:
 1 randomly initialize the variable assignments
 2 while termination criterion not met do
 3     with probability $p$, choose a variable not in the tabu list that
 4         increases the number of routed nets the most
 5     with probability $1 - p$, randomly choose an unsatisfied liveness
 6         clause for a net with the smallest group of unsatisfied liveness
 7         clauses and choose a variable in the clause but not in the tabu list
 8         which increases the number of routed nets the most
 9     flip the variable
10     if all exclusivity constraints are satisfied
11          and current-unrouted-nets < unrouted-nets$_{opt}$
12     then unrouted-nets$_{opt}$ = current-unrouted-nets
13     endif
14 end do
15 return unrouted-nets$_{opt}$
```

Figure 6.2: The *wpack* algorithm for optimizing FPGA detailed routing.

6.2.4 Experimental Results

The FPGA detailed routing optimization problem posts a great challenge to branch-and-bound *MinCostSat* and *IP* solvers. As we have seen in Table 2.5, *cplex* times out on 8 out of 10 benchmarks. In this section, we evaluate the performances of two local-search approaches: *wpack* (the solver specifically designed for FPGA detailed routing) and *eclipse-stoc* (the general-purpose local-search *MinCostSat* solver). All experiments ran on a Pentium IV@1.8Ghz with 256MB of RAM under Linux.

We first show our experiments under termination criterion A in Table 6.3. Recall that under this termination criterion, a local search solver is terminated when either a solution with best known cost has been discovered or when the runtime exceeds

Table 6.3: Runtime comparisons between *sub_SAT*, *wpack* and *eclipse-stoc* on FPGA detailed routing benchmarks.

benchmark	nets	opt	*sub_SAT*	*wpack*	*eclipse-stoc*
term1_gr_rcs_w3	88	7	1.1	6.6	7.3
apex7_gr_rcs_w4	126	2	0.1	0.3	32.2
9symml_gr_rcs_w5	79	1	0.1	1.6	123.9
c499_gr_rcs_w5	115	3	15.6	3600^*	3600^*
example2_gr_rcs_w5	205	2	20.5	11.2	3600^*
too_large_gr_rcs_w6	186	$\leq 3^\diamond$	3649.5	3600^*	3600^*
alu2_gr_rcs_w7	153	1	20.4	135.0	3600^*
c880_gr_rcs_w6	234	$\leq 4^\diamond$	6067.1	272.1	3600^*
vda_gr_rcs_w7	225	$\leq 13^\diamond$	21605.1	11.3	3600^*
k2fix_gr_rcs_w9	404	$\leq 11^\diamond$	16008.8	401.0	3600^*

$*$ solver times out at 3600 seconds.

$\diamond$ Some intermediate *sub_SAT* runs time out at 1800 seconds so the optimum is unknown.

This table compares *sub_SAT* with *wpack* and *eclipse-stoc*. In last two columns, we report the mean runtime for *wpack* and *eclipse-stoc*. Our solver *wpack* is capable of finding the best known solutions on all benchmarks except *c499_gr_rcs_w5* and *too_large_gr_rcs_w6*. The general-purpose local-search *MinCostSat* solver *eclipse-stoc* is not effective on this set of benchmarks.

a fixed timeout value, in this case, 3600 seconds. The second column shows the number of nets each benchmark contains. The third column shows the minimum number of unroutable nets (for four larger benchmarks, the optima are not known and we show the best values found by *sub_SAT* [13]). The last three columns show the runtime of *sub_SAT*, *wpack* and *eclipse-stoc*, respectively. For *wpack* and *eclipse-stoc*, we report the average of 32 runs each with a different initial variable assignments. Our solver *wpack* is effective in find best known solutions on all benchmarks except *c499_gr_rcs_w5* and *too_large_gr_rcs_w6*. The general-purpose BCP solver *eclipse-stoc* is not effective on this set of benchmarks.

In Table 6.4, we focus on the benchmarks whose minimum number of unroutable nets is unknown. For each benchmark, we only launch *wpack* once. We allow *wpack* to run for the same amount of time *sub_SAT* ran and return the best value it finds. For three out of the four benchmarks, we are able to improve upon the previous best solution, e.g., for the largest benchmark *k2fix_gr_rcs_w9*, *wpack* is able to find a solution with only 6 unrouted nets while the previous best solution has 11 unrouted nets. However, for *too_large_gr_rcs_w6*, *wpack* still cannot find a solution with the previously best known cost 3.

Table 6.4: Solution quality comparisons between *sub_SAT* and *wpack* on benchmarks whose optima are unknown.

	sub_SAT		*wpack*	
benchmark	best	time	best	time
too_large_gr_rcs_w6	3	3649.5	4	3649.5
vda_gr_rcs_w7	13	21605.1	7	21605.1
C880_gr_rcs_w6	4	6067.1	3	6067.1
k2fix_gr_rcs_w9	11	16008.8	6	16008.8

For the benchmarks whose optimum is unknown, we run *wpack* for the same time as *sub_SAT*. On three out of the four instances, *wpack* finds better quality solutions.

6.2.5 Summary

In this section, we first introduced the RCS formulation for FPGA detailed routing. We then presented the *sub_SAT* approach and its drawbacks. To address these drawbacks, we designed the *wpack* algorithm based on local search. We compared *sub_SAT*, *wpack* and *eclipse-stoc* on a set of FPGA detailed routing benchmarks. *Eclipse-stoc* is in general not competitive. *Wpack* is effectively in finding the best known solutions in most cases and is able to discover new best known solutions for large instances whose optimum is unknown.

6.3 MaxSat

Introduced in Chapter 2, *Max-Sat* is a combinatorial optimization problem that can be formulated as *MinCostSat* by introducing slack variables. *Max-Sat* is NP-complete, even when each clause contains only two literals (*Max-2-Sat*). It can be solved using either branch-and-bound algorithms [45, 46, 47, 48, 49] or local-search algorithms [50, 51, 52]. Branch-and-bound algorithms guarantee optimality and use the algorithm based on the Davis-Putnam-Logemann-Loveland (DPLL) procedure [80]. Local-search algorithms use variations of greedy search and consequently do not guarantee optimality. In this work, we study branch-and-bound algorithms for *Max-Sat* (local-search algorithms are only used to obtain an initial upper-bound). Three approaches for *Max-Sat* include: (1) the branch-and-bound approach under the original formulation, (2) the *IP* approach, and (3) the *MinCostSat* approach.

We first present an improved branch-and-bound *Max-Sat* solver, *qtmax* [60], that implements state-of-the-art lower-bounding techniques, search pruning techniques and variable selection heuristics. Our experimental results show that *qtmax* is competitive with state-of-the-art *Max-Sat* solvers and outperforms them on many benchmarks. We then study the three branch-and-bound algorithms for *Max-Sat* by comparing the performances of *qtmax*, *cplex* and *eclipse-lpr*[2] on two series of *Max-2-Sat* and *Max-3-Sat* benchmarks.

6.3.1 The Branch-and-Bound Algorithm – qtmax

Previous *Max-Sat* solvers include those of Wallace and Freuder [45], Borchers and Furman [46], and Alsinet et al. [47, 48]. Zhang et al. [49] have recently proposed an efficient algorithm for the special case of *Max-2-Sat*, where it performs considerably better than those in [45, 46, 47, 48], but our interest is in the more general problem. The *qtmax* algorithm is based on the DPLL procedure depicted in Figure 6.3. For convenience we think in terms of minimizing the number of unsatisfied clauses. Let

[2]We use *eclipse-lpr* instead of *eclipse-cp* because *eclipse-cp* is not competitive with *eclipse-lpr* for the BCP instances derived from *Max-Sat*.

```
function MaxSat-BB($F$: clause set, current-unsat: integer)
    Search-Tree-Pruning($F$, current_unsat)
    if Lower-Bound($F$, current-unsat) $\geq$ upper-bound
    then return upper-bound
    else if Lower-Bound($F$, current-unsat) $=$ upper-bound $-1$
        then Unit-Propagation($F$, current-unsat)
    if $F$ contains non-empty clauses and
       Lower-Bound($F$, current-unsat) $<$ upper-bound
    then $x =$ Select-Splitting-Variable($F$)
         return $min$(MaxSat-BB($F[x]$, current-unsat + $|\{\overline{x}\}|$),
                    MaxSat-BB($F[\overline{x}]$, current-unsat + $|\{x\}|$));
    if current-unsat $<$ upper-bound then upper-bound $=$ current-unsat
    return upper-bound
end function
```

Figure 6.3: A recursive DPLL-based *Max-Sat* algorithm.

$F[x]$ denote the set of clauses that remain after assigning $x =$ true, i.e. removing all clauses containing x and removing $\overline{x}$ from any clauses that contain it. We use $\{x\}$ to denote a unit clause containing x and $|\{x\}|$ for the number of such unit clauses. Initially called with current-unsat $= 0$ and upper-bound $= \infty$, the algorithm in Figure 6.3 solves the *Max-Sat* problem recursively.

Similar to a branch-and-bound *MinCostSat* solver, the performance of a branch-and-bound *Max-Sat* solver depends on (1) the lower-bounding technique, (2) the search-tree pruning technique, (3) the ability to obtain a good initial upper-bound and (4) the branching variable selection heuristic.

Lower-Bounding Techniques. An improvement on simply using current-unsat as the lower-bound is introduced in [45] and implemented by Alsinet, et al. [47]. It is based on the observation that in addition to the currently unsatisfied clauses, there can be unit clauses that contain opposite literals. Such literals will result in empty (unsatisfied) clauses when they are assigned, whether true or false. For example, if the unit clauses contain $\{x_1\}, \{\overline{x}_1\}, \{x_2\}$, $\{\overline{x}_2\}$ and $\{\overline{x}_2\}$, then at least two more unsatisfied clauses will be generated when x_1 and x_2 are assigned. The lower-bound can therefore be increased by $\sum_{i=1}^{n} \min(|\{x_i\}|, |\{\overline{x}_i\}|)$.

Search-Tree Pruning Techniques. The most obvious way to prune the search-tree is unit propagation. When there are unit clauses and the lower-bound (LB) is within 1 of the upper-bound (UB), a branch containing an unsatisfied unit clause need not be explored — that clause will make LB $\geq$ UB — and unit clauses can be assumed to be true and propagated.

To further reduce the search-tree size, we use two techniques. The *dominating unit-clause rule* [45] is based on the following observation: if $|\{x\}|$ is greater than the number of $\overline{x}$ occurrences in F, then the branch in which $x =$ false does not have to be explored — the best solution in that branch can be no better than one in which $x =$ true.

The *near upper-bound rule* [48] notes that if $|\{x\}| > |\{\overline{x}\}|$ and $\text{LB} + \text{abs}(|\{x\}| - |\{\overline{x}\}|) \geq \text{UB}$, there is no need to set $x =$ false: doing so will generate $|\{x\}|$ unsatisfied clauses and make current-unsat $>$ UB. The solver *qtmax* uses both pruning rules and the resulting improvements on selected benchmarks are shown in Table 6.5.

Obtaining a Good Upper-Bound. Instead of initializing UB $= \infty$, *Max-Sat* solvers often obtain a better upper-bound by running a local search. For example, Borchers and Furman's solver (*maxsat*) has two phases [46]. The first phase runs *gsat* [73] (a local-search algorithm for SAT) for a fixed number of tries, each try consisting of a fixed number of flips. The second phase is a branch-and-bound with UB initialized to the best solution found in the first phase. Our solver *qtmax* uses the same idea. Unlike in *eclipse*, we do not do local search at each node because *qtmax* is very effective in finding the optimal UB in the first phase.

Branching Variable Selection Heuristics. Branching variable selection heuristics are the subject of intensive research in the *Sat* community. These heuristics are very effective in reducing the size of the search-tree. Two of the heuristics are used for *Max-Sat*:

1. MOMS (see, e.g., [46]): Choose the variable that appears most often in clauses with minimum size. When there are unit clauses, such a scheme clearly favors

Table 6.5: Backtrack comparisons for three variants of *qtmax* on random *Max-2-Sat* and *Max-3-Sat* (reference instances only).

benchmark	unsat$_{opt}$	no *duc* rule	no *nub* rule	*qtmax*
2sat_v50_c100	4	434	242	236
2sat_v50_c150	8	873	411	299
2sat_v50_c200	16	16762	10038	5893
2sat_v50_c250	22	14069	11138	6650
2sat_v50_c300	32	149741	112161	68916
2sat_v50_c350	41	333100	261687	153763
2sat_v50_c400	45	135852	136737	78517
3sat_v50_c250	2	258	269	258
3sat_v50_c300	3	9137	10550	9099
3sat_v50_c350	8	142436	334077	141523
3sat_v50_c400	11	390924	943062	385749

This table compares three variations of *qtmax* on a set of *Max-2-Sat* and *Max-3-Sat* reference instances. The third column (no *duc* rule) represents the variation of *qtmax* that doesn't implement the *dominating unit-clause* rule. The next column (no *nub* rule) represents the variation of *qtmax* that doesn't implement the *near upper-bound* rule. The last column represents the variation of *qtmax* that implements both rules. It is clear that for all the benchmarks in this table, the *qtmax* that implements both *dominating unit-clause* rule and *near upper-bound* rule uses the least amount of backtracks.

the variables appearing often in the unit clauses. The MOMS heuristic works well on benchmarks with a small optimum number of unsatisfied clauses [47].

2. Jeroslow-Wang (JW) [81]: For each literal L in F, the JW heuristic calculates the function $J(L) = \sum_{L \in C} 2^{-|C|}$, where $|C|$ is the length of clause C and chooses the variable that maximizes $J(x) + J(\overline{x})$. Clearly, $J(x) + J(\overline{x})$ increases when $|C|$ decreases and the occurrences of x and $\overline{x}$ increase. Intuitively, this favors a variable that appears often in short clauses.

The branching variable selection heuristic in *qtmax* takes into account the value of UB. If UB is less than a given threshold, our heuristic is identical to MOMS; otherwise, we use a variant of MOMS: choose the variable with the most occurrences in the shortest *non-unit* clauses. In the latter case, we exclude unit clauses because we have already exploited them extensively in both our lower-bounding and search pruning techniques. Let v_m be the variable that occurs the most in non-unit clauses. The variable v_m is a good candidate for the splitting variable because (assuming $v_m = \text{true}$; a symmetric argument applies when $v_m = \text{false}$): (1) all the clauses containing v_m are eliminated from the formula, reducing the size of the formula significantly, and (2) $\overline{v}_m$ is removed from all the clauses containing it, potentially introducing new unit clauses that will raise the lower-bound.

Combining all the techniques we discussed so far, we implemented our branch-and-bound *Max-Sat* solver *qtmax*. Next, we compare *qtmax* with other state-of-the-art *Max-Sat* solvers.

6.3.2 Experimental Results with qtmax

We report on the performance of four *Max-Sat* solvers: *maxsat* as described in [46], *LB2+JW* and *LB2+MOMS* as described in [47, 48], and *qtmax* as described in this work. The class labels beginning with 2sat and 3sat denote classes of random instances, introduced in [46]. The numbers of variables and clauses in these instances are also contained in the labels. The minimum number of clauses that cannot be satisfied is known and is shown as $\text{unsat}_{\texttt{opt}}$ for each instance. The label steiner_a0027 denotes an instance derived from Steiner set-covering instances. The instance bw_large_a_u represents a well-known unsatisfiable block-world benchmark [82].

6.3.2.1 Comparative Results

Tables 6.6 and 6.7 report on the mean of backtracks and runtime[3] (in seconds) for each PC[4] class of size 32 of the randomly generated benchmarks. For all *Max-2-Sat* benchmarks, *qtmax* has the least number of mean backtracks among four solvers. For *Max-3-Sat* benchmarks, the results are less conclusive even though *qtmax* still has the least overall. It is interesting to note that the least number of backtracks doesn't always correspond to the shortest runtime.

Tables 6.8 and 6.9 provide a more thorough view of the backtrack and runtime data for selected *Max-2-Sat* and *Max-3-Sat* benchmarks as well as two structured benchmarks. It reports both the mean and standard deviation of backtracks and runtime for the entire population of instances in the class — as well as the observed distribution for each variable, based on the outcome of the χ^2-test ($\alpha = 0.05$).

Following the convention introduced in [58], we recognize the following distribution classes: normal (N), exponential (E), near-normal(nN), near-exponential (nE) and heavy-tailed (hT). In addition, the column labelled as *initV* reports on the backtrack and runtime values observed for the *reference instance*. When we observe the exponential distribution, the size of standard deviation is close to or identical to value of the mean; hence the difference between *initV* and *meanV* can be significant (as for bw_large_a_u).

[3]All experiment in Table 6.6–6.9 are done on a Pentium II@266Mhz with 196MB of RAM under Linux.

[4]The difference between PC-classes and P-classes is that PC-class allows complementation of the literals. For *Max-Sat*, such an operation doesn't change the minimum number of unsatisfiable clauses.

Table 6.6: Average performance comparisons of *maxsat*, *LB2+MOMS*, *LB2+JW* and *qtmax* on PC-classes of size 32 from *Max-2-Sat* benchmarks.

		maxsat		*LB2+MOMS*		*LB2+JW*		*qtmax*	
benchmark	unsat_{opt}	backtrack	time	backtrack	time	backtrack	time	backtrack	time
2sat_v050_c100	4	670	0.17	513	0.18	624	0.19	231	0.46
2sat_v050_c150	8	14,839	0.48	8,809	0.58	1,226	0.27	312	0.50
2sat_v050_c200	16	1,324,875	28.8	221,123	12.7	40,420	3.76	5,953	1.13
2sat_v050_c250	22	6,723,974	180	339,118	24.8	32,294	3.92	6,528	1.42
2sat_v050_c300	32	–	1800*	1,386,415	134	431,048	58.7	71,999	11.5
2sat_v050_c350	41	–	1800*	–	1800*	1,098,454	175	156,318	26.2
2sat_v050_c400	45	–	1800*	–	1800*	362,224	72.5	81,576	18.1
2sat_v100_c200	5	16,573	0.96	14,407	1.33	12,438	1.92	3,910	1.51
2sat_v100_c300	15	–	1800*	–	1800*	1,517,643	244	390,056	61.7
2sat_v100_c400	29	–	1800*	–	1800*	–	1800*	–	1800*
2sat_v150_c300	4	9,606	1.28	8,797	1.60	56,242	10.6	2,948	2.6

* solver times out at 1800 seconds.

This table compares four solvers' performance on PC-classes of size 32 from *Max-2-Sat* instances. The mean and standard deviation of runtime and backtracks are reported. For all benchmarks, *qtmax* consistently has the least average number of backtracks. The solver *LB2+JW* is the most competitive with *qtmax* on both backtracks and runtime; however, in most cases, it is more than three times as slow as *qtmax*.

Table 6.7: Average performance comparisons of *maxsat*, *LB2+MOMS*, *LB2+JW* and *qtmax* on PC-classes of size 32 from *Max-3-Sat* benchmarks.

		maxsat		*LB2+MOMS*		*LB2+JW*		*qtmax*	
benchmark	unsat$_{opt}$	backtrack	time	backtrack	time	backtrack	time	backtrack	time
3sat_v50_c250	2	365	0.29	333	0.27	3775	0.63	332	0.77
3sat_v50_c300	3	11,533	0.94	9,302	1.06	17,953	2.72	8,189	2.07
3sat_v50_c350	8	929,361	41.7	530,127	44.2	185,353	29.6	142,781	28.7
3sat_v50_c400	11	6,001,288	272	2,382,077	220	581,519	104	386,116	84.1
3sat_v50_c450	15	34,925,193	1,666	7,585,480	797	1,178,399	239	676,750	156
3sat_v50_c500	15	20,416,608	1,046	4,092,190	457	662,574	154	385,765	99.6
3sat_v100_c500	4	506,968	56.9	460,148	70.3	–	1800*	584,698	145.13
3sat_v150_c675	2	240,554	49.8	233,878	57.1	–	1800*	252,662	96.9

* solver times out at 1800 seconds.

This table compares four solvers' performance on PC-classes of size 32 from *Max-3-Sat* instances. The mean and standard deviation of runtime and backtracks are reported. For backtracks, *qtmax* has the least amount except for *3sat_v100_c500* and *3sat_v150_c675*. On these two benchmarks, *LB2+MOMS* has the least number of backtracks and also runs faster than *qtmax*. But it runs significantly slower than *qtmax* on other benchmarks.

Table 6.8: Backtrack and runtime comparisons for *maxsat*, *LB2+MOMS*, *LB2+JW* and *qtmax* on PC-classes of size 32 from two random *Max-Sat* instances.

Class Name = 2sat_v050_c250_PC, Size = 32, unsat$_{opt}$ = 22

	backtrack			time			
solverID	initV	meanV	stDev	initV	meanV	stDev	distributions
qtmax	6,405	6,528	951	1.39	1.42	0.12	N, N
LB2+JW	30,013	32,294	1,705	3.67	3.92	0.18	N, N
LB2+MOMS	333,843	339,118	31,103	24.1	24.8	2.26	N, N
maxsat	6,627,660	6,723,974	648,599	177.0	180	17.1	N, N

Class Name = 3sat_v050_c450_PC, Size = 32, unsat$_{opt}$ = 15

	backtrack			time			
solverID	initV	meanV	stDev	initV	meanV	stDev	distributions
qtmax	653,757	676,750	36,204	150	156	8.11	nN, N
LB2+JW	1,169,907	1,178,399	121,939	237	239	23.6	nN, nN
LB2+MOMS	7,417,235	7,585,480	170,382	781	797	17.8	nN, nN
maxsat	34,246,168	34,925,193	1,069,538	1,634	1,666	51.0	nN, nN

Table 6.9: Backtrack and runtime comparisons for *maxsat*, *LB2+MOMS*, *LB2+JW* and *qtmax* on PC-classes of size 32 from two structured *Max-Sat* instances.

Class Name = steiner_a0027_PC, Size = 32, unsat$_{opt}$ = 8

	backtrack			time			
solverID	initV	meanV	stDev	initV	meanV	stDev	distributions
qtmax	19,551	19,311	117	1.52	1.61	0.02	N, N
LB2+JW	161,155	162,748	725	5.50	5.65	0.04	nN, N
LB2+MOMS	340,484	348,304	4,605	8.43	8.72	0.12	N, N
maxsat	1,573,453	1,679,637	41,621	23.3	25.0	0.59	N, nN

Class Name = bw_large_a_u_PC, Size = 32, unsat$_{opt}$ = 4

	backtrack			time			
solverID	initV	meanV	stDev	initV	meanV	stDev	distributions
qtmax	141,639	240,822	254,264	128	254	242	nE, nE
LB2+MOMS	170,079	313,598	252,028	126	304	245	E, E
maxsat	322,793	541,858	422,741	112	261	202	E, E
LB2+JW	*	*	*	*	*	*	–

$*$ The solver times out at 1800 seconds.

6.3.3 Solving MaxSat as IP and MinCostSat

Joy et al. [62] introduced the *IP* formulation for *Max-Sat* and compared their branch-and-cut algorithm with EDPL (a DPLL based *Max-Sat* solver) [83]. Their experiments show that the branch-and-cut algorithm outperforms EDPL on most *Max-2-Sat* benchmarks but is not competitive on *Max-3-Sat* benchmarks. Borchers et al. introduced a new state-of-the-art branch-and-bound solver *maxsat* in [46] and compared it with the *IP* solver *minto* [84]. They concluded that *minto* is faster on *Max-2-Sat* benchmark whereas *maxsat* is superior on *Max-3-Sat* instances.

In this work, we renew their study and compare the branch-and-bound approach with the *IP* approach for solving *Max-Sat*. In order to have meaningful results, it is crucial that we only use the leading solvers for both approaches. Therefore, we compare *qtmax* (the most efficient branch-and-bound *Max-Sat* solver) with *cplex*(the industrial leader in *IP* solvers). We also add *eclipse-lpr* to the competition to see whether the *MinCostSat* approach can be competitive. In Table 6.10, we compare the performance of *qtmax*, *cplex* and *eclipse-lpr* on two series of *Max-2-Sat* and *Max-3-Sat* benchmarks.

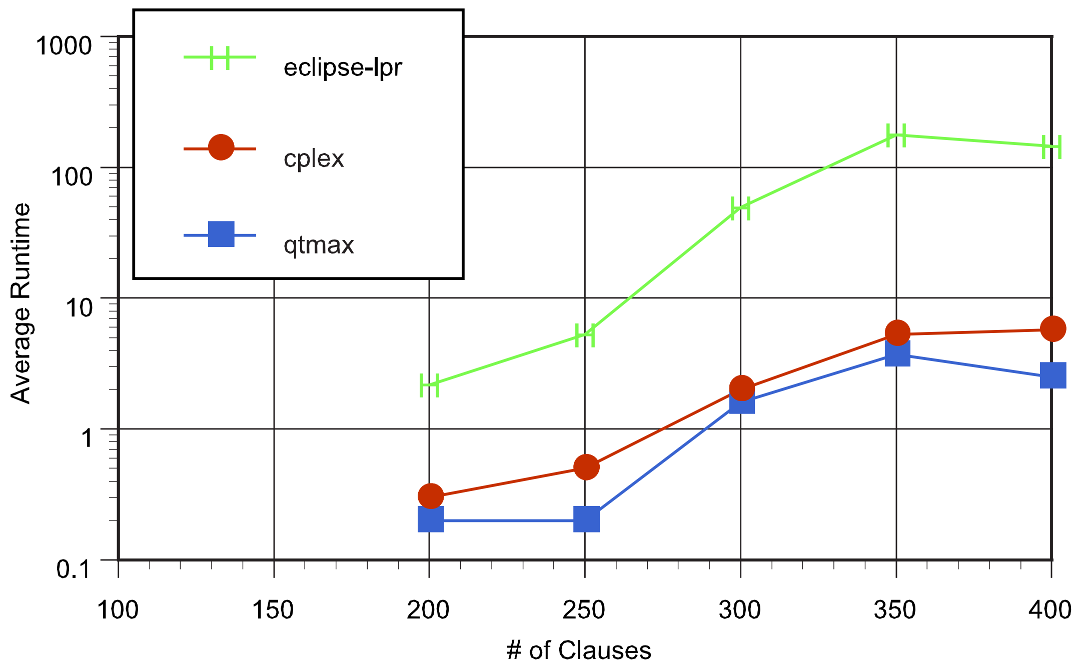
On the PC-classes of all the benchmarks in Table 6.10, *qtmax* has the best runtime performance. For example, on *2sat_v050_c400*, *qtmax*, *cplex* and *eclipse-lpr* have mean runtime of 2.5 seconds, 5.7 seconds and 144.5 seconds, respectively. On *Max-3-Sat* instances, the differences between *qtmax* and *cplex* are even more dramatic, e.g., for *3sat_v050_c450*, *qtmax* and *cplex* have mean runtime of 21.8 seconds, 606.3 seconds respectively. Overall, qtmax is able to outperform both *cplex* and *eclipse-lpr* on not only all the *Max-3-Sat* benchmarks but also all the *Max-2-Sat* benchmarks. This is in contrast with the conclusions drew in [46] and [62]. However, as we have seen, the data in Table 6.10 does indicate that *qtmax* is more dominant on *Max-3-Sat* instances than on *Max-2-Sat* instances. This means that if *qtmax* were less efficient, we may have observed the same dominance relationships as in [46] and [62]. Not surprisingly, *eclipse-lpr* is not competitive with *qtmax* because the *MinCostSat* benchmarks from *Max-Sat* don't represent covering problems.

In Figure 6.4, we plot the mean runtime and the number of nodes explored for the

Table 6.10: Results for *qtmax*, *cplex* and *eclipse* on PC-classes of size 32 from random *Max-2-Sat* and *Max-3-Sat* benchmarks.

benchmark	measure	*qtmax*	*cplex*	*eclipse-lpr*
2sat_v050_c200	time	0.2/0.01	0.3/0	2.2/0.2
	nodes	5953/552	165/44	104/12
2sat_v050_c250	time	0.2/0.01	0.5/0.1	5.3/0.3
	nodes	6528/951	261/70	233/11
2sat_v050_c300	time	1.6/0.2	2.0/0.5	49.1/1.8
	nodes	71999/7235	1499/459	1644/38
2sat_v050_c350	time	3.7/0.2	5.3/1.0	177.26/15.3
	nodes	156832/9664	4121/980	4074/233
2sat_v050_c400	time	2.5/0.5	5.7/0.8	144.5/13.3
	nodes	81576/17241	3367/548	3288/233
3sat_v050_c250	time	0.1/0.01	4.8/2.0	2.8/0.6
	nodes	332/296	3054/1508	111/26
3sat_v050_c300	time	0.3/0.01	18.0/7.0	20.8/5.5
	nodes	8189/236	11693/5162	792/207
3sat_v050_c350	time	4.0/0.1	123.9/53.2	201.2/39.9
	nodes	142781/4436	80386/38070	7045/927
3sat_v050_c400	time	11.8/0.5	364.7/133.8	1805.45/488.4
	nodes	386116/14865	208262/76581	22098/3284
3sat_v050_c450	time	21.8/1.1	606.3/336.2	5237.6/289.3
	nodes	676750/36204	312902/179101	23442/2255

This table presents a comparison of *qtmax*, *cplex* and *eclipse-lpr* on PC-classes of size 32 from random *Max-2-Sat* and random *Max-3-Sat* benchmarks. The mean and standard deviation of runtime and nodes are reported. Even though *eclipse-lpr* and *cplex* explore much fewer nodes than *qtmax*, *qtmax* clearly dominates both *cplex* and *eclipse-lpr* in runtime on all the *Max-2-Sat* and *Max-3-Sat* benchmarks. This is in contrast with the conclusions drew in [62] and [46]. However, the superiority of *qtmax* is much more obvious on the *Max-3-Sat* benchmarks on which it is about 30 times faster than *cplex*. For *Max-2-Sat* benchmarks,*qtmax* is only about 3 times faster than *cplex*.



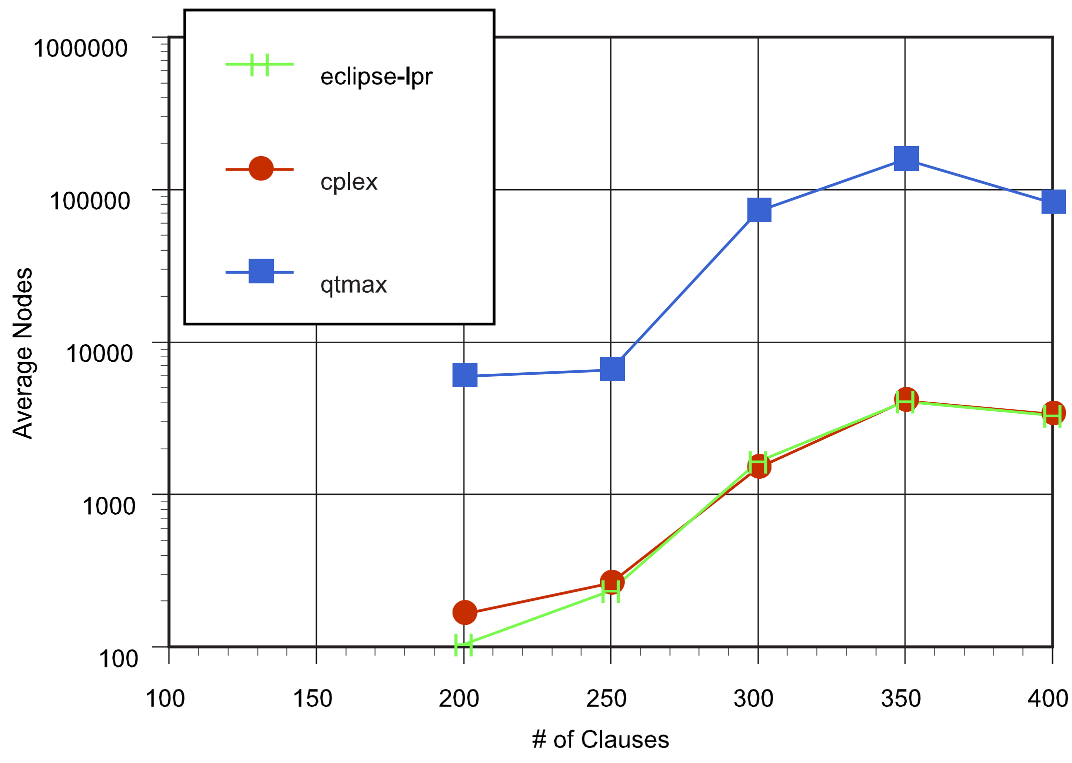


Figure 6.4: Asymptotic behavior of three solvers' runtime and nodes for the *Max-2-Sat* benchmarks.

three solvers on the *Max-2-Sat* series. All the instances in the series have 50 variables and the number of clauses ranges from 200 to 400 with increments of 50. The figure indicates that even though *qtmax* explores the most number of nodes (in the bottom figure), it still has the shortest runtime (in the top figure).

6.3.4 Summary

In this section, we first introduced *qtmax*, a new branch-and-bound *Max-Sat* solver. We then compared it with three state-of-the-art *Max-Sat* solvers on both randomly generated benchmarks and two structured ones. Overall, *qtmax* has the best performance in runtime and the number of nodes explored. *IP* solvers have been shown to be more efficient than branch-and-bound *Max-Sat* solvers on *Max-2-Sat* benchmarks but not as efficient on *Max-3-Sat* benchmarks. The introduction of *qtmax* changed this dominance relationship. After comparing *qtmax* with *cplex* and *eclipse-lpr*, we conclude that *qtmax* is the fastest solver for all the *Max-2-Sat* and *Max-3-Sat* benchmarks we considered.

6.4 Minimum-size Test Pattern Problem

Test pattern generation is the process of finding the input sequences that cause the defect to manifest itself at the output of the circuit. The most popular fault model is the *stuck-at* (stuck-at-0, stuck-at-1) fault model. A stuck-at fault occurs when a connection is permanently stuck at one logic value. A *test* for a stuck-at fault in a combinational circuit is simply an assignment of 0's and 1's to the primary inputs of the circuit that causes different outputs in the good and the faulty circuit. In the presence of incompletely specified primary input assignments, the minimum-size test pattern problem is to find a test with the minimum number of *specified* primary inputs (the other primary inputs are *don't cares*).

6.4.1 The BCP Formulation

When the minimum-size test pattern problem is formulated as a CNF formula F [85], two variables x^0 and x^1 are generated for each primary input x. If both x^0 and x^1 are 0, then the primary input x is a *don't care*. When x^0 is 1 (0) and x^1 is 0 (1), then the primary input x has to be 0 (1) in order to be a valid test. It is illegal to have $x^0 = 1$ and $x^1 = 1$. This combination is excluded by adding the clause $\overline{x^0} + \overline{x^1}$ to the CNF formula. The minimum-size test pattern problem is to:

$$\begin{gathered}\text{minimize} \sum_{x \in PI} (x^0 + x^1) \\ \text{subject to F}\end{gathered} \tag{6.2}$$

All the primary inputs variables have unit weight and the other variables have zero weight. The minimum-size test pattern problem represents a non-covering *MinCost-Sat* problem.

6.4.2 Experimental Results

The solver *bsolo* is the leading solver for the `ATPG` benchmarks. Other *MinCostSat* solvers including *eclipse* are not competitive with *bsolo* because they do not exploit the primary-input variables. In addition, as the data in Table 4.2 indicates, the lower-bounding techniques perform poorly on these benchmarks. Therefore, we take a different approach that explores the nodes as fast as possible and uses two very basic lower-bounding techniques: (1) the solver backtracks when the cost function above exceeds or is equal to the upper-bound, and (2) the solver backtracks when there is a conflict in the clauses.

We call this new non-covering solver *eclipse-bf* (*bf* stands for brute force). It branches on the weighted variables first (in the context of `ATPG` benchmarks, the weighted variables correspond to the primary inputs) because their assignments largely determine the values the rest of the variables. In Table 6.11, we compare *eclipse-bf* with *bsolo* on the P-classes of size 16 for five difficult benchmarks from [41]. For each benchmark, each solver runs 17 times, once on the reference benchmark and 16 times on the P-class instances. For each benchmark, the table reports the number of primary-input variables and the optimal cost. For each solver, the table reports the number of runs completed successfully within the 3000-second timeout. It also reports the mean and standard deviation of the runtime and nodes for the *successful* runs.

On all benchmarks except *misex3_Fb@1*, *bsolo* could not finish all 17 runs. For the bottom three benchmarks in Table 6.11, *bsolo* was only able to complete successfully on a small fraction of the 17 runs. For example, on *c6288_F69gat@1*, *bsolo* solves 2 out of 17 instances. However, *eclipse-bf* finishes 17 runs for all benchmarks except *c3540_F20@1*. The number of primary-input variables in *c3540_F20@1* is the most among all benchmarks. Since *eclipse-bf* chose to branch on these variables first, it faces a much larger search space on *c3540_F20@1* and this explains why it fails to solve any of the 17 instances. For *misex3_Fb@1* where both solvers are able to solve all 17 instances, *bsolo* has a mean runtime of 27.2 seconds and *eclipse-bf* runs slower with a mean runtime of 43.4 seconds. In addition, *eclipse-bf* explores significantly more nodes than *bsolo*.

6.4.3 Summary

We presented a branch-and-bound solver for minimum-size test pattern problem. Our solver *eclipse-bf* abandons techniques that are only effective on the covering problems, such as reduction and advanced lower-bounding. It attempts to explore the nodes as fast as possible and backtracks when either the current cost is greater than the upper bound or a conflict has occurred. We compared *eclipse-bf* with *bsolo* on P-classes of size 16 for five difficult benchmarks from [41]. The results show that *eclipse-bf* can solve all 17 instances for four out of the five benchmarks whereas *bsolo* can only do so for one benchmark. However, *eclipse-bf* fails to solve any instance for c3540_F20@1, which has the most primary-input variables among the five benchmarks we considered. It remains an open question whether advanced *Sat* techniques can be effectively incorporated into *eclipse-bf* and enable it to consistently solve benchmarks with large number of primary-input variables.

Table 6.11: Comparisons between *bsolo* and *eclipse-bf* on P-classes of size 16 of the ATPG benchmarks.

			bsolo			eclipse-bf		
benchmark	PI	opt	completed	nodes	time	completed	nodes	time
c432_F37gat@1	72	9	12	23498/10092*	*	17	7.87e6/5.85e6	466.8/570.8
misex3_Fb@1	28	8	17	1278/137	27.2/3.6	17	0.13e6/0.01e6	43.4/5.5
c1908_F469@0	66	11	5	16413/20837*	*	17	1.59e6/1.46e6	891.8/1225.4
c6288_F69gat@1	64	6	2	3967/4432*	*	17	0.60e6/0.14e6	1184.3/310.9
c3540_F20@1	100	6	2	19932/7473*	*	0	–	⋄

* *bsolo* time out on some instances.

⋄ *eclipse-bf* times out on all instances. The mean/std for the nodes of these unsuccessful runs is 4.65e6/3.92e6.

This table compares *bsolo* with *eclipse-bf* on the P-classes of size 16 for five difficult benchmarks from [41]. For each benchmark, each solver runs 17 times, once on the reference benchmark and 16 times on the P-class instances. For each benchmark, the table reports the number of primary-input variables and the optimal cost. Then for each solver, the table reports the number of runs completed successfully within the 3000 seconds timeout. It also reports the mean and standard deviation of the runtime and nodes for the *successful* runs.

Table 6.12: Comparison of *cplex*, *eclipse-cp* and *zchaff* on P-Classes of size of 32 from the SAT benchmarks.

benchmark	measure	*cplex*	*eclipse-cp*	*zchaff*
queen19	time	0.2/0.1	8.7/0.4	13.7/6.5
	nodes	5/8	2/0	292/3
hanoi3	time	1.7/1.4	3.0/0.7	0.01/0.01
	nodes	485/445	3/1	43/20
bw_large_b	time	64.6/39.7	147.7/63.7	0.01/0.01
	nodes	17/10	3/1	38/3
uf_250_1065_027	time	3600*	3600*	9.8/8.7
	nodes	–	–	39/2
sched07s_v1386	time	1.3/0.04	12.6/1.5	1.9/1.6
	nodes	1/0	1/0	115/64

This table compares *cplex*, *eclipse-cp* and *zchaff* on P-classes of size 32 from five common *Sat* benchmarks. The mean and standard deviation of runtime and nodes are reported. Since all these benchmarks are satisfiable, *zchaff* is expected to be much faster than *cplex* and *eclipse-cp* because it only needs to find *one* solution whereas the other two solvers may need to consider many solutions for quality comparison. However, on *queen19*, *zchaff* is slower than both *cplex* and *eclipse-cp*. On *sched07s_v1386*, *zchaff* is slower than *cplex*.

6.5 SAT Comparisons

MinCostSat is harder than *Sat* because in *Sat*, only one satisfying assignment is needed to prove the satisfiability of a given benchmark. In Table 6.12, we compare the state-of-the-art *Sat* solver *zchaff* with *cplex* and *eclipse-cp* on P-classes of size 32 for five common *Sat* benchmarks. Since all these benchmarks are satisfiable, *zchaff* is expected to be much faster than *cplex* and *eclipse-cp*. However, on *queen19*, *zchaff* is slower than both *cplex* and *eclipse-cp* because they explore many fewer nodes than *zchaff*. On *sched07s_v1386*, *zchaff* is slower than *cplex* and explores many more nodes than *cplex* and *eclipse-cp*. We observe that both *cplex* and *eclipse-cp* find an integer optimal solution to the *LP* relaxation at the root; therefore, no further searches are needed. For the other three benchmarks, there is no surprise: *zchaff* does run faster than *cplex* and *eclipse-cp*.

Chapter 7

Conclusions and Future Work

In the past, many *MinCostSat* problems were considered separately. This work, for the first time in the literature, tries to bring these seemingly isolated problems into a single framework of *MinCostSat*. Our first goal, achieved in Chapter 2, is to comprehensively classify *MinCostSat* problems and introduce their applications, benchmarks as well as solvers. We observe that native and non-native *MinCostSat* problems come from totally different problem domains. Even among native *MinCostSat* problems themselves, covering and non-covering problems are drastically different from each other, both in terms of applications and benchmark structures. The majority of this thesis has been limited to the study of covering problems and their algorithms.

Our second goal is to improve the state of the art for branch-and-bound *MinCostSat* algorithms. In Chapter 3, we surveyed essential techniques used in leading branch-and-bound covering solvers that include reduction algorithms, MIS-based lower-bounding and branching variable selection heuristics. We also proposed new techniques for LPR and CP lower bounding, upper bounding and search tree exploration in Chapter 4. Combining existing and new techniques, we implemented two covering solvers *eclipse-lpr* and *eclipse-cp*, which in most cases can solve an instance by exploring many fewer nodes than competing solvers. However, this saving does not always imply faster runtime because LPR and CP lower bounding can be very time consuming. Any significant improvement of *eclipse-lpr* and *eclipse-cp* requires speeding up their lower-bounding procedures.

The third goal is to devise an efficient local-search *MinCostSat* solver that scales much better than branch-and-bound solvers. In Chapter 5, we presented a local-search *MinCostSat* solver, *eclipse-stoc*, that utilizes well-known algorithms in the *Sat* literature to successfully address both the feasibility issue and the quality issue of *MinCostSat*.

Our last goal is to study special cases of *MinCostSat*. In Chapter 6, we presented five of them that include two-level logic minimization, FPGA detailed routing, *Max-Sat*, minimum-size test pattern problem, and *Sat*. In each case, we present new approaches that are competitive with the state of the art.

Future Research Directions. We believe the following directions are pursuing:

- **Efficient Lower-Bounding Procedures for Covering Problems.** The performance of *eclipse* depends largely on the speed of its lower-bounding procedure. We have observed that even though *eclipse-cp* is able to explore very few nodes compared to the other solvers, it does not always have the fastest runtime. Already articulated in Chapter 4.6, in order to make the lower bounding more efficient, we may (1) dynamically choose a suitable lower-bounding strategy to avoid using expensive lower-bounding on all problems, (2) do lower bounding incrementally without generating a new *LP* instance each time *eclipse* needs to do lower bounding, and (3) generate the Gomory cuts directly so we may be able to decide which Gomory cuts to generate. This may achieve a good balance between the quality of the lower-bounds returned and the time consumed by the lower-bounding procedure.

- **Algorithms for Non-Covering MinCostSat Problems.** Significant amount of work in this thesis has concentrated on a subset of *MinCostSat* — the covering problems. Our work on non-covering *MinCostSat* problems is currently very limited. We observe that even though *eclipse-bf* does not use any advanced *Sat* techniques, it can be competitive with *bsolo* on most minimum-cost test pattern benchmarks. However, *eclipse-bf* explores many more nodes than *bsolo* on the same benchmarks. In addition, the performance of *eclipse-bf* seems to degrade for instances with large number of primary-input variables. We plan to build a non-covering *MinCostSat* solver on top of *eclipse-bf* that implements *Sat*-based conflict diagnosis and non-chronological backtracking. It remains an open question whether the savings provided by these techniques can compensate their runtime overhead.

- **Branch-and-bound SAT Solver Enhanced by Local-Search Procedure.** In recent *Sat* competitions [86], local-search *Sat* solvers are shown to be inferior to branch-and-bound *Sat* solvers on large satisfiable industrial benchmarks. This is counter-intuitive because local-search *Sat* solvers are expected to have better scaling behavior for *large* benchmarks. One common problem local-search solvers have (not just for *Sat*) is their inability to escape local minima. We saw in Section 4.4.2 that doing local search at each node of the branching tree is more effective in finding the optimal solution than doing local search only at the root. This indicates that the branching helps to diversify the local search and effectively prevents it from getting stuck in local minima. Therefore, we expect a branch-and-bound *Sat* solver that conducts a local-search at each node to be effective on satisfiable instances — it combines the scalability of local-search solvers with the systematic search of branch-and-bound solvers. Clearly, this solver is also capable to proving an instance unsatisfiable.

Bibliography

[1] G.D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms.* Kluwer Academic Publishers, 1996.

[2] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In H. Shrobe and T. Senator, editors, *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 1194–1201, Menlo Park, California, 1996. AAAI Press.

[3] H. Kautz and B. Selman. Unifying SAT-based and graph-based planning. In J. Minker, editor, *Proceedings of the Workshop on Logic-Based Artificial Intelligence, Washington, DC, June 14–16, 1999*, College Park, Maryland, 1999. Computer Science Department, University of Maryland.

[4] A. Meier, C. Gomes, and E. Melis. Heavy-tailed behavior and randomization in proof planning. In *Proceedings of the Workshop on Model-Based Validation of Intelligence on AAAI Spring Symposium*, 2001.

[5] J.P. Marques-Silva and K.A. Sakallah. Boolean satisfiability in electronic design automation. In *Proceedings of the Design Automation Conference*, 2000.

[6] J.P. Marques-Silva and K.A. Sakallah. Robust search algorithms for test pattern generation. In *Proceedings of the Fault-Tolerant Computing Symposium*, 1997.

[7] A. Gupta and P. Ashar. Integrating a boolean satisfiability checker and BDDs for

combinatorial equivalence checking. In *Proceedings of International Conference in VLSI Design*, 1998.

[8] J.P. Marques-Silva and T. Glass. Combinatorial equivalence checking using satisfiability and recursive learning. In *Proceedings of the Design and Test in Europe Conference*, 1999.

[9] R.G. Wood and R.A. Rutenbar. FPGA routing and routability estimation via Boolean satisfiability. In *Proceedings of the ACM Fifth International Symposium on Field-Programmable Gate Arrays*, pages 119–125. ACM Press, 1997.

[10] G. Nam, K.A. Sakallah, and R.A. Rutenbar. Satisfiability-based layout revisited: Detailed routing of complex FPGAs via search-based Boolean SAT. In *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, pages 167–175. ACM Press, 1999.

[11] G. Nam, F. Aloul, K.A. Sakallah, and R. Rutenbar. A comparative study of two Boolean formulations of FPGA detailed routing constraints. In *Proceedings of the 2001 International Symposium on Physical Design*, pages 222–227. ACM Press, 2001.

[12] G. Nam, K.A. Sakallah, and R.A. Rutenbar. A Boolean satisfiability-based incremental rerouting approach with application to FPGAs. In *Proceedings of the Design Automation and Testing Europe*, 2001.

[13] H. Xu, R.A. Rutenbar, and K.A. Sakallah. sub-SAT: A formulation for related Boolean satisfiability with applications in routing. *IEEE Transactions on Computer-Aided Design*, 22:814–820, June 2003.

[14] S.O. Memik and F. Fallah. Accelerated SAT-based scheduling of control/data flow graphs. In *Proceedings of the International Conference on Computer Design*, 2002.

[15] J.P. Marques-Silva and K.A. Sakallah. GRASP: A new search algorithm for satisfiability. *CSE-TR-292-96, the University of Michigan*, 1996.

[16] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the Design Automation Conference*, 2001.

[17] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT solver. In *Proceedings of the Design Automation and Test in Europe*, pages 142–149, 2002.

[18] S. Khanna, M. Sudan, L. Trevisan, and D. Williamson. The approximability of constraint satisfaction problems. *SIAM Journal of Computing*, 30(6):1863–1920, 2001.

[19] G.D. Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill Publishers, 1994.

[20] ILOG. CPLEX Homepage, 2004. Information on CPLEX is available at `http://www.ilog.com/products/cplex/`.

[21] J. Bramel and D. Simchi-Levi. On the effectiveness of set covering formulations for the vehicle routing problem. *Operations Research*, 45:295–301, 1997.

[22] J. Holm. Airline crew scheduling during tracking. *Master's Thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU*, 2002.

[23] R.L. Rudell. Logic synthesis for vlsi design. *Ph.D. Dissertation, Department of EECS, University of California at Berkeley*, 1989.

[24] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. A fully implicit algorithm for exact state minimization. In *Proceedings of the 31st Design Automation Conference*, pages 684–690. ACM Press, 1994.

[25] S. Jeong and F. Somenzi. A new algorithm for the binate covering problem and its application to the minimization of Boolean relations. In *Proceedings of the 1992 IEEE/ACM International Conference on Computer-aided Design*, pages 417–420. IEEE Computer Society Press, 1992.

[26] J.E. Beasley. An algorithm for set covering problems. *European Journal of Operations Research*, 31:85–93, 1987.

[27] O. Coudert and J.C. Madre. New ideas for solving covering problems. In *Proceedings of the ACM/IEEE Design Automation Conference*, 1995.

[28] O. Coudert. On solving covering problems. In *Proceedings of the 33rd Design Automation Conference*, pages 197–202, 1996.

[29] S. Liao and S. Devadas. Solving covering problems using lpr-based lower bounds. In *Proceedings of the 34th Design Automation Conference*, pages 117–120, 1997.

[30] E.I. Goldberg, L.P. Carloni, T. Villa, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Negative thinking in branch-and-bound: the case of unate covering. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19:1–16, 2000.

[31] T. Villa, T. Kam, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Explict and implicit algorithms for binate covering problems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16:677–691, 1997.

[32] T.A. Feo and M. Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8:67–71, 1989.

[33] J.E. Beasle. A Lagrangian heuristic for set-covering problems. *Naval Research Logistics*, 37(1):151–164, 1990.

[34] N. Karmarkar, M. Resende, and K.G. Ramakrishnan. An interior point algorithm to solve computationally difficult set covering problems. *Mathematical Programming*, 52:597–618, 1991.

[35] A. Caprara, M. Fischetti, and P. Toth. A heuristic method for the set covering problem. *Lecture Notes in Computer Science*, 1084:72–84, 1995.

[36] M.S.F. Catalamo and F. Malucelli. Parallel randomized heuristics for the set covering problem. *Technical Report, Transportation and Traffic Engineering Section, Delft University of Technology, the Netherlands*, 2000.

[37] P.F. Flores, H.C. Neto, and J.P. Marques-Silva. An exact solution to the minimum size test pattern problem. *IEEE Transactions on Design Automation of Electronic Systems*, 6(4):629–644, 2001.

[38] M. Balducini, G. Brignoli, G.A. Lanzarone, F. Magni, and A. Provetti. Experiments in answer sets planning. *Quarterly Bullettin of the Italian Artificial Intelligence Association*, 2000.

[39] R. Sebastiani, P. Giorgini, and J. Mylopoulos. Simple and minimum-cost satisfiability for goal models. *Lecture Notes in Computer Science*, 3084:7–11, 2004.

[40] P. Liberatore. Algorithms and experiments on finding minimal models. *Technical Report, Department of Computer and System Sciences, University of Rome "La Sapienze"*, 1999.

[41] V.M. Manquinho and J.P. Marques-Silva. Search pruning techniques in SAT-based branch-and-bound algorithms for the binate covering problem. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21:505–516, 2002.

[42] F. Aloul, A. Ramari, I. Markov, and K.A. Sakallah. Generic ILP versus specialized 0-1 ILP: an update. In *Proceedings of the International Conference on Computer Aided Design*, pages 450–457, 2002.

[43] B. Cha, K. Iwama, Y. Kambayashi, and S. Miyazaki. Local search algorithms for partial MAXSAT. In *Proceedings of AAAI-97*, pages 263–268, 1997.

[44] B. Randerath, E. Speckenmeyer, E. Boros, P. Hammer, A. Kogao, K. Makino, B. Simeone, and O. Cepek. A satisfiability formulation of problems on level graphs. *Rutcor Research Report 40*, 2001.

[45] R.J. Wallace and E.C. Freuder. Comparative studies of constraint satisfaction and davis-putnam algorithms for maximum satisfiability problems. *Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge, Amer. Math. Soc., Providence, RI, USA*, 1995.

[46] B. Borchers and J. Furman. An two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. *Journal of Combinatorial Optimization*, 2:299–306, 1999. The benchmarks and the solver are available at `http://www.nmt.edu/∼borchers/maxsat.html`.

[47] T. Alsinet, F. Manya, and J. Planes. Improved branch and bound algorithms for MAX-SAT. In *Proceedings of SAT2003, Sixth International Symposium on the Theory and Applications of Satisfiability Testing, May 5-8 2003, S. Margherita Ligure - Portofino, Italy*, May 2003.

[48] T. Alsinet, F. Manya, and J. Planes. Improved brach and bound algorithms for Max-2-SAT and weighted Max-2-SAT. In *Proceedings of the Sixth Catalan Conference on Artificial Intelligence (CCIA 2003)*, October 2003.

[49] H. Zhang, H. Shen, and F. Manya. Exact algorithms for MAX-SAT. *Electronic Notes in Theoretical Computer Science*, 2003. To appear.

[50] P. Mills and E. Tsang. Guided local search for solving SAT and weighted MAX-SAT problems. *SAT2000 - Highlights of Satisfiability Research in the Year 2000*, pages 459–465, 2000.

[51] M. Yagiura and T. Ibaraki. Efficient 2 and 3-flip neighborhood search algorithms for the MAX-SAT: Experimental evaluation. *Journal of Heuristics*, 7:423–442, 2001.

[52] K. Smyth, H. Hoos, and T. Stützle. Iterated robust tabu search for MAX-SAT. In *Proceedings of the 16th Canadian Conference on Artificial Intelligence*, 2003.

[53] H. Kautz, B. Selman, and Y. Jiang. A general stochastic approach to solving problems with hard and soft constraints. In D. Gu, J. Du, and P. Pardalos, editors, *The Satisfiability Problem: Theory and Applications*, pages 573–586, 1997.

[54] H. Zhang. SATO: An efficient propositional prover. In *Proceedings of the Conference on Automated Deduction*, pages 272–275, 1997. Version 3.2 of SATO is available at `ftp://cs.uiowa.edu/pub/hzhang/sato/sato.tar.gz`.

[55] D.R. Fulkerson and G.L. Nemhauser. Two computationally difficult set covering problems that arise in computing the 1-width of incidence matrices of Steiner triple systems. *Mathematical Programming Study*, 2:72–81, 1974.

[56] S. Yang. Logic synthesis and optimization benchmarks user guide. Technical Report 1991-IWLS-UG-Saeyang, MCNC, Research Triangle Park, NC, January 1991. This report is now available from `http://www.cbl.ncsu.edu/publications/` and benchmark directories archived at `http://www.cbl.ncsu.edu/benchmarks/` .

[57] H. Hoos and T. Stützle. SATLIB: An online resource for research on SAT, 2000. For more information, see `http://www.satlib.org`.

[58] F. Brglez, X. Y. Li, and M. Stallmann. On SAT instance classes and a method for reliable performance experiments with SAT solvers. *Annals of Mathematics and Artificial Intelligence*, 2004. In print. For links to pre-prints, data sets and experimental testbed, see also `http://www.cbl.ncsu.edu/publications/`.

[59] H. Zhang and M.E. Stickel. Implementing the Davis-Putnam method. *Kluwer Academic Publisher*, 2000.

[60] X.Y. Li, M.F. Stallmann, and F. Brglez. Engineering an improved MAX-SAT solver. Technical Report 01, Computer Science Department, North Carolina State University, Raleigh, NC, January 2004.

[61] J.P. Walser and H. Kautz. *Integer Optimization by Local Search: A Domain-Independent Approach (Lecture Notes in Artificial Intelligence)*. Springer Verlag, 1999.

[62] S. Joy, J. Mitchell, and B. Borchers. A branch and cut algorithm for MAX-SAT and weighted MAX-SAT. In *Proceedings of the DIMACS Workshop on Satisfiability: Theory and Applications*, 1998.

[63] R. Boppana and M. Halldórsson. Approximating maximum independent sets by excluding subgraphs. *Bit*, 32:180–196, 1992.

[64] P. Barth. A Davis-Putnam enumeration algorithm for linear pseudo-Boolean optimization. Technical Report MPI-I-95-2-003, Max Plank Institute Computer Science, 1995.

[65] V. Manquinho, P. Flores, J.P. Marques-Silva, and A. Oliverira. Prime implicant computation using satisfiability algorithms. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence*, pages 117–120, 1999.

[66] J.P. Marques-Silva. Search algorithms for satisfiability problems in combinational switching circuits. *Ph.D. Dissertation, EECS Department, University of Michigan*, 1995.

[67] R.E. Gomory. Outline of an algorithm for integer solution to linear programs. *Bulletin of the American Mathematical Society*, 64:275, 1958.

[68] R.E. Gomory. An algorithm for the mixed integer problem. *RM-2537. Santa Monica California: Rand Corporation*, 1960.

[69] T.M. Ozan. *Applied Mathematical Programming for Production and Engineering Management.* Prentice-Hall, 1986.

[70] D.A. McAllester, B. Selman, and H. Kautz. Evidence for invariants in local search. In *Proceedings of AAAI/IAAI*, pages 321–326, 1997.

[71] E. Hirsch and A. Kojevnikov. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination. In *Electronic Proceedings of Fifth International Symposium on the Theory and Applications of Satisfiability Testing*, 2002.

[72] X.Y. Li, M.F. Stallmann, and F. Brglez. A local search solver using an effective switching strategy and an efficient unit propagation. *Lecture Notes in Computer Science*, 2919:53–68, 2004. For links to re-prints, data sets and experimental testbed, see also `http://www.cbl.ncsu.edu/publications/`.

[73] B. Selman, H.J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In P. Rosenbloom and P. Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California, 1992. AAAI Press.

[74] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 5:533–549, 1986.

[75] W.V.O. Quine. The problem of simplifying truth functions. *American Mathematics Monthly*, 59:521–531, 1952.

[76] P.C. McGeer, J.V. Sanghavi, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Espresso-signature: A new exact minimizer for logic functions. In *Proceedings of the 30th Design Automation Conference*, pages 618–624, 1993.

[77] V. Betz and J. Rose. VPR: A new packing, placement and routing tool for FPGA research. In *Proceedings of the Seventh Annual Workshop on Field Programmable Logic and Applications*, pages 213–222, 1997.

[78] G. Lemieux and S. Brown. A detailed router for allocating wire segments in FPGAs. In *Proceedings of ACM Physical Design Workshop*, 1993.

[79] S. Wilton. Architecture and algorithms for field-programmable gate arrays with embedded memories. *Ph.D. Dissertation, University of Toronto*, 1997.

[80] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[81] R.G. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

[82] H. Kautz, D. McAllester, and B. Selman. Encoding plans in propositional logic. *KR'96: Principles of Knowledge Representation and Reasoning*, pages 374–384, 1996. The SATPLAN benchmark set is available at `http://sat.inesc.pt/benchmarks/cnf/satplan/`.

[83] B. Borchers and J. Furman. A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. *Technical Report, Mathematics Department, New Mexico Tech*, 1995.

[84] G. Nemhauser, M. Savelsbergh, and G. Sigismondi. MINTO: a Mixed INTeger Optimizer. *Operations Research Letters*, 15:425–441, 1994.

[85] P. Flores, H. Neto, and J.P. Marques-Silva. An exact solution to the minimum size test pattern problem. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, pages 510–515, 1998.

[86] D. Le Berre and L. Simon. SAT Solver Competition, in conjunction with 2003 SAT Conference, May 2003. For more information, see `http://www.satlive.org/SATCompetition/2003/comp03report/index.html`.