Local Search Methods for Constraint Problems

Muhammad Rafiq Bin Muhammad

This thesis is submitted in total fulfilment of the requirements of the degree of

Master in Computer Science

NICTA Victoria Research Lab
Department of Computer Science and Software Engineering
University of Melbourne
Australia

November, 2006
Revised On October, 2007
Final Revised On February, 2008

(Acid-Free Paper)
Abstract

This thesis investigates the use of local search methods in solving constraint problems. Such problems are very hard in general and local search offers a useful and successful alternative to existing techniques.

The focus of the thesis is to analyze the techniques of invariants used in local search. The use of invariants have recently become the cornerstone of local search technology as they provide a declarative way to specify incremental algorithms. We have produced a series of program libraries in C++ known as the One-Way-Solver. The One-Way-Solver includes the implementation of incremental data structures and is a useful tool for the implementation of local search. The One-Way-Solver is applied to two challenging constraint problems, the Boolean Satisfiability testing (SAT) and university course timetabling problems.

In the implementation of Boolean Satisfiability testing (SAT), we introduced a pure non-clausal solver based on well-known stochastic local search techniques. It works directly on arbitrary non-clausal formula including pseudo-Boolean constructs and other extended Boolean operators. We introduced a novel scoring function using positive and negative values and empirically show that on certain benchmarks our non-CNF local search solver can outperform highly optimized CNF local search solvers as well as existing CNF and non-CNF complete solvers.

In the implementation of university course timetabling problem, we developed a weighted penalty scheme (WPS) local search technique. WPS is a local search method guided by a multiplier that provides a force to determine the concentration of the search as well as mechanism to escape local minima hence it can be considered as a meta-heuristic technique. When used with the One-Way-Solver, WPS achieved very promising results in solving university course timetabling problem.
Declaration

The work in this thesis is based on research carried out at the NICTA Victoria Research Lab, the Department of Computer Science and Software Engineering, University of Melbourne, Australia.

This is to certify that

(i) no part of this thesis has been submitted elsewhere for any other degree or qualification.

(ii) the thesis comprises only my original work towards the Master.

(iii) due acknowledgement has been made in the text to all other material used.

(iv) the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

..................................................  
(MUHAMMAD RAFIQ BIN MUHAMMAD)

Copyright © 2006 by Muhammad Rafiq Bin Muhammad.
“The copyright of this thesis rests with the author. No quotations from it should be published without the author’s prior written consent and information derived from it should be acknowledged”.

iii
Acknowledgements

Mostly I would like to thank my supervisor Professor Peter Stuckey for his boundless help, criticism and patience. Without his supervision and expert knowledge in Artificial Intelligence especially in constraint optimization techniques, this thesis would not have been possible. I would also like to thank all the members of G12 research group. In no particular order, thanks to: Mark Wallace, Ralph Becket, Mark Brown, Zoltan, Kim Marriott, Peter Schachte, Neng Fa and Reza who have provided help and advice during my attachment with the group. Special thanks to Noraini for her thorough proof reading and constructive criticism. Finally, I thank my wife Nurul and my kids Nurin and Amin for their patience, encouragement and companionship. Thanks to all who gave me the support during these last few years. Thanks to God Almighty for making my effort a success.
Contents

Abstract ii
Declaration iii
Acknowledgements iv

1 Introduction 1

1.1 Constraint Satisfaction Problem (CSP) ......................... 2
1.2 Techniques in solving Constraint Problems .................... 3
1.2.1 Mathematical Programming Techniques ..................... 3
1.2.2 Searching Techniques ..................................... 4
1.3 Constraint Programming ......................................... 7
1.4 Thesis Contributions ............................................... 8
1.5 Thesis Organization ............................................... 9

2 Local Search and Invariants 11

2.1 Stochastic Local Search ........................................... 12
2.1.1 Problem Definition ........................................... 13
2.1.2 Objective Function .......................................... 14
2.1.3 Local Move and Neighborhood Function .................... 15
2.1.4 Local Minima/Maxima ....................................... 16
2.1.5 Meta-Heuristics .............................................. 17
2.1.6 Beyond Meta-Heuristics ..................................... 25
2.2 Invariants: One-Way-Solver .................................... 25
2.2.1 Invariants ...................................................... 27
## Contents

2.2.2 Arithmetic Invariants ........................................ 27  
2.2.3 List Invariants ................................................. 35  
2.2.4 Implementation .................................................. 42  
2.3 Chapter Summary .................................................. 44  

3 Stochastic Non-CNF SAT Solver ..................................... 46  
3.1 Satisfiability Problem ............................................. 46  
3.1.1 SAT Solvers ......................................................... 47  
3.1.2 Stochastic Local Search and SAT ................................ 48  
3.2 Non-CNF SAT ......................................................... 49  
3.2.1 The State of non-CNF SAT Solvers ............................... 51  
3.3 Stochastic Local Search Non-CNF SAT ............................. 52  
3.3.1 Score ................................................................. 52  
3.3.2 Searching for a Solution ......................................... 58  
3.3.3 Implementation ..................................................... 60  
3.4 Non-CNF SAT Benchmarks .......................................... 61  
3.4.1 Hard Random Non-CNF Formula .................................. 61  
3.4.2 Social Golfers as Non-CNF SAT Problem ......................... 61  
3.4.3 Minimum Disagreement Parity Problem ......................... 64  
3.5 Experimental Results .............................................. 67  
3.5.1 Hard Random Non-CNF Formula ................................. 68  
3.5.2 Social Golfer Problem ............................................. 69  
3.5.3 Minimal Disagreement Parity Problem ......................... 70  
3.6 Chapter Summary ..................................................... 70  

4 University Course Timetabling Problem .............................. 72  
4.1 University Course Timetabling ..................................... 72  
4.2 Methods in Course Timetabling .................................... 74  
4.2.1 Sequential Approaches ........................................... 74  
4.2.2 Cluster Approaches ................................................ 75  
4.2.3 Constraint Programming Approaches ............................ 75  
4.2.4 Meta-Heuristic Local Search Approaches ....................... 76
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.5 Hybrid Approaches</td>
<td>77</td>
</tr>
<tr>
<td>4.3 Course Timetabling Benchmarks</td>
<td>77</td>
</tr>
<tr>
<td>4.3.1 A Simple Timetabling Example</td>
<td>78</td>
</tr>
<tr>
<td>4.4 Timetabling Problem Modeling</td>
<td>79</td>
</tr>
<tr>
<td>4.5 Solution Representation</td>
<td>81</td>
</tr>
<tr>
<td>4.6 Searching for Solution</td>
<td>82</td>
</tr>
<tr>
<td>4.6.1 Searching for Feasible Solution</td>
<td>82</td>
</tr>
<tr>
<td>4.6.2 Searching for Quality Solution</td>
<td>86</td>
</tr>
<tr>
<td>4.7 Implementation</td>
<td>86</td>
</tr>
<tr>
<td>4.8 Experiment Results</td>
<td>87</td>
</tr>
<tr>
<td>4.9 Chapter Summary</td>
<td>89</td>
</tr>
<tr>
<td>5 Conclusions</td>
<td>94</td>
</tr>
<tr>
<td>5.1 Invariants</td>
<td>94</td>
</tr>
<tr>
<td>5.2 Non-CNF SAT</td>
<td>95</td>
</tr>
<tr>
<td>5.3 Timetabling</td>
<td>96</td>
</tr>
<tr>
<td>Bibliography</td>
<td>97</td>
</tr>
</tbody>
</table>
# List of Figures

2.1 Local minima[59] ................................................................. 17  
2.2 Crossover of two chromosomes ........................................... 24  
2.3 Topological order of arithmetic operations ................................. 28  
2.4 Valuation of $f(x)$ when $A = 3$, $B = 4$, $C = 5$ and $D = 7$ .......... 28  
2.5 Incremental update of equation tree ....................................... 29  
2.6 Structure representing the invariant variable for addition and subtraction operations ........................................ 29  
2.7 Structure representing the invariant variable for multiplication and division operations ........................................ 31  
2.8 Topological order of arithmetic operations ................................ 32  
2.9 Incremental update of equation tree ....................................... 33  
2.10 Priority queue ..................................................................... 34  
2.11 Don’t care nodes ................................................................... 35  
2.12 Min heap for list 3, 4, 5, 6, 7, 8 .............................................. 37  

3.1 The propositional formula $\phi = (((\neg A \vee \neg B \vee \neg C) \land \neg D \land (\neg E \lor \neg F)) \land \neg C \land ((\neg D \lor A \lor \neg E) \oplus (C \land F)))$ with scores for the valuation \{$A, \neg B, C, \neg D, \neg E, F$\} bracketed. ............................................ 53  
3.2 AND score (a) where the node is false and (b) where it is true ........ 54  
3.3 OR score (a) where the node is false and (b) where it is true .......... 54  
3.4 XOR score (a) where the node is false and (b) where it is true ........ 55  
3.5 ATMOST 2 score (a) where the node is false and (b) where it is true 56  
3.6 ATLEAST 2 score (a) where the node is false and (b) where it is true 57
3.7 COUNT 2 score (a) where the node is true and (b)(c) where they are false ........................................ 58
3.8 The formula $\phi$ with scores for the valuation $\{A, \neg B, C, \neg D, E, F\}$... 60
3.9 The formula $\phi$ with scores for the valuation $\{A, \neg B, C, \neg D, E, \neg F\}$. 60

4.1 Example of empty timetable slots ........................................ 80
4.2 WPS: Escaping local minima ........................................ 83
4.3 Result of soft constraint violation for small instances ............. 89
4.4 Result of soft constraint violation for medium instances .......... 90
4.5 Result of soft constraint violation for large01 instances ........... 90
4.6 Result of soft constraint violation for large02 instances .......... 91
4.7 Percentage of invalid solution for large01 .................. 91
4.8 Percentage of invalid solution for large02 .................. 92
4.9 Comparison for Feasible Time between Naive and Heuristic Move for Medium Instances .......................... 92
4.10 Comparison for Feasible Time between Naive and Heuristic Move for Large Instances ..................... 93
List of Tables

2.1 Gene encoding of variable $x$ ........................................ 23

3.1 Comparison of number of variables and clauses between CNF and Non-CNF encoding ........................................ 64

3.2 Comparison of CNF and non-CNF encoding of MDP problem ........ 67

3.3 Comparative results for CNF solvers on hard random formulas .... 68

3.4 Comparative results for non-CNF solvers on hard random formulas .. 69

3.5 Comparative results for solvers on social golfer problems .......... 69

3.6 Comparative results for solvers on MDP problems ................. 70

4.1 Room Features ...................................................... 78

4.2 Features required by each event .................................... 78

4.3 Students subscription to events ...................................... 79

4.4 Attributes of benchmark instances ................................ 87

4.5 Summary of Results for All Timetabling Problem Instances ........ 88
Chapter 1

Introduction

Constraint problems can be found in all aspects of human life. From simple day to day planning to complex resource allocation and timetable scheduling, they are omnipresent in almost every aspect of our life. When dealing with constraint problems, humans can quickly fiddle with the constraint requirements and construct solutions that are satisfactory or least sub-optimal. However, this is often limited by the size and complexity of the problems.

For decades, computers have been used to assist humans in solving large and complex problems. This is due to the mechanical computation capability of computer, using fixed programmed rules to allow them to perform simple monotonous tasks repeatedly and reliably, which humans are ill-suited to. Nevertheless, solving constraint problems can be extremely challenging. Many of these problems fall into the class of combinatorial optimization problems which are NP (Nondeterministic Polynomial)-complete [29]. Assuming that $P \neq NP$, then an NP-complete problem cannot be solved exactly in polynomial time and no single approach is likely to be effective on all problems. The solution requires a combination of heuristics and combinatorial search methods to be solved in a reasonable time. In this thesis it is this class of constraint problems that we are referring to. Problems like transportation arrangement, supply-chain management, job-shop scheduling, Boolean satisfiability testing and university course timetabling are just a few well-known examples of NP-Complete constraint problems.

Constraint problems are the subject of intense research in both artificial intel-
1.1 Constraint Satisfaction Problem (CSP)

Constraint Satisfaction Problem or CSP [125] as defined in [109] is a set of variables, $X_1, X_2, ..., X_n$, and a set of constraints, $C_1, C_2, ..., C_m$. Each variable $X_i$ has a nonempty domain $D_i$ of possible values. Each constraint $C_i$ involves some subset of the variables and specifies the allowable combinations of values for that subset. A state of the problem is defined by an assignment of values to some or all of the variables, $X_i = v_i, X_j = v_j, ...$. An assignment that does not violate any constraints is called a consistent or legal assignment. A complete assignment is one in which every variable is mentioned, and a solution to a CSP is a complete assignment that satisfies all the constraints.

Consider a simple CSP example. Let variables $A$ and $B$ be positive integer and $C = A < 10, B < A$ be the set of constraints for variables $A$ and $B$. Initially, $A$ and $B$ have the domain $D = 0, 1, 2, ..., \infty$, i.e. the set of positive integers. The constraint $A < 10$ practically limits the domain of $A$, $D_A$ to $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$ since assigning variable $A$ to value 10 is illegal because it will violate the constraint $A < 10$. Assigning $A$ to 5 will result in the domain of $B$, $D_B$, becoming $0, 1, 2, 3, 4$ due to the constraint $B < A$. An example of complete assignment is $A = 5, B = 4$. If $A$ is assigned with value 0, $D_B$ becomes $\phi$. This is a case of a non-solution where a complete assignment cannot be achieved. Similarly, if $B = 9$, then $D_A = \phi$. We can conclude that by combining the constraints in $C$ the domain for $A$ and $B$ are $1, 2, 3, 4, 5, 6, 7, 8, 9$ and $0, 1, 2, 3, 4, 5, 6, 7, 8$ respectively.

intelligence and operations research. They are a focal point for computer scientists as they pose challenging design problems for algorithmic development. This helps to push the computer research to its limit and results in the birth of many innovative tools and techniques in computer science. For the operational research community, constraint problems have its root in many industrial applications. The progress in constraint solving technology helps industries to address critical situations and to become more reactive and cost-effective.
1.2 Techniques in solving Constraint Problems

Over the years, many techniques have been developed to solve constraint problems. They can be categorized into several groups based on the general theme of solving techniques. These include linear programming techniques, constraint programming techniques and search techniques. Each of these techniques has their own strengths and limitations. A brief discussion of each technique is given in the following sections.

1.2.1 Mathematical Programming Techniques

Mathematical Programming (MP) techniques are long established and well-documented approaches to solving optimization problem predominantly in the operation research community. The most commonly used MP formulations are linear programming (LP) and its subset, integer programming (IP). A good introduction to LP can be found in [28]. In LP, all the mathematical functions for the objective function and the constraint must be linear. A nonlinear problem has to be linearized before these techniques can be applied, but often leads to large linear programs due to many new variables and constraints introduced in the linearization process.

In LP, problems are expressed in the standard form as follows:

\[
\begin{align*}
\text{minimize} & \quad cx \\
\text{subject to} & \quad Ax = b \\
& \quad x \geq 0
\end{align*}
\]

(1.1)

where \(x\) is the vector of variables to be solved for, \(A\) is a matrix of known coefficients, and \(c\) and \(b\) are vectors of known coefficients. The aim is to minimize the objective function, \(cx\) subjects to a set of constraints \(Ax = b\) and that the values of \(x\) are positive real number.\(^1\)

Once the problem has been expressed in standard form, next it is solved to optimality through some means. Two families of solution techniques are in wide use

\(^1\)Other forms, such as minimization, as well as problems involving negative variables can always be rewritten into an equivalent problem in standard form.
today. The Simplex method [143] introduced in 1950s which runs along polytope edges of the visualization solid to find the best answer. A much more efficient polynomial time algorithm was found by Karmarkar [78]. This method goes through the interior points hence known as an interior points method. It has received a great deal of attention ever since its introduction in 1984. Both techniques visit a progressively improving series of trial solutions, until a solution is reached that satisfies the conditions of optimality.

Integer programming (IP) (or integer linear programming, strictly speaking) is the subset of LP where the variables are required to take integer values. They have the advantage of being more realistic than LP, but the disadvantage of being much harder to solve. The primary limitation of IP is when dealing with problems that are “highly combinatorial” such as scheduling, sequencing, and assignment, where the construction of an equivalent IP would require the definition of large numbers of zero-one variables.

1.2.2 Searching Techniques

Most of the constraint problems are typically solved using a form of search, i.e. to explore the possible combinations of values in order to determine the appropriate solution. These techniques have gained popularity as they offer substantial flexibility and efficiency in dealing with various constraint problems. The most used techniques are variants of backtracking, constraint propagation, and local search.

Backtracking

Backtracking is a recursive branch and search algorithm. It starts with a list of unassigned variables. At each step, a variable is chosen, and a possible value is assigned to it. For each value, the consistency of the assignment with the constraints is checked; in case of consistency, the algorithm proceeds with the next variable. Otherwise, a new possible value is chosen for assignment. When all values of a variable have been tried, the algorithm backtracks. A solution is found when a complete assignment (i.e. all the variables are assigned with value) that satisfies all the constraint is achieved.
The basic backtrack algorithm is indeed very simple. However it can be very inefficient. Several variants of “clever” backtracking exists. Backmarking [48] improves the efficiency of checking consistency by maintaining information about the last time a variable was instantiated to a value and the effect of the instantiation. By doing so, Backmarking avoids redundant consistency checks by quickly replicating the effect when an instantiation of value is known to produce similar effect. Backmarking does not reduce the search space but only possibly reduces the number of consistency checks for a partial solution.

Backjumping [49] reduces search, therefore increases efficiency. Instead of going up one level in search tree when all values failed, backjumping goes up to more than one level. This is the case when the variable at one (or more) level above is known in advance that they will certainly fail irrespective of the values they are going to take. Backjumping skips parts of search tree hence improves the efficiency.

Look-ahead [106] or forward-checking [109] is also often used in backtracking to attempt to foresee the effects of choosing a variable or a value. The search can skip these variables or values if they are foreseen to fail. Forward-checking can be more time-consuming but may produce better results.

**Constraint Propagation**

Constraint propagation are techniques used to enforce local consistency. When a partial assignment can be extended to another variable in such a way that the resulting assignment is consistent, the local consistency condition is met. Among the most well-known local consistencies are node consistency, arc consistency, hyper-arc consistency and path consistency [91]. The idea behind local consistency is to transform a constraint problem into an equivalent constraint problem but one in which the domains of the variables are reduced. Such a transformation is called constraint propagation. Constraint propagation reduces the domains of variables, strengthening constraints, or creating new ones, leading to a reduction of the search space, hence making the problem easier to solve. In addition, constraint propagation can be used to check for the satisfiability of a problem, although incomplete in general but complete in some particular cases.
Constraint propagation algorithms are often used along with backtracking search (although they can be used with other search techniques). At each step of the backtracking search, constraint propagation is used to rule out domain values of partial assignments that cannot be extended into a solution (i.e. violate local consistency conditions). This is like adding “intelligence” to search because values that are not viable are avoided hence reducing search steps considerably. Consider a simple constraint problem of $X < 4$ and $X + Y = 2$. Let the domain of variables $X$ and $Y$ be positive integer values between 0 and 9. Given the constraint $X < 4$, constraint propagation can immediately rule out the domain values of $X$ that are not locally consistent to $X < 4$. The new domain for $X$ is between 0 and 3. Adding the second constraint $X + Y = 2$, the domain of $X$ is further pruned to be between 0 and 2. Likewise, the domain of $Y$ must now be between 0 and 2.

The search space of the constraint problem has been reduced significantly even before search begins. Now let's assign the value of 2 to $X$. Since $X = 2$ is the valid domain value for $X$, it is guaranteed to be locally consistent with all the constraints. Assigning $X$ to 2 results in the partial solution where further propagation can be conducted. Constraint 2 has now become $2 + Y = 2$ and the only possible value for $Y$ to be locally consistent is 0. As we have seen in this example, constraint propagation can be a very handy tool to improve search efficiency. When augmented with all the other “clever” search techniques, they can form a powerful package to solve complex combinatoric constraint problems.

**Local Search**

Backtrack search is in the class of complete search algorithms, that is the algorithm explores the search space exhaustively and is guaranteed to find a solution or fail if none exist. Local search methods on the other hand are incomplete search algorithms. Such algorithms may find a solution, but then they may fail even if solution to the problem exists. Unlike backtracking search, local search moves from one complete assignment to neighboring complete assignments with the aim of increasing the number of constraints satisfied by the assignments. This is guided by an objective function where the value is computed by the number of constraints satisfied. For
1.3. Constraint Programming

that reason local search is often associated with constraint optimization problems. From a theoretical point of view local search algorithms perform a biased exploration of search space. Although they may fail to find optimal or even a quality solution, in many problem instances they isolate optimal or near-optimal solutions within a very reasonable time (for example see [117]).

One of the key strengths of local search techniques is that the next state in the search is only determined by the current state and the action executed. In other words, local search algorithms operate using single current state and do not retain the path followed by the search algorithms and therefore use very little memory. This makes local search particularly good for large-scale constraint problems and online optimization problem where completeness guarantees have to be sacrificed for performance. The constraint optimization nature of local search also makes it a technique of choice for many optimization problem where systematic search techniques are less effective.

In this thesis, we shall devote our discussion to local search techniques for solving constraint problems. This is an emerging field that has become the subject of intense research in recent decades. As we are constantly challenged by new problems with complexity growing at an unimaginable rate, we hope this study will help us to increase our understanding, produce new and more effective techniques in solving large complex constraint problems.

1.3 Constraint Programming

Constraint Programming (CP) [11] is an emergent concept in computer programming. For years, computer problem solving has revolved around data structures and algorithms. CP on the other hand looks into declarative description of a problem by stating constraints (requirements) about the problem area and, consequently, finding a solution satisfying all the constraints without defining a computational procedure to enforce the constraints. The emergence of CP is one of the the most exciting developments in programming languages of the last decade and, not surprisingly, it has recently been identified by the ACM (Association for Computing Machinery) as
1.4 Thesis Contributions

one of the strategic directions in computer research [45].

The idea of constraint programming is to encapsulate constraint solving technologies (algorithms, techniques, etc) and expose only the declarative aspect of programming to users. Declarative programming in general, is that the user states what has to be solved instead of how to solve it. The constraint solving technologies can range from various algorithms and techniques such as constraint satisfaction via systematic search or local search, constraint consistency techniques and constraint propagation are completely transparent to users.

The main step towards CP was achieved when Gallaire [47], Jaffar and Lassez [73] noted that Logic Programming (LP) was just a particular kind of constraint programming. LP has the property of being declarative in nature, which is very close to the idea of CP. Therefore the combination of constraints and logic programming is very natural and Constraint Logic Programming (CLP) makes a nice declarative environment for solving problems by means of constraints. However, it does not mean that constraint programming is restricted to CLP. Constraints were integrated to typical imperative languages like C++ [62] and Java as well. The strength of CP is that it is backed by a strong theoretical foundation and it is attracting widespread commercial interest as well, in particular, in areas of planning, scheduling and optimisation.

1.4 Thesis Contributions

A major part of this thesis tries to analyze the techniques of invariants, introduced by Michel and Hentenryck [103] to be used in local search. The use of invariants have recently become the cornerstone of local search technology as they provide a declarative way to specify incremental algorithms. We also analyze some of the most commonly used techniques and heuristics in local search. The purpose is to understand the performance implications of employing these techniques in order to optimize the local search performance in solving real world instances. The result of the analysis is the One-Way-Solver, a series of program libraries in C++ for specifying incremental algorithm declaratively. The One-Way-Solver includes the implementa-
tion of arithmetic and list invariants is used intensively in the application of local search. We apply our One-Way-Solver to two challenging constraint problems, the Boolean Satisfiability testing (SAT) and the university course timetabling problem.

In our implementation of Boolean Satisfiability testing (SAT), we introduced a pure non-clausal solver based on well-known stochastic local search techniques. It works directly on arbitrary non-clausal formula including pseudo-Boolean constructs and other extended Boolean operators. To our knowledge, this is the most comprehensive implementation of non-clausal solver based on stochastic local search techniques. Earlier implementations are mainly extension of clausal solver to take conjunction ($\land$), disjunction ($\lor$) and negation ($\neg$) operations in free form, such as [121]. Our solver has included the implementation of the following Boolean gates: NAND, NOR, XOR, XNOR, IMPLIES, IFF, ATLEAST, ATMOST and COUNT operations. In addition, we also introduce a novel way of expressing the scoring function using positive and negative value. This complex scoring is necessary since non-clausal formulas are not restricted to negation normal forms. It provides more accurate heuristic information to assist the local search process.

Another contribution of the thesis is in the implementation of local search technique using Weighted Penalty Scheme (WPS). WPS is a local search method guided by a multiplier that provides a force to determine the concentration search as well as mechanism to escape local minima hence can be considered as one of the meta-heuristic techniques. Together with One-Way-Solver, we applied WPS to solve university course timetabling problem and achieved promising results.

1.5 Thesis Organization

The organization of this thesis is as follows. Chapter 2 provides an introduction to the concepts of local search and invariants. A brief introduction to the theoretical foundation and various techniques used in local search is provided. The implementation of One-Way-Solver is also described in this chapter. In particular, the concepts, theories and algorithm for arithmetic and list invariants are presented in detail.

In Chapter 3, the implementation of Stochastic non-CNF SAT Solver is discussed
1.5. Thesis Organization

in detail. We start with a brief review of SAT theories and existing SAT solver technology. Then we introduce the concept of a non-CNF SAT solver and show that there are advantages of solving SAT problems in non-CNF format. Here, we present our novel scoring function using positive and negative value and empirically show that on certain benchmarks our non-CNF local search solver can outperform highly optimized CNF local search solvers as well as existing CNF and non-CNF complete solvers.

Chapter 4 presents the work on university course timetabling. The Weighted Penalty Scheme (WPS) local search technique is introduced and our empirical study on the performance of WPS as compared to other local search meta-heuristic techniques is provided to validate the effectiveness of the proposed algorithm.

Finally, Chapter 5 contains a summary of the thesis and a discussion of some future research directions.
Chapter 2

Local Search and Invariants

One of the principles of computer problem solving is solving by searching. This approach of problem solving finds solutions by exploring the possible combinations of values of the variables or model of solution. They can be divided into two major categories, i.e. the systematic search algorithms and non-systematic algorithms.

A systematic algorithm explores the search space systematically, i.e. by keeping the search paths in the memory and recording which alternatives have been explored at each point along the path and which have not. When a goal is found, the path to that goal also constitutes a solution to the problem. The requirement of a systematic search method is to formulate the problem such that the state space can be explored exhaustively. Common structures for state space modeling are list, tree and graph. These approaches guarantee to return a solution or fail if none exist.

A systematic search is constrained by the available computing resources (memory, processing time, etc) for exploring every possible value. When dealing with problems that are NP-Hard, such as integrated-circuit design, factory-floor layout, job-shop scheduling, vehicle routing, and timetabling, the search space is often too large to be considered practical for an exhaustive search. As such we have to turn to non-systematic search algorithms whereby the search space is explored independently of any deterministic structure. Often, the search performs a biased exploration of the search space, guided by some form of search heuristics with the hope to find a solution closest to the goal. Approximation in output become necessary when problems cannot be solved computationally with realistic resources. Although they
may fail to find optimal or even quality solutions, on many problem non-systematic
search can find optimal or near-optimal solutions within very reasonable times (for
example see [117]).

This chapter is concerned with non-systematic search algorithms. We shall focus
our attention on one of the most important class of non-systematic search algorithms,
.i.e. the stochastic local search, or in short, local search algorithms. We begin by
discussing the key concepts in local search and how they can be used to solve many
NP-Hard combinatorial problems. Then, we explain the design of One-Way-Solver,
a series of program libraries in C++ that we develop to assist the implementation
of local search. One-Way-Solver combines the aspects of declarative and imperative
programming to ease the process of implementing local search. The imperative
part of the One-Way-Solver provides easy integration into existing programming
languages (i.e. C++) whereas the declarative nature encapsulates the complex
and error-prone implementation of incremental data structure found in local search.
One-Way-Solver helps to speed up the implementation of local search by providing
a declarative way to specify invariants, i.e. the functions and variables that need
to be maintained in order to define the penalty and neighborhood functions of local
search.

2.1 Stochastic Local Search

Stochastic local search works on complete assignments and moves from a complete
assignment to neighboring complete assignments in order to obtain a solution or
a (near-optimal) solution (in optimization problems). The term stochastic is used
in reference to the nondeterministic nature of the search where the move to the
next state can sometime not be determined by the current state and the action
executed. The moves can be simple (change the value of a variable) or involved
more complex strategies (such as change a subset of related variables). The strength
of local search is that it operates using a single current state rather than multiple
paths (hence named “local search”). By using a single current state, local search
algorithms do not retain the path followed by the search algorithm and therefore
have the advantage that they use very little memory and can often find reasonable solutions in large or infinite (continuous) state space for which systematic algorithms are unsuitable.

### 2.1.1 Problem Definition

A Local search problem belongs to the class of mathematical optimization problems. It is a primitive form of continuous optimisation in the discrete search space. The basic idea of optimization in local search is to construct an objective function, and try to minimize this objective by assigning values to variables.

The formal mathematical definition of local search as in [58] is that given a minimization (maximization) problem with objective function $f$ and feasible region $R$, a typical local search algorithm requires that, with each solution point $x_i \in R$, there is a predefined neighborhood $N(x_i) \subset R$. Given a current solution point $x_i \in R$, the set $N(x_i)$ is searched for a point $x_i + 1$ with $f(x_i + 1) < f(x_i)$ ($f(x_i + 1) > f(x_i)$). If such a point exists, it becomes the new current solution point, and the process is iterated. Otherwise, $x_i$ is retained as the local optimum with respect to $N(x_i)$. Then, a set of feasible solution points is generated, and each of them is “locally” improved within its neighborhood.

Typically local search uses constraints to select which moves are desirable. A key aspect of local search is the ability to use local information to guide the search. This is achieved via an objective function $f(x)$ such that a move is guided by what is likely to improve the objective function. Algorithm 2.1 depicts a simple generic local search template.

The search begin by generating an initial complete solution, $s$ (Line 2). The initial solution, $s$ is assumed to be the best solution so far, hence is remembered in variable $s^*$ (Line 3). Next, it enters the optimization loop whereby the algorithm moves from one complete solution, $s$ to another complete solution, $s'$ with the aim to minimize objective function $f(x)$ (Line 4 – 13). The process is repeated for $\texttt{MaxTries}$ number of times and the best solution achieved so far, $s^*$ is returned.

There is no guarantee that $s^*$ is the optimal solution. The search may stop even if the best solution found by the algorithm is not optimal. This can happen due to
2.1. Stochastic Local Search

**Algorithm 2.1 Generic Local Search Template**

1. Procedure LocalSearch
2. s := GenerateInitialSolution()
3. s* := s
4. For k := 0 To MaxTries Do
5. Move(s, s')
6. if f(s') < f(s) then
7. s* := s'
8. s := s'
9. End If
10. End For
11. Return s*
12. End LocalSearch

the impossibility of improving the solution, as the optimal solution can lie far from
the neighborhood of the solutions examined by the algorithm. We can set conditions
to determine if the optimality of the solution has been achieved (such as when \( f(x) = 0 \)). Nevertheless, if the solution does not fulfill the optimality criteria, we cannot
conclusively say that one does not exist. The reason is because the search is not
guaranteed to be exhaustive hence it may overlook possible optimal solutions. As
such, local search is a typical **incomplete algorithm**, i.e. an algorithm that can
only find satisfiability of a problem but not unsatisfiability.

2.1.2 Objective Function

The objective function of local search problem is commonly modeled as discrete un-
constrained optimization problem (although they can also be modeled as continuous
optimization problem). In discrete unconstrained formulation, the model consists
of the sum of the penalties for constraint violations. A typical penalty formulation
is as follows:

\[
f(x) = \sum_{i=1}^{n} w_i g_i(x)
\]  \hspace{1cm} (2.1)

where \( f(x) \) is the penalty function, \( g_i(x) \) is the penalty term and \( w_i \) is the weight coefficients. The value of \( w_i \) can be static or dynamic. A static formulation uses
a fixed value for \( w_i \). In this case, a local minimum of \( f(x) \) is a constrained local
minimum, and a global minimum of \( f(x) \) is a constrained global minimum.

In general, hard-to-satisfy constraints should carry larger penalties than easy-to-satisfy ones. However, the degree of difficulty in satisfying a constraint may depend on other constraints in a problem. Without the ability to vary penalties dynamically, search techniques for unconstrained problems will likely get stuck in infeasible local optima. Dynamic-penalty methods overcome the limitation in static-penalty methods by varying penalties values. By dynamically adjusting the value for \( w_i \), the methods transform the search into a sequence of unconstrained subproblems where the solution in each subproblem can act as a starting point for the next subproblem. This will result in asymptotic convergence if each unconstrained subproblem in the sequence is solved optimally. Nonetheless, this is difficult to achieve in practice, given only a finite amount of time to solve each subproblem. Consequently, approximation in the result is still unavoidable. Various dynamic strategies have been discussed in [68, 76, 100, 55, 9]. Dynamic-penalty strategies suffer from the limitation that besides requiring domain specific knowledge, most of these heuristics have difficulties in finding a feasible region or in maintaining feasibility for nonlinear constraints and get stuck easily in local minima [99, 132, 129].

### 2.1.3 Local Move and Neighborhood Function

Local search methods rely on local probes or perturbations to generate candidate trial points and advance their search trajectories.

Let \( S \) be a finite set of user-defined trial points.

**Definition 2.1.1** A *move* is define as, \( Move(s, s') \) where \( s, s' \in S \) and \( s \neq s' \).

A move, \( Move(s, s') \) is said to take *one step* if it requires only a single user-defined operation to institute the move from \( s \) to \( s' \).

Neighbor points can be defined as follow:

**Definition 2.1.2** \( N_{dn}(s) \), the neighborhood points of \( s \), is a finite set of user-defined points \( s' \in S \) such that \( s' \) is reachable from \( s \) in one step i.e. \( Move(s, s') \).

The key to a good local search algorithm is to use a good neighborhood function,
a user-defined function that selects the neighbor for next perturbation. Unfortunately, selecting good neighbors is a heuristic\textsuperscript{1} process hence is problem dependent. Consequently, there is no universal neighborhood function. In spite of this, most neighborhood heuristic functions fall into the following categories:

- Changing Value (Random) - a new value is generated randomly (random-restart heuristic)
- Flipping - suitable for problem with binary value (e.g. Boolean satisfiability problem)
- Increasing/Decreasing (linear or based on a functor value)
- Averaging - move to a state which is the average value of the neighboring states.
- Swapping Value - swapping value between two neighboring states.

**Definition 2.1.3** An *Improving Move* is a move to neighbor that improves the objective value, i.e. \{Move(s, s') | f(s') < f(s)\}

**Definition 2.1.4** A *Sideways Move* is a move to neighbor that neither improves nor worsen the objective value, i.e. \{Move(s, s') | f(s') = f(s)\}

Greedy approaches are often applied along with the heuristic technique (such as that in definition 2.1.3). Furthermore, it is shown that by allowing sideways move, one can improve the quality of search result significantly (e.g. see [117]).

### 2.1.4 Local Minima/Maxima

A local Minima/Maxima\textsuperscript{2} is a non-solution state in the search space that none of the valid move shall improve the search state (objective function) further. Formally, a local minima can be defined as follows:

**Definition 2.1.5** Point \( s \) is the local minimum if \( \forall s' \in N_{dn}(s), f(s) < f(s') \) where \( N_{dn}(s) \) are the set of neighbors for \( s \) (as defined in 2.1.2) and \( f(s) \) is the objective function for \( s \).

\textsuperscript{1}Heuristic is a word derived from Greek word heuriskein (εὑρίσκειν) which means to find [112]. It is a technique based on rule of thumb. Unlike algorithms, heuristics do not guarantee optimal, or even feasible, solutions and are often used with no theoretical guarantee.

\textsuperscript{2}If the aim is to minimize the objective function, then the term “minima” is used and vice versa.
2.1. Stochastic Local Search

Local minima are problematic because they form a trap in the search space that locks the search algorithm at a static position. Figure 2.1 depicts types of local minimum structures. A trap is a “well” of local minima and is difficult to deal with.

2.1.5 Meta-Heuristics

Much effort has been devoted to develop local search techniques that can “escape” local minima. These techniques are known as meta-heuristics, a term that combines the Greek prefix “meta” (“beyond”) with heuristic to give the sense of “higher level” methods of controlling the behavior of the search heuristic. Meta-heuristics have developed dramatically since their inception in the early 1980s. They have had widespread success in attacking a variety of practical and difficult combinatorial optimization problems.

Formally, Osman and Kelly [110] defined a meta-heuristic as “an iterative generation process which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the search space, learning strategies are used to structure information in order to find efficiently near-optimal solutions.” For that reason, meta-heuristics include, but are not limited to genetic algorithms [67], greedy random adaptive search procedures (GRASP) [42], neural networks [70, 71],
2.1. Stochastic Local Search

non-monotonic search strategies [36, 84], problem and heuristic space-search [122], simulated annealing [82], tabu search [53, 54], threshold algorithms and their hybrids [37, 84]. In the following, we briefly describe the basic principles of some of the most common meta-heuristic techniques.

**Random Restart**

Random restart is the simplest of the local search meta-heuristic techniques. It is often used in conjunction with the greatest descent heuristic. A formal discussion of random restart and greatest descent has been undertaken by Michalewicz and Fogel [98] and Hromkovic [72]. The essence of the technique is to minimize the objective function. The algorithm starts with an initial solution generated through some randomized means. Should the neighboring solution have a better value for the objective function, then the neighboring solution becomes the current solution. Otherwise the current solution remains the current solution. The process then repeats for a number of times, each attempts to move towards better neighboring solutions. The process stops after a predetermined number of iterations, or the conditions for optimality are met. If an optimal solution is not found after the predetermined number of iterations, the algorithm is assumed to have stuck in local minima. In order to escape local minima, the algorithm simply restarts from a fresh initial solution hoping that the new initial solution may lead to a completely new search state hence towards the optimal solution. Algorithm 2.2 depicts the random restart meta-heuristic algorithm.

**Algorithm 2.2** Random Restart Meta-Heuristic Algorithm

1. Procedure RandomRestart
   2. For i = 0 To MaxRestart Do
   3. s := GenerateInitialSolution()
   4. s* := s
   5. For j = 0 to MaxTries Do
   6. Move(s, s’)
   7. If f(s’) < f(s) Then
   8. s* := s’
   9. s := s’
   10. End If
   11. End For
   12. End For
   13. Return s*
   14. End RandomRestart

The major weakness is that it allows no uphill (non-improving) or sideways
2.1. Stochastic Local Search

moves. The success of the method depends upon the starting solution being near a good local minimum. It is particularly poor for problems that have many local minima. A further weakness is that, because of the randomized nature, it may miss some good solutions that are not ever generated or not generated at an appropriate time. Moreover, there is no mechanism to exclude the valuation that have been found infeasible. Consequently, the successive perturbation may obliviously visit the same infeasible solutions repeatedly.

Simulated Annealing

In 1983, Kirkpatrick et al. [82] proposed a local search method using Metropolis Monte Carlo simulation [97] to find the lowest energy (most stable) orientation of a system. Their method is based upon the procedure used to make the strongest possible glass. This procedure heats the glass to a high temperature so that the glass is a liquid and the atoms can move more freely. The temperature of the glass is slowly lowered so that at each temperature the atoms can move enough to begin adopting the most stable orientation. If the glass is cooled slowly enough, the atoms are able to ”relax” into the most stable orientation. This slow cooling process is known as annealing, and so their method is known as Simulated Annealing.

We have seen that the main problem with random restart is that it does not allow uphill (non-improving) and sideways move. When search reaches the local minima, there is no way to improve further except by restarting from a fresh solution. The intuition behind the introduction of Simulated Annealing is to overcome this problem by allowing some uphill moves as a mean to escape local minima. The introduction of uphill moves may break out of local minima and allow the search to move to a superior point of the search space.

It is assumed that during the initial state, the search tends to get stuck in poor local minima, hence the probability of allowing uphill moves should be high. As search moves towards the end, most likely the search should have settled at some high quality local minima (or optimal solution) and therefore uphill moves should be restricted. The variant from a high probability of accepting uphill moves to a low probability of allowing such moves resembles the annealing process of the high temperature state (when atoms can move more freely) moving to a low temperature state (when movement of atoms are restricted) respectively.

In the implementation of Simulated Annealing local search algorithm, an improving move is always accepted. A worsening move is accepted depending on a probability, $P$, where $P$ is set according to the same criterion set by Metropolis
2.1. Stochastic Local Search

et al. [97] in the original glass annealing process, i.e.:

\[ P = e^{(-\Delta e/T)} \]  

(2.2)

\( \Delta e \) is the offset of the value of objective function, \( f \) of the current solution, \( s \) and the neighboring solution, \( s' \). Therefore

\[ \Delta e = f(s) - f(s') \]  

(2.3)

\( T \) is the temperature parameter which controls the acceptance probability. A high value of \( T \) results in high acceptance probability and vice versa. The rate that \( T \) is reduced is the cooling schedule. The most basic (and common) cooling schedule for Simulated Annealing is geometric cooling where \( T \) decreases according to the equation:

\[ T = \alpha T \]

\[ 0 < \alpha < 1 \]  

(2.4)

Algorithm 2.3 depicts the Simulated Annealing meta-heuristic algorithm. The termination condition in Line 5 can be of various forms, such as a pre-fixed number of iterations, or the number of moves accepted [82], or when the objective function at the state of equilibrium, i.e. the value of objective function oscillates around a particular value [27].

**Tabu Search**

Tabu Search [53, 54] is an extension to many existing local search techniques, such as random walk or simulated annealing. What distinguishes it from other methods is that in order to prevent cycling, the Tabu Search algorithm maintains a “tabu list”. In its simplest form, a tabu list contains the solutions that have been visited in the recent past (less than \( n \) moves ago, where \( n \) is the size of tabu list). Moves to those solutions are suppressed.

Tabu lists will allow the search to become much more effective, although they raise new problems. Maintaining the tabu list requires additional computing resources. As the size of the tabu list increases, tracking a tabu item can be difficult. A simple solution is to limit the size of the tabu list. But limiting the size of the list may defeat its purpose. Furthermore, with forbidding an attribute as tabu, typically more than one solution is declared as tabu. Some of these solutions that must now be avoided might be of excellent quality and have not yet been visited. To overcome this problem, aspiration criteria are introduced which allow overriding the tabu state of a solution and thus include it in the allowed set. Nonetheless, to
2.1. Stochastic Local Search

Algorithm 2.3 Simulated Annealing Meta-Heuristic Algorithm
1. Procedure SimulatedAnnealing
2. \( T := \text{StartTemperature} \)
3. \( s := \text{GenerateInitialSolution()} \)
4. While \( T > \text{End Temperature} \)
5. While Termination Condition Not Met
6. Move\((s, s')\)
7. If \( f(s') < f(s) \) Then //Always accept improving move
8. \( s := s' \)
9. Else
10. \( \Delta e = f(s) - f(s') \)
11. \( P = \exp(-\Delta e/T) \)
12. \( R = \text{Random}(0, 1) \)//R = rand number between 0 and 1
13. If \( R < P \) Then //Accept worsening move
14. \( s := s' \)
15. End If
16. End If
17. End While
18. \( T = \alpha \times T \)//Decrease temperature
19. End While
20. Return \( s \)
21. End SimulatedAnnealing

check if the aspiration criteria are met can be an expensive endeavor.

Iterated Local Search

Iterated local search [90] is a meta-heuristic that is simple and yet effective. The idea of iterated local search came from the realization that in the quest for greater performance, modern meta-heuristic techniques have become more dependent on problem-specific knowledge, hence lose their simplicity and generality. To counter this, iterated local search reinvents the modularity by decomposing a meta-heuristic algorithm into a few parts, each with its own specificity. In particular, it has a general purpose part, while any problem-specific knowledge built into the meta-heuristic would be restricted to another part.

The essence of the iterated local search can be given as follows: Given an initial solution \( s_0 \), first the problem is perturbed with a generic perturbation procedure leading to intermediate solution \( s' \). Next, a problem specific local search procedure is applied onto \( s' \) to return an improved solution \( s'' \). Finally if \( s'' \) passes an acceptance test, \( s'' \) becomes \( s \), i.e. we accept \( s'' \) as the starting point for next perturbation. The algorithm is outline in Algorithm 2.4

The LocalSearch procedure (Line 5) can be of any local search meta-heuristic tech-
2.1. Stochastic Local Search

Algorithm 2.4 Iterated Local Search Meta-Heuristic Algorithm

1. Procedure IteratedLocalSearch
2. \( s := \text{GenerateInitialSolution()} \)
3. While termination condition not met
4. \( s' = \text{GenericPerturbation}(s) \)
5. \( s'' = \text{LocalSearch}(s') \)
6. If AcceptanceCriteria\( (s'') \) Then
7. \( s = s'' \)
8. End If
9. End While
10. Return \( s \)
11. End IteratedLocalSearch

niques, such as Random Walk, Simulated Annealing or Tabu Search. Iterated local search has been shown to produce superior result than using random restart, for example see \([75, 90, 111]\).

Iterated Local Search tries to escape from local minima by applying perturbations to the current local minimum. The perturbations determine the intensification of search whereby strict acceptance criterion forces the search to improve on the current local minima. On the other hand, loose acceptance criterion accepts all new solution \( s'' \). This will allow the search to explore as many local minima as possible. The success of Iterated Local Search is highly dependent on the perturbation schemes, acceptance criterion and the choice of the local search techniques. Unfortunately, there is no clear theoretical guideline in the selection process. Consequently, many of the implementation turns out to depend mainly on experimentation.

Evolutionary/Genetic Algorithms

Evolutionary algorithm is a generic term used to indicate any population-based meta-heuristic optimization algorithm that uses mechanisms inspired by biological evolution, such as reproduction, mutation, recombination, natural selection and survival of the fittest. Genetic algorithms (GAs)\([67]\) are one of the most popular classes of evolutionary algorithms. The idea was invented in 1960 by Holland \([67]\) but came to popularity much later after one of his students, Goldberg, used genetic algorithms to solve a gas-pipeline problem \([56]\). Ever since then, thousands of research and journals papers and books have been written on this subject with many formal conferences dedicated to address the issue.

GAs use the principle found in biological genetic evolution, where the process aims to evolve a set of solutions into a superior set of solutions using the principle of survival of the fittest. By encoding problems to resemble biological genes, and
2.1. Stochastic Local Search

applying the principles found in genetic processes (such as reproduction, cross-over, mutation, competition and selection), it is anticipated that computer problem solving will behave like biological evolution where through the survival of the fittest, an inferior solution will eventually “evolve” into a superior solution.

A typical GA has two components to be defined: (1) a genetic representation of solutions, (2) a fitness function to evaluate them. A common representation is an array of binary values. Arrays of other types and structures can be used in essentially the same way. To facilitate our understanding of how GA work consider a simple problem to find the minimum value of the function $f(x) = x^2 - 3x$. For simplicity, we assume that $x$ only takes integer value and is in the range of $-7 \leq x \leq 7$. Thus the chromosome can be build with four genes (4 bits). The first bit indicates the sign of the number (i.e. 0 for positive number and 1 for negative number) and the last three bits are the binary representation of the number. The encoding of gene is shown in Table 2.1. The fitness function is simply $f(x) = x^2 - 3x$ where the fittest value is when $f(x)$ is the minimum.

<table>
<thead>
<tr>
<th>Integer</th>
<th>Gene Code</th>
<th>Integer</th>
<th>Gene Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>-1</td>
<td>1001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>-2</td>
<td>1010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>-3</td>
<td>1011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>-5</td>
<td>1101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>-6</td>
<td>1110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>-7</td>
<td>1111</td>
</tr>
</tbody>
</table>

Table 2.1: Gene encoding of variable $x$

Once we have defined the genetic representation of the problem and the fitness function, we can apply GA to solve the problem. A typical GA has the following steps:

1. Randomly generate an initial population of chromosomes of size $n$

   $x_1, x_2, ..., x_n$

2. Calculate the fitness of each individual chromosome

   $f(x_1), f(x_2), ..., f(x_n)$

3. Select a pair of chromosomes for mating from the current population. Parent
chromosomes are selected with a probability related to their fitness. Highly fit chromosomes have a higher probability of being selected for mating than less fit chromosomes.

4. Mating produces a pair of offspring chromosomes. The mating process is the cross-over of the pair of chromosomes selected for mating. Mutation occurs with the probability $p$ (very small probability).

5. Place the created offspring chromosomes in the new population.

6. Repeat Step 3 until the size of the new chromosome population becomes equal to the size of initial population, $n$.

7. Replace the initial (parent) chromosome population with the new (offspring) population.

8. Repeat step 2 until termination criterion is satisfied.

In the given problem, let us say we have a population $n = 4$. In the first step, we generate an initial population $X = \{-7, -3, 4, 5\} \equiv \{1111, 1011, 0100, 0101\}$. The fitness for each chromosome is $\{f(-7) = 70, f(-3) = 18, f(4) = 4, f(5) = 10\}$. Two chromosomes are selected for mating, lets say $-3$ and $4$. The crossover process is shown in Figure 2.2. The crossover can be at any point in the chromosome. In this example, crossover happens at bit 3. The crossover results in two new chromosome, $0$ and $7$, each with fitness $f(0) = 0$ and $f(7) = 28$. This process is repeated until termination condition is met and the fittest chromosome is chosen as the optimal solution.

This example represents the simplest and yet naive implementation of GA. In reality there are many parameters and features that need to be tuned by the user of a genetic algorithm: the size of the pool of solutions, the method of mutation, the method for creating offspring solutions, the way that solutions are chosen to mutate or produce offspring, and the way that the pool of solutions is culled. Variants of genetic algorithms also exist, such as the memetic algorithms, introduced by
Moscato and Norman [104]. A memetic algorithm works very similarly to a genetic algorithm, except that the new solutions generated (through mutation or offspring) are improved in some way before they are placed in the pool of solutions. The improvement procedure is often a local search algorithm such as hill climbing, simulated annealing, tabu search, etc. This has the effect of only creating good quality new solutions, which reduces the search space to the set of local minima.

GAs have the advantage over other local search methods that they are not tied to one initial solution. The presence of many different solutions allows for greater diversity in the search of the solution space. However GAs tend to be slower than other meta-heuristics because more solutions are maintained, and because there are many more parameters and choices that alter the quality of the solutions produced by these methods. Moreover, successful implementation of GAs often require problem-specific information to enhance their performance making GAs less attractive and more complicated to implement. Some criticism on GA technique can be found in [113].

2.1.6 Beyond Meta-Heuristics

In general, it is not known that there exist a universal meta-heuristic technique that performs well on all problems. For that reason, Burke et al. [15] introduced the term Hyper-Heuristic; i.e. heuristic to choose the suitable type of heuristic. Hyper-heuristics techniques are developed so that an algorithm can intelligently adopt the best heuristic techniques to the problem that it is trying to solve. In essence the idea is to use a number of different heuristics together, so that the actual heuristic applied may differ at each decision point. An example of the use of hyper-heuristics is given in [41], in which a genetic algorithm (GA) is applied to open-shop scheduling problems, to evolve the choice of heuristic to apply whenever a task is to be added to the schedule under construction. The process was called ‘evolving heuristic choice’. Another illustrative example is given in [123], for solving some large real-world examination timetabling problems.

2.2 Invariants: One-Way-Solver

As we have presented earlier, the idea behind local search is to confine search only to the neighborhood of a state. By doing so, it allows local search to be efficient in terms of memory consumption hence allow search to be performed on large or infinite (continuous) state spaces for which systematic search algorithms are unsuitable. In order to ensure that local search returns a desirable result, at each
2.2. Invariants: One-Way-Solver

state, the algorithm will evaluate the search condition based on an objective function \( f(x) \) and move to neighbor state when the move results in improvement of the objective function towards the search goal. The bottleneck of local search lies in the efficiency of computation of \( f(x) \). Consider the implementation of university course timetable scheduling program using local search. Assuming that the objective function \( f(x) \) is computed by summing up the penalty value for violation of \( n \) constraints in each course. If \( c_i \) is the penalty value for course \( i \), then \( f(x) \) can be computed as 
\[
f(x) = \sum_{i=1}^{n} c_i.
\]
If an instance of timetabling contains 100 courses, then making each neighbor move will require the local search algorithm to recompute the sum of violated constraints for all the 100 courses in order to derive the state of the search as a result of the new local move. This is obviously computationally intensive. Nonetheless, neighbors are often closely related, hence a move does not often instigate major change in the structure.

Efficiency in local search can be achieved if incremental algorithms can be used to maintain the data structure of the local search. Incremental algorithms only recompute the areas (variables) affected by the most recent move hence saving unnecessary steps in recomputing values that indeed never change. More formally, given a computation of \( g(x) \) with result \( r \) and a new input \( x \oplus \delta x \) to the program \( g \), an incremental algorithm \( g \) computes the result of \( g(x \oplus \delta x) \) by making use of \( r \), avoiding computations previously done by \( g(x) \) \[89\].

Unfortunately, incorporating incremental algorithms into local search is one of the most tedious and time-consuming task. The reason is because these algorithms maintain sophisticated data structures in order to decide where to move next efficiently. In addition, for the algorithms to be truly efficient, some problem-dependent information needs to be used to guide the decision process. The situation is worsened by the fact that current implementation of local search in problem solving is still art rather than science, which may involve considerable experimentation. It is clear that abstraction of local search may bring significant benefits. Recent research has started addressing these challenges. The first phase of the proposals focuses the attention to frameworks that make it easy to experiment with different meta-heuristics (e.g. see \[6, 43, 115\]). The next phase focuses at the high-level modeling issues and incrementality. The introduction of LOCALIZER \[103\] in 1998 is a significant revolution in the field where it is, the first modeling language for local search that uses invariants to specify incremental algorithm declaratively. The idea further developed into Constraint-based local search \[101\], of using constraints to describe and control local search. New modeling and control abstractions appeared later \[63, 64, 65\] and are included in the programming language COMET \[102\].
2.2. Invariants: One-Way-Solver

2.2.1 Invariants

The concept of Invariants was introduced by Michel and Hentenryck [103] in 1998. An Invariant is an expression in the form: \( v = f(x) \) where the solver guarantees that at any time during the computation, the value of variable \( v \) is the value of function \( f(x) \). For example, \( v = \text{sum}(x_1, x_2, x_3, ..., x_n) \). When a value \( x_i \) changes, the value of \( v \) will be updated to the new sum value. Since an invariant ensures one side of the operand is always consistent with the other side of the operand, it is also called One-Way-Solver. The solver uses efficient incremental algorithms to maintain the invariant and propagate updates in constant time. The benefit is that it frees the user from the tedious implementation of complex incremental data structures by allowing the user to define what needs to be maintained but not considering how to do so. Invariants are specified in a declarative way using standard arithmetic, Boolean connectivity, and aggregating functions such as sum, min, max, alldifferent, allsame, subset, etc thus simplifying development of complex incremental algorithms.

2.2.2 Arithmetic Invariants

In the most primitive form, invariants exist as basic arithmetic operations (addition, subtraction, multiplication and division). Given \( v = f(x) \), where \( f(x) \) is an arithmetic operation, the invariant ensures that at any time during the computation, \( v \) is the result of \( f(x) \).

Data Structure

An arithmetic expression is represented by a directed acyclic graph whose topological order allowing updates to be propagated only to the nodes that are affected by the changes. For instance, expression \( f(x) = A * B + C * (C + D) \) can be represented as in Figure 2.3.

If the value of \( A = 3 \), \( B = 4 \), \( C = 5 \) and \( D = 7 \), the equation can be evaluated as in Figure 2.4.

If the value of \( C \) changes to 3, only three nodes require recomputation. Figure 2.5 shows the process of updating the graph in topological order when the value of \( C \) changes to 3.

Given a significantly large arithmetic equation, using the topological order will be able to save considerable computation time in the process of maintaining the invariant.
Addition and Subtraction

In maintaining the arithmetic invariant, addition and subtraction operations can be generalized into single addition operation. The reason is because subtraction is indeed an addition with a negative value. For example $A - B$ is equivalent to $A + (-B)$, i.e. a positive $A$ plus a negative $B$. We can represent an invariant variable with a structure consisting of an integer to store the absolute value of the variable and a Boolean value representing the sign of the value. Figure 2.6 shows the structure representing an invariant variable for addition and subtraction operations.

Having only to deal with the addition operation, update can be easily computed incrementally. If $v = f(x)$ where $f(x)$ is an expression with only the addition operation, if $x_i$ in $f(x)$ changes from $x_i \rightarrow x'_i$, then the new result of $f(x)$, $v'$ can be computed incrementally as:

$$v' = v - x_i + x'_i \quad (2.5)$$

Let

$$v = g(x) + x_i$$
$$g(x) = v - x_i \quad (2.6)$$
2.2. Invariants: One-Way-Solver

\[
f(x) = 42
\]

\[
\begin{array}{c}
\text{struct var} \\
\{ \\
\quad \text{int value; //store the value of the variable} \\
\quad \text{bool sign; //store the sign of the variable} \\
\}
\end{array}
\]

Figure 2.5: Incremental update of equation tree

Figure 2.6: Structure representing the invariant variable for addition and subtraction operations

If the value \(x_i\) change to \(x'_i\), then

\[
v' = g(x) + x'_i \tag{2.7}
\]

By replacing \(g(x)\) with (2.6), we get

\[
v' = v - x_i + x'_i
\]

**Example:** \(f(x) = A + B - C + D\)

The equation can be rewritten as: \(f(x) = A + B + C' + D\) where \(C' = -C\)

Let \(A = 10, B = 3, C = 5, D = 7\), i.e. the expression: \(10 + 3 - 5 + 7\) which evaluates to \(10 + 3 + (-5) + 7 = 15\). If the value of B is changed to 5, the new result is computed incrementally as:

\[
v' = 15 - 3 + 5 \\
v' = 17
\]

If the value of C is changed to 10, the new result is computed incrementally as:

\[
v' = 15 - (-5) + (-10) \\
v' = 10
\]

The process requires one step and has constant running time irrespective of the number of variables.
2.2. Invariants: One-Way-Solver

Multiplication and Division

We have generalized addition and subtraction into a single addition operation. We can also do the same with multiplication and division as single multiplication operation. The reason is because division is actually the multiplication of the inverted value. For instance $9 \div 3$ is equivalent to $9 \times \frac{1}{3}$.

If $v = f(x)$ when $f(x)$ is an expression with only multiplication operation, if $x_i$ in $f(x)$ changes from $x_i \rightarrow x'_i$, then the new result of $f(x)$, $v'$ can be computed incrementally as:

$$v' = \left( \frac{v}{x_i} \right) \times x'_i$$

Where $x_i \neq 0 \quad (2.8)$

Let

$$v = g(x) \times x_i \quad (2.9)$$

If the value $x_i$ change to $x'_i$, then

$$v' = g(x) \times x'_i \quad (2.10)$$

By replacing $g(x)$ with (2.9), we get

$$v' = \left( \frac{v}{x_i} \right) \times x'_i$$

where $x_i \neq 0$

Example: $f(x) = A \times B \div C \times D$ Let $A = 5$, $B = 6$, $C = 2$, $D = 4$

$$f(x) = 5 \times 6 \div 2 \times 4 \text{ which can be written as}$$

$$f(x) = 5 \times 6 \times \frac{1}{2} \times 4$$

$$= 60$$

If the value of $A$ is changed to 10, the new value of $f(x)$ is calculated incrementally as

$$f(x) = \frac{60}{5} \times 10$$

$$= 120$$

If the value of $C$ is changed to 3, the new value of $f(x)$ calculated incrementally as

$$f(x) = \frac{60}{2} \times 3$$

$$= 90$$

One exception is that if the value of $f(x)$ before the change is 0, we cannot perform incremental computation. The entire equation has to be recomputed naively.
2.2. Invariants: One-Way-Solver

```c
struct var {
    int value; //store the value of the variable
    bool sign; //store the sign of the variable
    bool invert; //indicates if value is inverted
}
```

Figure 2.7: Structure representing the invariant variable for multiplication and division operations

To allow multiplication and division, we have to extend our variable structure to have an additional field to indicate if the value of the variable should be inverted (for the case of division). The new structure is shown in Figure 2.7.

Putting an Equation Together

Common arithmetic operations consist of the mixture of the four arithmetic operations. In the last two sections, we have shown how basic arithmetic operations can be generalized into two basic operations, i.e. addition and multiplication.

Definition 2.2.1 \( f(x) \) is a primitive equation if it contains only one type of operation i.e. addition expression or multiplication expression.

Definition 2.2.2 \( f(x) \) is a composite equation if it is an aggregation of more than one primitive equation.

Any mathematical expression is either a primitive equation or a composite equation. When constructing an equation tree, we break composite equations into a series of primitive equations organized by its topological order. For each primitive equation, we can incrementally update the value of the equation when one or few of the values changed.

Example:

\[
f(x) = A \times B + C - A \times D - D \div E
\]

\( f(x) \) can be rewritten as

\[
W = X + C + Y + Z \quad \land \\
X = A \times B \quad \land \\
Y = A' \times D \quad \land \quad A' = -A \quad \land \\
Z = D' \times E' \quad \land \quad D' = -D \quad \land \quad E' = \frac{1}{E}
\]

whereby \( f(x) \) is a composite equation and \( W, X, Y, Z \) are the primitive equations of \( f(x) \). The equation tree for \( f(x) \) is shown in Figure 2.8.
Once we have the equation tree with all the nodes as primitive equations, each node can be recomputed using the incremental computation models (2.6) and (2.9) for addition and multiplication nodes respectively. The values of $W$, $X$, $Y$ and $Z$ are computed, hence they are called auxiliary variables. On the other hand the values of $A$, $B$, $C$, $D$, and $E$ are assigned hence they are called primary variables. All the primary variables appear at the leaves of the tree.

**Definition 2.2.3**  $v$ is a primary variable if its value is given by assignment (i.e. independent from primitive or composite equation).

**Definition 2.2.4**  $v$ is an auxiliary variable if the value is computed either from a primitive or composite equation.

**Incremental Update**

Whenever a primary variable value is changed, the auxiliary variables are updated so that each auxiliary variable has the value resulting from the changes. The update can be done incrementally. We say that a node $N$ require consideration if some variable appearing (explicitly) in $N$ undergoes change during the update. We use a priority queue (heap) ordered by the depth of the tree to process the nodes that require consideration. In this way, we only visit the variables which require consideration. Moreover the priority queue ensure that each node requiring consideration is visited once and only once. If we assume that the tree has a bounded number of arguments, then the time required for this update operation is $n \log n$ where $n$ is the number of nodes requiring consideration because each heap operation can take $O(\log n)$ time.

Consider the problem in the Figure 2.9. Let $P$, $Q$, $R$, $S$ and $T$ being primitive equations and $A$, $B$, $C$ and $D$ are the primary variables. If the value of $B$ changes and recalculation of $R$ is not delayed until the recalculation of $P$ and $Q$ are completed, $R$ would be recalculated twice. Similarly, it will result $T$ to be recalculated twice as well. In a huge tree where many changes occur simultaneously, it can significantly reduce the efficiency of update process.
2.2. Invariants: One-Way-Solver

Let's consider the update process in more detail. Assume this time that the value of $B$ and $C$ have changed simultaneously. The process of update is as follow:

1. Since the value of $B$ has changed, it becomes the node that require consideration. All the parents of $B$ (auxiliary variable nodes that are dependent on $B$, i.e. $P$ and $Q$) also become nodes that require consideration and therefore are inserted into the priority queue (note that $B$ are primary variable node, thus they do not need to go into the queue) (see Figure 2.10a).

2. Next, variable $C$ is considered. All its parents are inserted into the queue. The parents of $C$ are $Q$ and $S$. However, since $Q$ is already in the queue (as a result of its dependency on $B$) we do not insert it into the queue. When $S$ is inserted into the queue, it will be at the top of the queue (i.e. the top priority) (see Figure 2.10b). The reason is because $S$ has the highest level in the equation tree.

3. When there is no more primary variables that require consideration, we begin the recomputation process. The first node to be considered is $S$. This is done by dequeuing $S$ from the queue. Recomputation is done using the formula in (2.6) and (2.9) for addition and multiplication operations respectively. If recomputation results in new value of $S$, we enqueue all the parents of $S$ that do not appear the queue into the queue. However if the value of $S$ does not change after recomputation, we do not perform the enqueue because no further recomputation of the parents is required. In the case of $S$, since the parent of $S$ already in the queue, we do not perform any enqueue (see Figure 2.10c).

4. Next we dequeue $P$ for recomputation. Assuming that the recomputation of $P$ results in new value, we enqueue the parent of $P$ (i.e. $R$) into the queue (see Figure 2.10d). Since the level of $R$ is smaller than that of $Q$, it will be inserted after $Q$.

5. Next, we dequeue $Q$ for recomputation but we do not enqueue the parent of $Q$ because $R$ is already in the queue (see Figure 2.10e).

6. Next, node $R$ will be considered and node $T$ will be inserted into the queue (see Figure 2.10f).
7. Finally, node $T$ is dequeued for consideration. The update ends when the priority queue becomes empty.

We have observed in the example above that by using the priority queue, the node with the highest level will be considered first and each node in the equation tree is considered once. Algorithm 2.5 represent the incremental update process of the equation tree.

**Don’t Care**

In certain conditions, we can ignore updating the values in the tree despite the fact that the value of the variables in the expression has changed. Consider the example $f(x) = A \times B + C \times (C + D)$. The equation tree is shown in Figure 2.11. Notice that if the variable $C$ is 0, the right sub-tree will always evaluate to zero irrespective of the value of $D$. In such a situation, we can mark all the nodes in the right sub-tree as don’t care. Until the condition of don’t care is lifted (i.e. $C \neq 0$), we do not have to consider any update at the right sub-tree. Nevertheless, when the don’t care situation is lifted, the entire node from the highest point of don’t care has to be
2.2. Invariants: One-Way-Solver

Algorithm 2.5 Incremental update algorithm

1. PriorityQueue Q
2. For all A in parent of primary variables that change
3. Q->Enqueue(A);
4. End for
5. While Q is not empty
6. N = Dequeue()
7. M = Recompute(N)
8. If M != N Then //value has change
9. For all B in parent of N
10. If B Not In Q Then
11. Q->Enqueue(B)
12. End If
13. End For
14. End If
15. End While

\[ f(x) \]

\[ A \]
\[ B \]
\[ C = 0 \]
\[ D \]

Figure 2.11: Don’t care nodes

recomputed from scratch as incremental computation is no longer possible because the values at these nodes are outdated.

2.2.3 List Invariants

We have so far discussed arithmetic invariants. A list is another class of data structure of extreme importance in computer application because it is used to hold collections of values with similar data type. Unfortunately it is also one of the most computational intensive data structure to maintain. The One-Way-Solver provides efficient ways to maintain invariant in list data structures. There are several list invariants considered in our implementation.

Let \( L \) be a list of numeric values,
2.2. Invariants: One-Way-Solver

*Sum Invariant*

\[ v = \text{Sum}(L) \quad (2.11) \]

is an invariant which ensures that at any point of the computation, the value of \( v \) is always the sum of the values in the list. Implementation of the sum invariant is similar to the additional arithmetic invariant, i.e. update of \( v \) is computed as (2.6).

*Min and Max*

\[ v = \text{Min}(L) \quad (2.12) \]

is an invariant which ensures that at any point of the computation, the value of \( v \) is always the smallest value of the list.

\[ v = \text{Max}(L) \quad (2.13) \]

is an invariant which ensures that at any point of the computation, the value of \( v \) is always the largest value of the list.

*Min* and *Max* invariants are implemented using a min-heap and max-heap respectively. In a heap, the value of the root (or more commonly known as peak) is always the smallest/largest value. When a value changes, if the new value is smaller (for the case of *Min*) or larger (for the case of *Max*) than the peak node, re-heap is conducted to reposition the smallest or largest value at the peak node.

One problem of using heap to maintain smallest or largest value in a list is that other than the peak value, all other values do not have a fixed position. For instance, in Figure 2.12, given a list 3, 4, 5, 6, 7, 8 in a *Min* heap, (a) and (b) are both valid heap structures. As long as the parent nodes have value larger than that of their children, and the heap tree is complete (i.e. filled from top-down, left-right), the heap is valid. The problem arises in locating an internal node of the heap. For example, if the value of 8 changes to 2, in order to perform re-heap, we must first locate the position of 8 and there is no other way than to search through the heap starting from the peak node. This is obviously inefficient.

In order to overcome the problem, we must have a pointer for each value in the *Min/Max* list to point to their respective location in the heap. In case their value changes, we can locate the variable in the heap using the pointer and perform re-heap in an efficient manner.
2.2. Invariants: One-Way-Solver

![Min heap for list 3, 4, 5, 6, 7, 8](image)

**alldifferent** Invariant

Literally alldifferent constraint means no two elements can take the same value. Unlike a complete solver (which commonly found in constraint programming language), alldifferent invariant in One-Way-Solver does not ensure that the elements take different value, but rather it returns the state of alldifferent in the list. Given

\[ v = \text{alldifferent}(d_1, d_2, ..., d_n) \]  \hspace{1cm} (2.14)

alldifferent invariant ensures that at any point of the computation, the value of \( v \) is the score of the items in the list that do not conform to the alldifferent constraint. The computation of the score, \( v \) is shown in the next section.

**Computing alldifferent Score**

Let \( L \) be list of variables with finite domains \( D(x) \), a naive way of computing the penalty is to search the list for repeated values. For each repeating value, it incurs 1 penalty point (score), otherwise 0. The algorithm requires two straight loops and is \( O(n^2) \). We count the number of times an element, \( L_i \) appears in the list. If an item \( L_i \) appears \( m \) times in \( L \), then the count \( c_i = m \). The score for element \( L_i \), \( r_i \) can be computed as:

\[ r_i = c_i \times (c_i - 1) \]  \hspace{1cm} (2.15)

If \( L_i \) exists \( c \) times in \( L \), then there are a total of \( \sum_{i=0}^{c-1} i \) pairs of conflicting \( L_i \). Each pair will incur 2 points and it is known in mathematical series that

\[ \sum_{i=0}^{n} i = \frac{n(n+1)}{2} \]  \hspace{1cm} (2.16)
Therefore the score for $\sum_{i=0}^{c-1} i$ pairs of conflicting value is

$$\sum_{i=0}^{c-1} i \times 2 \equiv \frac{(c-1)((c-1)+1)}{2} \times 2 \equiv c \times (c - 1)$$

We can define the score for \textit{alldifferent}, $R$ as:

$$R = \text{alldifferent}(L) \Rightarrow \forall r_i \in L$$

$$\sum_{i=1}^{n} r_i$$

$$R = \sum_{i=1}^{n} c_i \times (c_i - 1) \quad (2.17)$$

For example, let list $L$ contain the following items:

$$5 \quad 3 \quad 2 \quad 5 \quad 3 \quad 1 \quad 6 \quad 7$$

In the above example, $L$ will produce penalty score list, as follow:

<table>
<thead>
<tr>
<th>Item ($L_i$)</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count ($c_i$)</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Score ($r_i$)</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The total penalty incur by $L$, $R$ is the sum of score for $r_i$

$$R = \sum_{i=1}^{n} r_i \quad (2.18)$$

By replacing $r_i$ in 2.18 with 2.15 we obtain

$$R = \sum_{i=1}^{n} c_i \times (c_i - 1) \quad (2.19)$$

where $c_i$ is the number of times item $L_i$ repeats in $L$.

In other words, we can compute the score by knowing how many times an item appeared in the list. In the above example, the total penalty incurred, $R$ is 4. If we change 6 to 5, the new penalty point for $R$ is 8 and the corresponding score table is shown below:
2.2. Invariants: One-Way-Solver

<table>
<thead>
<tr>
<th>Item ((L_i))</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count ((c_i))</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Score ((r_i))</td>
<td>6</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Incremental Implementation of alldifferent**

The cost of maintaining an alldifferent is to keep track of the changes in \(R\) as a result of update in \(L\). When a value, \(L_i\) changes from \(v \rightarrow v'\), the alldifferent algorithm has to search in \(R\) for \(v\) and \(v'\) (if \(v'\) does not exist add \(v'\) to \(R\)), reduces and increases the count respectively and recompute the new penalty. Each update involve two steps (i.e. search for \(v\) and \(v'\) and recompute the sum of scores) and each step is \(O(n)\) in worst case. Although it is simple, the algorithm is obviously very computationally intensive for a list that involve frequent change.

A simple way to improve alldifferent is to store \(R\) in an array containing all the values in between the upper and lower bound of \(L\). This would allow locating \(v\) and \(v'\) to be achieved in constant time irrespective of the size of the list. For example, if the list \(L\) can take value from 0 \(-\) 9, then \(R\) would have the value from 0 \(-\) 9 and can be viewed as follow:

\[
R = \begin{array}{cccccccccc}
\text{Item} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\text{Count} & 0 & 1 & 1 & 2 & 0 & 2 & 1 & 1 & 0 & 0 \\
\end{array}
\]

(0 indicates the item do not exist in the list)

When the value 6 changes to 5, the alldifferent algorithm can locate 6 and 5 by its index in the array and update the value in constant time. For the recomputation of the penalty, we can use the incremental model for summation as in (2.5). Nevertheless, this approach requires \(\Omega(n)\) space where \(n\) is the offset of the upper and lower bound of \(L\). This is extremely inefficient in terms of memory management especially if the problem has large gap between upper and lower bound. In addition, it cannot be applied to a list where the upper and lower bound is unknown or too large to be practically implemented.

In order to overcome this limitation, we use a hash table of size \(2n\) where \(n\) is the size of the list (i.e. a hash table double the size of the list). A hash table only stores the values occur in the list hence allowing the implementation of alldifferent invariant to be memory efficient. Furthermore, the hash table has almost constant access time and this ensures the efficiency of our implementation. There is no specific theoretical reason behind the choice of \(2n\) but rather it is chosen heuristically to be sufficient to avoid excessive collision when storing all the values in the list. The hash
2.2. Invariants: One-Way-Solver

Table uses the chaining strategy in handling collisions as this strategy is suitable for problems that have frequent value change. Linear hashing is not suitable as when a value is changed, considerable reshuffling is required to retain the structure of the hash table.

**Incremental Update of Hash Table**

When $L_i$ changes from $v \rightarrow v'$, two update processes are required.

1. Reduce the score for $r_i \rightarrow r'_i$ where $r_i$ is the score for $L_i = v$. The new penalty is $R'$. Using the incremental update model defined in (2.5), $R'$ can be computed as:

   \[ R' = R - r_i + r'_i \]  

2. Increase the score for $r_j \rightarrow r'_j$ where $r_j$ is the score for $L_j = v'$. The new penalty is $R''$ can be computed as:

   \[ R'' = R' - r_j + r'_j \]  

By replacing $R'$ from (2.20) into (2.21) we get:

\[ R'' = R - r_i + r'_i - r_j + r'_j \]  

(2.22)

**Example:** Let $L$ has the following scoring hash table:

<table>
<thead>
<tr>
<th>Item</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Score</td>
<td>6</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Let the value 3 changes to 5. Therefore $c_2$ changes from $2 \rightarrow 1$ and $c_1$ changes from $3 \rightarrow 4$. The new state of the scoring hash table is:

<table>
<thead>
<tr>
<th>Item</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Score</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The new penalty can be computed incrementally as:

\[ R'' = R - r_2 + r'_2 - r_1 + r'_1 \]
\[ = 8 - 2 + 0 - 6 + 12 \]
\[ = 12 \]
2.2. Invariants: One-Way-Solver

**allsame Invariant**

If alldifferent means all element to take different value, allsame is the opposite of alldifferent. Similar to alldifferent, allsame does not inflict each element to take same value, but rather it returns the state of allsame in the list. The definition of allsame invariant is the opposite of the definition of alldifferent. Given

\[ v = \text{allsame}(d_1, d_2, ..., d_n) \]  

(2.23)

allsame invariant ensures that at any point of the computation, the value of \( v \) is the total number of item in the list that do not conform to allsame. \( v \) can be used to compute the penalty incurred by the list for not fulfilling the allsame constraint.

**Computing allsame Score**

Implementation of allsame is an extension of alldifferent. It shares the similar data structure as that of alldifferent, however the penalty is computed in different way. Let \( L \) be list of variables with finite domains \( D(x) \) and \( L \) has a total of \( T \) items. If an element, \( L_i \) in list \( L \) has \( c_i \) count (i.e. \( c_i \) number of items in \( L \) that have similar value), there are a total of \( c_i \times (T - c_i) \) pairs of conflicting allsame. Each pair incurs 1 penalty point hence the score, \( r_i \) is

\[ r_i = c_i \times (T - c_i) \]  

(2.24)

Therefore, the score for allsame, \( R \) is defined as:

\[ R = \text{allsame}(L) \Rightarrow \forall r_i \in L \]

\[ R = \sum_{i=1}^{n} r_i \]

(2.25)

\[ R = \sum_{i=1}^{n} c_i \times (T - c_i) \]

For instance, in the earlier example where list \( L \) contains the following items:

\[
\begin{array}{cccccccc}
5 & 3 & 2 & 5 & 3 & 1 & 6 & 7 \\
\end{array}
\]

\( L \) produces the counting table of:

\[
\begin{array}{cccccccc}
\text{Item} & 5 & 3 & 2 & 1 & 6 & 7 \\
\text{c_i} & 2 & 2 & 1 & 1 & 1 & 1 \\
\end{array}
\]

where \( c_i \) is the number of times an item repeats in the list.
2.2. Invariants: One-Way-Solver

There are eight items in \( L \) and therefore \( T = 8 \). Consider \( L_1 \) i.e. the item with value 5 where \( c_1 = 2 \). For each value 5 there are six other items in the list that are in conflict (i.e. \( T - c_1 \)). The score for \( L_1 \), \( R_1 = 2 \times 6 = 12 \).

Now consider \( L_2 \), i.e. item with value 3. \( c_2 = 2 \). Technically, there are also six other items in the list that are in conflict with 3. Nevertheless, four pairs of the conflict are actually with item 5, in which the score has already been counted in the scoring of for item 5, i.e. \( R_1 \). To avoid the recurring penalty of the same conflict, therefore \( R_2 = (2 \times 6) - 4 = 8 \).

For \( L_3 \), there are seven conflicting items, but only three are not recurring conflicts, therefore the score, \( R_3 = 3 \). The scoring table for all the items is as follow:

\[
\begin{array}{|c|c|c|c|c|c|c|}
\hline
\text{Item} & 5 & 3 & 2 & 1 & 6 & 7 \\
\hline
\text{Count} & 2 & 2 & 1 & 1 & 1 & 1 \\
\hline
\text{Score} & 12 & 8 & 3 & 2 & 1 & 0 \\
\hline
\end{array}
\]

We can generalize the formula in counting the score, \( r_i \) for each item. The non-recurring conflict item for \( L_i \) is \( \sum_{j=i+1}^{n} c_j \), i.e. the sum of \( c_j \) where \( j > i \). We can rewrite equation 2.24 as:

\[
r_i = c_i \times \sum_{j=i+1}^{n} c_j
\]

and the total penalty for \textit{allsame} \( R \) is the sum of penalty for all items:

\[
R = \sum_{i=1}^{n} r_i
\]

\subsection*{Incremental Update of \textit{allsame}}

The method for incremental update of \textit{allsame} is similar to the incremental update of \textit{alldifferent} as in (2.22). Here we reinstate the formula again.

\[
R'' = R - r_i + r'_i - r_j + r'_j
\]

Note that \( r_i, r'_i, r_j \) and \( r'_j \) are computed using (2.26)

\subsection*{2.2.4 Implementation}

The One-Way-Solver is implemented as collections of C++ libraries. The One-Way-Solver combines the aspects of declarative and imperative programming to ease the process of implementing local search. The imperative part of the One-Way-Solver provides easy integration into existing programming languages (i.e. C++) whereas
the declarative nature encapsulates the complex and error-prone implementation of incremental data structure found in local search. This helps to speed up the implementation of local search by providing a declarative way to specify invariants, i.e. the functions and variables that need to be maintained in order to define the penalty and neighborhood functions of local search.

To understand the implementation of One-Way-Solver, it is best to consider the basic implementation model. Algorithm 2.6 depicts a simple implementation of arithmetic model. In line 1, we include the invariant class for integer variable. Using the incremental integer variable class, we declare four variables of a, b, c, d (line 4–7) and initialize with the respective value as shown in the code. Note that we do not give any value to a and by default, the value will be null. In line 8, we construct the arithmetic operation, i.e. initialize $a = b + c \times d$. a will be the invariant product of $b + c \times d$. This form of initialization is made possible by using operator overloading in C++. Line 9 outputs the value of a which is 7. In line 10, the value of variable b is changed to 5 and the One-Way-Solver recompute the value of a incrementally. When the value of a is output again in line 11, the result would be 11. The implementation in Algorithm 2.6 is the simplest form of arithmetic invariant invariant. Given a significantly large equation, the One-Way-Solver is a handy tool for implementation of incremental computation.

**Algorithm 2.6 Arithmetic Invariant using One-Way-Solver**

```cpp
1. #include "in_int.h"
2. main()
3. {
4.   in_int a = new in_int();
5.   in_int b = new in_int(1);
6.   in_int c = new in_int(2);
7.   in_int d = new in_int(3);
8.   a = b + c * d;
9.   a.showValue();
10.  b.setValue(5);
11.  a.showValue();
12. return 0;
13. }
```

In the next example (Algorithm 2.7) we show the implementation of the list invariant. the list invariant is implemented by include the invariant list class in in_list.h (line 2). An invariant list is created as any object in C++ by passing the number of elements to be created in the list as argument when creating a new list (line 5). In the example, we created a list with 10 elements. Line 6–7 initialize the
value of the list. They set the value of element $i$ with the value of $i$. In other words, element 0 will be initialized with value 0, element 1 with 1 and so on. The setValue method takes two arguments. The first argument is the value to be set and the second argument is the element where the value is to be set. Line 8–12 activate the invariant features of the list. Each invariant feature must be activated explicitly. This allows programmer full control of which invariant features to be implemented in their program. The more invariant features that are activated, the more computing resources that are required by the data structure. Once the invariant feature is activated, we can call the methods that will return the value of the invariant. For instance, in line 13, the call to max() returns the element with the maximum value in the list. If the value of one of the element is changed (such as in line 18), the One-Way-Solver will automatically reconstructs the new invariant values using the incremental techniques discussed earlier.

Algorithm 2.7 List Invariant using One-Way-Solver

```cpp
1. #include <iostream.h>
2. #include "in_list.h"
3. main()
4. {
5.    in_list aList = new in_list(10);
6.    for(int i = 0; i < 10; i++)
7.       aList.setValue(i, i);
8.    aList.makeSum();
9.    aList.makeMin();
10.   aList.makeMax();
11.   aList.makeAllDifferent();
12.   aList.makeAllSame();
13.   cout << aList.sum() << endl;
14.   cout << aList.min() << endl;
15.   cout << aList.max() << endl;
16.   cout << aList.allDiffScore() << endl;
17.   cout << aList.allSameScore() << endl;
18.   aList.setValue(6, 3);
19.   return 0;
20. }
```

2.3 Chapter Summary

Over the past decades, there has been considerable interest in local search. This is due to the fact of its success in solving many difficult constraint optimization
problems. From a theoretical standpoint, local search algorithms sacrifice completeness for performance. They may fail to find optimal or even high quality solutions. However, on many problems they can find a solution in a reasonable time when systematic search may fail or be less effective. There has been substantial progress in the field of local search. It has grown from a simple heuristic idea into a complex algorithmic approach. As the algorithms become more and more sophisticated, the ideal case has been pushed aside in the quest for greater performance. Focus has been given to talk about neighborhoods, local moves, meta-heuristics and problem-specific knowledge to guide the heuristic process. Little opportunity is given to address the software engineering issues, such as reuse, separation and modularity.

In this chapter, we have presented a brief overview of local search techniques in solving constraint optimization problems. Several techniques have been presented such as the random walk, simulated annealing, tabu search, iterated local search and genetic algorithm. We adopted the idea of [103] and design a series of Invariants (or One-Way-Solver) to assist the implementation of local search. One-Way-Solver combines the aspects of declarative and imperative programming to ease the process of implementing local search. What is more important is that we want to clear the perception that local search is a collection of heterogeneous techniques lacking a unifying theme. Our One-Way-Solver shows that there is a possibility of designing local search algorithms in high-level language structure that offers both modeling and search abstractions. This is the point of benefit for local search from software engineering standpoint. In the next two chapters, we build local search algorithms using our One-Way-Solver to solve two classic combinatorial optimization problems, the non-CNF Satisfiability problem (SAT) and university course scheduling problem.
Chapter 3

Stochastic Non-CNF SAT Solver

In this chapter we apply the concepts of invariants and local search techniques presented in Chapter 2 onto the Boolean satisfiability problem (SAT). Besides demonstrating the application of invariants and local search we introduce an incomplete non-CNF SAT solver based on a novel technique for expressing the scoring function for evaluating the valuation for an objective function. In the solver, we use positive and negative scores based on the current valuation of the penalty function. By doing so, we are not only able to define the “falsity” (or penalty) of a non solution, but also the “truthfulness” of an assignment, that is, how true it makes the result. This is required since we need to negate “truthfulness” to obtain “falsity”. This complex scoring is necessary since we do not restrict ourselves to negation normal forms. It provides more accurate heuristic information to assist the local search process. This strategy opposes the usual stochastic methods such as [117, 118, 94, 131, 121, 116] which represent simply truth or degree of falsity (penalty). In addition, we include the implementation of extended Boolean operations such as NAND, NOR, XOR, XNOR, IMPLIES, IFF, ATLEAST, ATMOST and COUNT. To our knowledge, this is the first non-CNF solver that includes the implementation of all these operations. Our preliminary experiments show that on certain benchmarks the non-CNF local search solver can outperform highly optimized CNF local search solvers as well as existing CNF and non-CNF complete solvers.

3.1 Satisfiability Problem

The problem of determining whether a propositional Boolean formula can be evaluated to true is called the Boolean Satisfiability Problem or SAT. Indeed it is one of the first problems identified as NP(Nondeterministic Polynomial) complete [29] in computer science. Being in the class NP-complete means there is no known de-
3.1. Satisfiability Problem

A deterministic polynomial time algorithm that can find the solution. Given a Boolean
formula, the SAT problem asks for an assignment of variables such that the formula
evaluates to true, or a proof that no such assignment exists. Finding the solution to
a SAT formula is often complicated because the space of possible solutions for most
of the problem is far too large for brute-force or exhaustive search to be feasibly
applied.

Most modern SAT solvers operate on SAT formulas in Conjunctive Normal Form
(CNF). CNF format contains a single conjunction of one or more disjunctions of one
or more literals. A literal is a variable or its negation. A conjunction of literals is
also known as clause hence the term clausal is use interchangeably with CNF. An
example of a propositional formula in CNF is:

\[ \phi = (x \lor y \lor z) \land (\overline{x} \lor \overline{y} \lor z) \land (\overline{y} \lor z) \land (x \lor z) \land x \land \overline{z} \]

Recent development has seen remarkable progress in SAT solvers. They have
gone beyond research labs into many real world applications, such as constraint-
based planning [13] and [81], problems in finite algebra [46], verification of hardware
and software, scheduling [31], circuit synthesis [77], circuit diagnosis [85] and many
other domains including natural language processing and machine learning. The
value of research on SAT ultimately depends on our ability to find suitable SAT
encodings of real-world problem.

The key strength of SAT lies in its generality. Once a problem has been encoded
as SAT, any standard SAT solver can be used to find the solution to the problem.

3.1.1 SAT Solvers

Current SAT solvers can be categorized into complete or incomplete solvers de-
pending on the type of algorithms they employ. A complete solver uses systematic
search algorithms so the search space is explored systematically and exhaustively.
As a result given enough run time (and memory), a complete SAT algorithm can
always find a solution for a SAT instance or prove that no solution exists. While
there are variety of complete algorithms, the most successful ones are variants of the
Davis-Putnam-Longemann-Loveland (DPLL) algorithm [33]. The basic algorithm
has existed for more than forty years and a great deal of effort has gone into pro-
ducing robust and efficient implementations. Solvers like POSIT, TABLEAU [30],
GRASP [120], SATZ [88], REL-SAT [12] are well-known variants of DPLL-based
solvers. The advantage of complete algorithms is that they can detect infeasibility
when a SAT problem is infeasible. Unfortunately they are computationally expen-
sive for large instances. The reason is because the search algorithms keep track of the search paths and may become highly inefficient when the size of the problem is too large. Recent DPLL search has included clever techniques such as branching heuristics, non-chronological backtracking and conflict-driven learning. These techniques have indeed improved DPLL significantly and led to a new generation of solvers such as SATO [144], ZCHAFF [105] and most recently MINISAT [39].

Incomplete solvers on the other hand employ non-systematic search techniques such as stochastic local search (SLS). They cannot prove a SAT instance to be unsatisfiable. Nevertheless, for some hard satisfiable instances, stochastic methods may find a solution very quickly [see 23, 118]. In addition, they can handle problems an order magnitude larger than those that can be handled by complete solvers because they only maintain a single current state. Among the most successful variants of incomplete solvers are GSAT [117], WALKSAT [118], Novelty+ [94] and DLM [131].

3.1.2 Stochastic Local Search and SAT

Stochastic local search (SLS) algorithms have been successfully applied to many optimization problems. Nevertheless, only recently they have been applied to solve SAT problem and shown surprisingly good results. In 1993, GSAT [117] outperformed the best known systematic search algorithms on certain class of large satisfiability problems. It can find satisfying assignments of computationally hard randomly-generated 3CNF formulas with over 2000 variables, whereas the current fastest systematic search algorithms cannot handle instances from the same distribution with more than 400 variables [23]. The success of GSAT gave birth to several variants based on stochastic local search techniques [118, 94, 131, 121, 116]. SAT solvers designed using SLS are usually incomplete, although there are some recent attempts to make them complete [40, 119, 93, 51].

The principles of SLS SAT algorithms can be interpreted as performing a biased random walk in a search space given by the set of all complete truth assignments. The aim is to minimize the value of an objective function (which is often computed as the number of unsatisfied clauses). The main problem for SLS methods is that they may be trapped by local minima in the search space. A local minimum is a state whose local neighborhood does not include any state that is strictly better.

As we have discussed in Chapter 2, SLS will employ some mechanism to escape from local minima. Among the common techniques used in SLS SAT solvers are multi-restart, probabilistic uphill move, simulated annealing [118, 94] and dynamic objective functions using Discrete Lagrange Multiplier [131]. Algorithm 3.1 shows the outline of local search procedure for SAT.
3.2. Non-CNF SAT

Even though researchers have been studying SAT solving algorithms for a long time, efforts have mainly concentrated on solving SAT problem in Conjunctive Normal Form (CNF). Unfortunately, solving Boolean formula in CNF can be disadvantageous for problems which are more naturally encoded as arbitrary propositional formula whereby the formula has to be converted to CNF format. The conversion to CNF may increase the size of the formula exponentially, or significantly reduce the strength of the formulation. The translation may introduce many new variables which increases the size of the raw valuation space through which the solver must search. For example, converting formula to CNF using the Tseitin formulation [126] is linear in the size for Boolean operators that have linear clausal encodings. But operators that do not have linear clausal encodings such as $k$-ary XOR, $k$-ary bi-conditionals, $k$-ary counting operators like ATMOST, ATLEAST and COUNT may result in exponential blow-up.

There is a recent interest in the design of non-CNF SAT solvers. Non-CNF SAT solvers takes propositional formula in free form and support more complex Boolean operations. Thiffault et al. [124] argued that conversion to CNF is unnecessary. Indeed it results in the loss of structural information and increase in the search space. They implemented NOCLAUSE, a complete non-CNF solver that is capable of achieving an efficiency very similar to that of modern highly optimized CNF solvers using the techniques very similar to these solvers. The additional structural information present in non-CNF propositional formula allow NOCLAUSE to achieve significant gains in solving power, to the point where on various benchmarks their
NOCLAUSE has outperformed the CNF solver it was based on. After more than forty years of engineering efforts in CNF SAT solvers, it may appear that we have reached a plateau in terms of what can be achieved in practice. We cannot bet on any breakthrough without fundamentally changing the way we approach the problem. The success of NOCLAUSE gives a promising indication of the potential direction in the future of SAT. The key challenge is to invent new techniques for non-CNF solvers that will consistently outperform all the CNF solvers on a wide range of benchmark problems.

Non-CNF SAT Notations

Non-CNF SAT solvers take propositional formula in free form. It supports more complex Boolean operations. DIMACS extended style (for non-CNFS\cite{8}) defines the following operations apply to non-CNF propositional formula: \textit{TRUE}, \textit{FALSE}, \textit{NOT} (\texttt{¬}), \textit{AND} (\texttt{∧}), \textit{NAND} (\texttt{¬∧}), \textit{OR} (\texttt{∨}), \textit{NOR} (\texttt{¬∨}), \textit{XOR} (\texttt{⊕}), \textit{XNOR} (\texttt{¬⊕}), \textit{IMPLIES} (\Rightarrow), \textit{IFF} (\Leftrightarrow), \textit{IFTHENELSE} (<), \textit{ATLEAST} (\geq), \textit{ATMOST} (\leq) and \textit{COUNT} (≡). For the benefit of the readers, we shall define the operations used in the encoding of the benchmarks.

\textbf{NOT} (\texttt{¬}) - Given a variable \(v\) defined with \textit{NOT}, \(v\) is the negation of its input.

\textbf{AND} (\texttt{∧}) - Given a variable \(v\) defined with \textit{AND} operation, \(v\) is \textit{true}, if and only if all the inputs of \(v\) are \textit{true}, otherwise \(v\) is \textit{false}.

\textbf{OR} (\texttt{∨}) - Given a variable \(v\) defined with \textit{OR} operation, \(v\) is \textit{false}, if and only if all the inputs of \(v\) are \textit{false}, otherwise \(v\) is \textit{true}.

\textbf{XOR} (\texttt{⊕}) - Given a variable \(v\) defined with \textit{XOR} operation, \(v\) is \textit{false} if the parity of the total number of \textit{true} value is even, otherwise \(v\) is \textit{true}.

\textbf{ATMOST} (\texttt{≤k}) - Given a variable \(v\) with parameter \(k\) and \(n\) input, \(v\) is \textit{true} if at most \(k\) inputs of \(v\) are \textit{true}. Otherwise \(v\) is \textit{false}.

\textbf{ATLEAST} (\texttt{≥k}) - Given a variable \(v\) with parameter \(k\) and \(n\) input, \(v\) is \textit{true} if at least \(k\) inputs of \(v\) are \textit{true}. Otherwise \(v\) is \textit{false}.

\textbf{COUNT} (\texttt{≡k}) - Given a variable \(v\) with parameter \(k\) and \(n\) input, \(v\) is \textit{true} if exactly \(k\) inputs of \(v\) are \textit{true}. Otherwise \(v\) is \textit{false}.

Non-CNF formulas are often expressed as directed acyclic graphs (DAG). Formulas represented as DAGS are called Boolean circuits. Each internal node represents a Boolean operator and its children are subtrees representing each operand. The variables representing the internal nodes are known as \textit{auxiliary variables} whereby
their value depends on the value of their children. The leaves are known as primary variables. In this paper, the following notation will be used to represent a SAT node where $I_0$ is the node and $I_1$ to $I_n$ are the children of $I_0$.

$$I_0 = GATE(I_1, I_2, ..., I_n) \quad (3.1)$$

### 3.2.1 The State of non-CNF SAT Solvers

Unlike CNF SAT solvers, which have been under intensive investigation for the last forty years, non-CNF solvers have not seen much research effort until the last several years. As a result, current non-CNF solvers are far less sophisticated compared to CNF SAT solvers. Just as the CNF Solvers, non-CNF solvers are also classified as complete or incomplete. At present, the trend focuses on generalizing well-known clausal algorithms to take non-CNF formula. Within the class of complete non-CNF solvers, in 1993, Armando and Giunchiglia [7] introduced PTAUT, which was the generalization of Davis-Putnam (DP) algorithm to work on non-CNF formula. The primary drawback of this implementation was the performance which was far below current implementation of CNF solver based on DP algorithm. This is due to the fact that the algorithms failed to exploit fast propagation, smart algorithm and efficient data structures that are used in manipulating CNF formulas, e.g. [30, 145]. The same work was improved by Giunchiglia and Sebastiani [52] to allow the algorithm to exploit all the present and future sophisticated technology of DP implementations at the same time accept input in non-CNF format. Unfortunately, this improvement requires the formula in CNF in order to ensure the correctness and completeness of DP procedure. What is unique in this variant is that the DP procedure will not backtrack on new variables introduced in CNF conversion hence maintaining the search space of the original non-CNF formulas despite the fact the formulas have been converted to CNF with an addition of $k$ variables. The most notable implementation of a complete non-clausal solver is probably the work of Thiffault et al. [124] who generalized Davis-Putnam-Logemann-Loveland (DPLL) algorithm [33] onto non-CNF formulas. They argued that conversion to CNF is unnecessary and results in the drawback of losing structural information and increase in the search space. They implemented NOCLAUSE, a complete non-CNF DPLL like solver that is capable of achieving the efficiency very similar to that of modern highly optimized CNF solvers using the techniques very similar to these solvers. Unfortunately, NOCLAUSE only supports AND, OR and NOT operations, i.e. similar to that of CNF solvers except NOCLAUSE accepts formula in free form.

There are also attempts to implement incomplete non-CNF solvers based on
stochastic local search techniques. Several authors have attempted to generalize popular local search techniques found in CNF solvers onto non-CNF formula. Sebastiani [116] suggested how to modify GSAT to be applied to non-CNF formula however no experiment result has been published based on his idea. Moreover, the implementation is only limited to the three basic Boolean operation, i.e. NOT, AND, OR. Similarly, Stachniak [121] introduced polWSAT an evolution of WalkSAT [118] to handle non-Clausal formula. polWSAT has shown to perform better than classic WalkSAT when augmented with structure preserving translation of non-CNF to CNF. Unfortunately the input of polWSAT is restricted to conjunction and disjunction operations. Kautz et al. [79] also introduced DAGSat, a general structural representation of propositional formula that improves handling of variable with dependencies. It can be applied onto any arbitrary propositional structure (such as CNF and non-CNF format). It has been shown that CNF WalkSAT improved when implemented using DAGSat. Theoretically, DAGSat can be extended to represent formula in non-CNF format, but does not guarantee that the resultant solver will perform better than its CNF counterpart.

3.3 Stochastic Local Search Non-CNF SAT

Let $\phi$ be a propositional formula, our aim is to decide if $\phi$ is satisfiable. The propositional formula is represented as a Boolean DAG where each internal node represents a Boolean operator and its children are sub-trees representing its operands. Our DAG structure similar to that of [79]. For example, Figure 3.1 represents a Boolean formula $\phi$ as a DAG. The leaves in the Boolean DAG represent the Boolean variables. The satisfiability of $\phi$ can be determined when all the value of the variables are known. For example, if $A = \text{false}$, $B = \text{true}$, $C = \text{false}$, $D = \text{true}$, $E = \text{false}$ and $F = \text{false}$, then $\phi$ is satisfiable.

The aim of our algorithm is to find values for the variables such that they will result in the satisfiability of the propositional formula. For each assignment, a score is computed for each node in the DAG to represent the state of satisfiability of the corresponding propositional (sub-)formula.

3.3.1 Score

The notion of score plays a key role in determining the “distance” from the current valuation to a satisfying one. We allow our scoring function to take positive and negative values in order to enable our algorithm to express the “truthfulness” as well as the “falsity” of the state of the truth assignment. This is required since we need
to negate “truthfulness” to obtain “falsity”. This complex scoring is necessary since we do not restrict ourselves to negation normal forms. It provides more accurate heuristic information to assist the local search process. To our knowledge, this is the first stochastic local search solver that uses this form of scoring technique.

All Boolean variables (leaf nodes) have a score of either 1 (true) or (−1) false. For each internal node we calculate the score as defined below. The calculations are such that:

- A negative score indicates that the node is false and a positive score indicates that the node is true.
- If the Boolean DAG were a tree then a score of +n at node φ₀ means that n children of φ₀ must change their score from positive to negative for the polarity of φ₀ to become false, while if the score of node φ₀ is −n then n children of φ₀ must change their score from negative to positive for the polarity of φ₀ to become true.

We will determine the score s₀ for a node φ₀, in terms of the scores s₁, . . . sₙ of its n children φ₁, . . . φₙ.

NOT(¬) φ₀ = ¬φ₁

NOT is the negation of the truth value of a variable, therefore the score for a negated node, s₀, is the negation of the score of its child, s₁: s₀ = −(s₁).

AND(∧) φ₀ = φ₁ ∧ · · · ∧ φₙ

An AND node (with n children) can only be true if all its children are true, therefore, if the node is currently false, the score of the node is the sum of negative

Figure 3.1: The propositional formula \( \phi = ((\neg A \lor \neg B \lor \neg C) \land \neg D \land (\neg E \lor \neg F)) \land 
\neg C \land ((\neg D \lor A \lor \neg E) \oplus (C \land F)) \) with scores for the valuation \{A, \neg B, C, \neg D, \neg E, F\} bracketed.
3.3. Stochastic Local Search Non-CNF SAT

![Diagram of AND and ORscore](image.png)

Figure 3.2: AND score (a) where the node is false and (b) where it is true

Figure 3.3: OR score (a) where the node is false and (b) where it is true

children (Fig. 3.2(a)) and if the node is true, the score of the node is the score of the child with minimum value (Fig. 3.2(b)).

\[
s_0 = \begin{cases} 
\sum \{s_i | 1 \leq i \leq n, s_i < 0\} & \forall 1 \leq i \leq n, s_i < 0 \quad (false) \\
\min\{s_i | 1 \leq i \leq n\} & \text{otherwise} \quad (true) 
\end{cases}
\]

For example, in order to change the node in Fig. 3.2(a) to true, we have to change the nodes with \([-2\], \([-3\] and \([-1\] to have a positive score, hence the score of the parent is \([-6\]). On the other hand, the cost of the node in Fig. 3.2(b) is \([+1\]) because the minimum change to turn the node to false, is by turning the truth value of the child with cost \([+1\]) to false.

**OR(\lor) \ \phi_0 = \phi_1 \lor \cdots \lor \phi_n**

An OR node becomes true when any one of its children is true, and false if all of its children are false, therefore, if the node is false, the score of the node is the score of the child with maximum value and if the node is true, the score of the node is the sum of positive children.

\[
s_0 = \begin{cases} 
\max\{s_i | 1 \leq i \leq n\} & \forall 1 \leq i \leq n, s_i < 0 \quad (false) \\
\sum \{s_i | 1 \leq i \leq n, s_i > 0\} & \text{otherwise} \quad (true) 
\end{cases}
\]

For example in order to invert the node in Fig. 3.3(a) to true, the minimum cost is \([-2\]) i.e. by making the child with \([-2\] score to true. On the other hand, to invert the node in Fig. 3.3(b) to false, we have to turn the children with score \([+1\] and \([+2\] to false therefore giving the score of the parent as \([+3\].

**XOR(\oplus) \ \phi_0 = \phi_1 \oplus \cdots \oplus \phi_n**

A naive clausal representation of an XOR constraint requires an exponential number of clauses. This gives the advantage of non-clausal over clausal solver on
3.3. Stochastic Local Search Non-CNF SAT

Figure 3.4: XOR score (a) where the node is false and (b) where it is true

problem with large XOR formula, such as in linear cryptanalysis [92] problem and the DIMACS 32 bits parity problem [74].

An XOR \(\oplus\) is true if the parity of the total number of true children is odd. Simply flipping the truth value of any one of the children of \(\phi_0\) will change the parity from odd to even and vice versa. Hence the XOR node that is false has a score equal to the negative of the smallest absolute value of any child score, while if its true, the score is the smallest absolute value of the child score.

\[
s_0 = \begin{cases} 
-\min\{|s_i| \mid 1 \leq i \leq n\} & \{s_i | 1 \leq i \leq n, s_i > 0\} \mod 2 = 0 \quad (false) \\
+\min\{|s_i| \mid 1 \leq i \leq n\} & \text{otherwise} \quad (true)
\end{cases}
\]

For example, the XOR node in Fig. 3.4(a) is false because two of the children are true (even number). In order to invert the node to true, we have to flip the truth value of one of the children. The least change required is to change the [+2] child to false, hence the score is [−2]. Fig. 3.4(b) shows the score of an XOR node that is true. The score is [+2] as the least change require to invert the node to false is to flip the child with score [−2] to [+1].

\[\text{IFF}(\iff) \quad \phi_0 = \phi_1 \iff \cdots \iff \phi_n\]

Equivalent (IFF) is the opposite of XOR. An IFF node \(\phi_0\) is true if the parity of the total number of true children is even. The resulting score function is then:

\[
s_0 = \begin{cases} 
-\min\{|s_i| \mid 1 \leq i \leq n\} & \{s_i | 1 \leq i \leq n, s_i > 0\} \mod 2 = 1 \quad (false) \\
+\min\{|s_i| \mid 1 \leq i \leq n\} & \text{otherwise} \quad (true)
\end{cases}
\]

\[\text{IMPLIES}(\Rightarrow) \quad \phi_0 = \phi_1 \Rightarrow \phi_2\]

Implication is a simple binary operator. An implication formula is false if and only if the left operand is true and right operand is false. If the implication node is false, then flipping either operand will turn the implication node to true. If the implication node is true, the cost of turning the node to false would be the sum of the cost of turning the left operand to true \( \left( l_i \right) \) and the cost of turning the right
3.3. Stochastic Local Search Non-CNF SAT

Figure 3.5: ATMOST 2 score (a) where the node is false and (b) where it is true.

\[
s_0 = \begin{cases} 
\max\{-s_1, +s_2\} & s_1 > 0 \land s_2 < 0 \quad \text{(false)} \\
\max\{s_2, 0\} - \min\{s_1, 0\} & \text{otherwise} \quad \text{(true)} 
\end{cases}
\]

**ATMOST** \((\leq k)\) \(\phi_0 = (\phi_1 + \cdots + \phi_n \leq k)\)

Given a node with \(n\) inputs, if the node is defined with ATMOST operator \((\leq)\) with parameter \(k\), it is true iff at most \(k\) inputs of the node are true. If the node is false, and currently \(k' > k\) children are true, we need to make false \(k' - k\) more variables, so the score is negative of the minimum sum required to do this. Similarly, if the node is true, and currently \(k' < k\) children are true, we need to turn \(k - k'\) children false.

\[
s_0 = \begin{cases} 
-\sum_{k' - k} T & T = \{s_i \mid 1 \leq i \leq n, s_i > 0\}, \\
\k' | T |, k' > k & \text{(false)} \\
-\sum_{k' - k + 1} \max\{s_i \mid 1 \leq i \leq n\} - T & T = \{s_i \mid 1 \leq i \leq n, s_i > 0\}, \\
\k' | T |, k' \leq k & \text{(true)} 
\end{cases}
\]

where \(\min_{l} S\) returns the minimal \(l\) elements of \(S\), that is a subset \(M\) of \(S\) of cardinality \(l\) such that \(\forall x \in S - M, \forall y \in M, x \geq y\). Similarly \(\max_{l} S\) returns the maximal \(l\) elements of \(S\). For example, in Fig. 3.5(a), to change the node to true, we need to change two children to false. The cheapest way of doing this is with the \([+1]\) and \([+2]\) children. \(T = \{+6, +4, +1, +2\}\), \(k' = 4\) and \(\min_{2} T = \{+1, +2\}\). Hence the resulting score is \([-3]\). In Fig. 3.5(b), to change the node to false, we would need to make at least two more children true. \(T = \{+6\}\), \(k' = 1\) and \(\max_{2}\{-4, -3, -2\} = \{-2, -3\}\). Hence the score is \([+5]\).

**ATLEAST** \((\geq k)\) \(\phi_0 = (\phi_1 + \cdots + \phi_n \geq k)\)

Given a node with \(n\) inputs, if the node is defined with ATLEAST operator \((\geq)\) with parameter \(k\), it means that at least \(k\) inputs of the node are true. ATLEAST is the reverse implementation of ATMOST. If a node is true, to invert the node to false we have to flip \(x\) number of true children such that the total number of true
3.3. Stochastic Local Search Non-CNF SAT

If a node is false, to invert the node to true we have to flip \( x \) number of false children such that the total number of true children is greater or equal to \( k \). The resulting scoring function is thus a mirrored form.

\[
s_0 = \begin{cases} 
\sum \max_{k' - k'} \{s_i \mid 1 \leq i \leq n\} - T \quad & T = \{s_i \mid 1 \leq i \leq n, s_i > 0\}, \\
\sum \min_{k' - k} T \quad & k' | T |, k' < k \quad (false) \\
\min \{|s_i| \mid 1 \leq i \leq n\} \quad & \text{otherwise} \\
\end{cases}
\]

For example, in Fig. 3.6(a), to invert the node to true, we would flip the value of children with scores \([-2]\) and \([-3]\) because they are the most vulnerable children. Therefore the score of the node is \([-5]\). In Fig. 3.6(b), to invert the node to false, we choose to flip the value of children with scores \([+1]\), \([+2]\), and \([+4]\) because they are the most vulnerable children. The score of the node is therefore \([+7]\).

\[
\begin{align*}
\text{(a)} & \quad \geq 2 & \quad [−5] \\
\end{align*}
\]

\[
\begin{align*}
\text{(b)} & \quad \geq 2 & \quad [+7] \\
\end{align*}
\]

Figure 3.6: ATLEAST 2 score (a) where the node is false and (b) where it is true

COUNT(\(\equiv k\)) \(\phi_0 = (\phi_1 + \cdots + \phi_n = k)\)

Given a node with \(n\) inputs, if the node is defined with COUNT operator \((\equiv)\) with parameter \(k\), it means that exactly \(k\) inputs of the node are true. The COUNT operator is simply a combination of the ATMOST and ATLEAST operators, but it ends up with a slightly different form of scoring function, after simplifying. If the node is false to make it true we need to flip some truth values to bring the count either up or down to \(k\). To make it false from true we only have to flip one truth value.

\[
s_0 = \begin{cases} 
\sum \max_{k' - k'} \{s_i \mid 1 \leq i \leq n\} - T \quad & T = \{s_i \mid 1 \leq i \leq n, s_i > 0\}, \\
- \sum \min_{k' - k} T \quad & k' | T |, k' < k \quad (false) \\
\min \{|s_i| \mid 1 \leq i \leq n\} \quad & \text{otherwise} \quad (true)
\end{cases}
\]

In Fig. 3.7(a), turning the node to false is simply flipping the truth value of any of its children. The child \([-1]\) is the most vulnerable node, therefore it is the score
of the node. In Fig. 3.7(b), since the total number of true children is greater than $k$, to invert the node to true, we have to reduce the number of children with true by two. The children [+1] and [+2] are chosen because they are the most vulnerable children. Therefore the score is $[-3]$. In Fig. 3.7(c), since the total number of true children is less than $k$, to invert the node to true, we have to increase the number of children with true by two. The children $[-2]$ and $[-3]$ are chosen because they are the most vulnerable children. Therefore the score is $[-5]$.

**Other Operators** There are also other operators defined in DIMACS non-clausal style [8] such as NAND, NOR, and XNOR. These operators are the negation of operators we have already defined. We can generate their scoring functions by simply negating the corresponding versions: NAND = – AND, NOR = – OR, and XNOR = – IFF.

### 3.3.2 Searching for a Solution

All local search based solvers work in essentially the same way. A candidate valuation for the variables is determined, and then the search looks for a “neighboring” valuation which is better. In SAT solvers the usual definition of neighboring valuations, is those obtained by flipping the value of one Boolean variable. A valuation is considered better if it satisfies more clauses, or satisfies a greater sum of weighted clauses. In the non-clausal solver the situation is the same. We have a current valuation and its score for the overall formula. We look at neighboring valuations which improve the score, attempting to drive the score to be positive (and hence satisfying the root propositional formula).

We consider a search strategy similar to the approach used by WalkSAT [118] and its extension to non-clausal solvers [121]. WalkSAT works by first selecting an unsatisfied clause, and then selecting a variable in that clause for flipping. A
CNF formula is simply a root AND node above many OR nodes relating to literals (variables or NOT variables sub trees). How do we generalize the WalkSAT approach to an arbitrary formula? We consider it in this way. The WalkSAT approach begins at the root and considers which child nodes could improve the root score and randomly selects one. Depending on the type of operation, the selection is guided by the following policies:

- Select same sign: AND, OR, ATLEAST
- Select opposite sign: ATMOST
- Select at random: XOR, IFF
- Combination of the above: COUNT

Hence the generalization is clear. At node $n$ we select a child node based on the policies, for which flipping the truth value would move the nodes score towards the opposite sign it has now. We continue this process until we reach a variable node. This is the variable we then flip in the search process. We flip the variables truth value and with probability $p$ accept the change if it is downhill (improves the score of the root) or flat (unchanged score), and with probability $1 - p$ accept the move whether it is downhill, uphill or flat. For our experiments we found a value of $p = 0.9$ was best and use this value throughout our experiments.

Examining the DAG shown in Figure 3.8. At the root the score of $[-3]$ is the sum of the negative children of $[-1]$, since the operation for the root node is AND ($\land$) so we select the children with the same sign (i.e. negative). There are three children that qualified for the selection, so we randomly select one of them, say the leftmost AND node. Its score is as a result of one of its children being negative so again we random select the children with the same sign. In this case, there is only one child qualified, i.e. the rightmost OR node. Its score is negative because all of its children are negative. For $OR$ the selection policy is to choose the child with similar sign, so we randomly choose one say the right child. The right child is the negation of variable node $F$ so we pick variable node $F$ for flipping. Flipping the value of variable $F$ from positive ($+1$) to negative ($-1$) results in new valuation for the score of the tree. It has indeed improved the score of the root node. Figure 3.9 shows the scores of the tree after the value of $F$ changes from positive to negative. Indeed, by the next cycle, $\phi$ would be solved as the selection process will lead to variable node $C$. Flipping $C$ from positive to negative would solve $\phi$. 

3.3.3 Implementation

We implemented our stochastic non-clausal SAT solver in C++ using the technology of One-Way-Solver. Each node in the graph corresponds to an invariant function which computes the value of the score in terms of changes to the scores of its child nodes. We use incremental approaches to calculate the change in scores efficiently, and schedule each node to be evaluated after all its children have been evaluated. Of course we do not reevaluate nodes that have not had changes in their children. This corresponds to the static scheduling of [103].

The One-Way-Solver plays an important part in implementing invariants for computing score for each node. For instance, consider an AND node. As defined in Section 3.3.1, the score of an AND node when it is false is the sum of negative children and when it is true is the score of the child with minimum value. Here, there
are two invariants to be maintained, i.e. the sum and the min invariant respectively. These invariants can be implemented easily and efficiently using the `makeSum` and `makeMin` features in the `in_list` data structure provided by the One-Way-Solver. The same applies to other type of operations, such as an OR node requires the max and sum invariants, XOR requires min and max invariants, etc.

### 3.4 Non-CNF SAT Benchmarks

#### 3.4.1 Hard Random Non-CNF Formula

Randomly generated formulas provide a good test bed for evaluating the performance of satisfiability algorithms. We used the random formula generator of Navarro and Voronkov [107] which is based on a fixed shape model. The generator is capable of generating formulas with different levels of difficulty in non-CNF as well as CNF format hence making it the ideal choice to test our stochastic non-CNF solver. The difficulty and the satisfiability of the output formulas are controlled by \( r \) the ratio of formulas-to-variables. Small \( r \) produces formulas that are satisfiable and under-constrained, while large \( r \) results in formulas that are unsatisfiable and over-constrained. The hardest problems appear in the phase transition region where there are just enough constraints to make the problem potentially unsatisfiable, but not too many to make it easy for a solver to determine [107]. We generated 200-variable random formula of \( ⟨3,3,2⟩ \) shape (as defined in [107]) with \( r \) increasing by 0.2 within the range from 1.6 to 3.0.

#### 3.4.2 Social Golfers as Non-CNF SAT Problem

The social golfers problem is derived from a question posted to `sci.op-research` in May 1998:

> The coordinator of a local golf club has come to you with the following problem. In her club, there are 32 social golfers, each of whom play golf once a week, and always in groups of 4. She would like you to come up with a schedule of play for these golfers, to last as many weeks as possible, such that no golfer plays in the same group as any other golfer on more than one occasion.

Since 1998, this problem has become a famous combinatorial problem. It is problem number 10 in CSPLib (http://www.csplib.org). A solution is said to be optimal when maximum socialisation is achieved, i.e. when one golfer plays with as
many other golfers as possible. The CNF encoding of social golfer problem has been implemented by Gent and Lynce [50]. In this section, we present a non-CNF encoding of the social golfers problem using the \textit{COUNT} gate. The encoding achieves a very significant gain in terms of problem size compared to its CNF counterpart.

The Social golfers problem can be described explicitly with four constraints which must be satisfied:

1. The golf club has 32 members (players).
2. Each member plays golf once a week.
4. No golfer plays in the same group as any other golfer twice.

Although the original problem was described for 32 golfers playing in groups of 4, it can be easily generalized. An instance to the problem is characterized by a triple \(w - p - g\), where \(w\) is the number of weeks, \(p\) is the number of players per group and \(g\) is the number of groups. Therefore, the original problem is a \(w - 4 - 8\) problem.

To encode the social golfers problem as non-CNF SAT, we begin by defining a set of variables representing the players, the groups and weeks they play with each other. For a \(w - p - g\) social golfers problem, we require \(w \times (p \times g) \times g\) variables. Let \(n\) be the total number of golfers (i.e. \(p \times g\)), we can define on \(\text{week}_i, \ \text{player}_j\) plays in \(\text{group}_k\) as \(G(i, j, k)\) where \(0 \leq i < w\), \(0 \leq j < n\) and \(0 \leq k < g\). If in \(\text{week}_i\), \(\text{player}_j\) plays in \(\text{group}_j\), than \(G(i, j, k)\) is true, otherwise false.

Having defined the variables, we can begin to define the constraints associated to the variables. The constraints need to specify that:

- Each golfer plays exactly once per week i.e.:
  - Each golfer plays at least once per week.
  - Each golfer plays at most once per week.

- Each group in each week has exactly \(p\) players.

- No golfer plays with any other golfer twice.

\textbf{Each Golfer Plays Exactly Once Per Week}

A naive CNF way to define the constraint requires three sets of clauses. The first set is to define that each golfer plays at least once per week. The second set is to define that each golfer plays at most once in the same group and finally, the third set ensures that each golfer plays at most once per week. [see 50].
In our non-CNF encoding, we use the COUNT ($\equiv$) operator to encode the constraint. Since a COUNT gate means exactly $k$ input is true, if $k = 1$, then it literally means “one and only one.” Therefore we only require a single set of encoding to define the constraint that each golfer plays exactly once per week. It can be defined as:

$$\equiv 1(G(i, j, 1), G(i, j, 2), ..., G(i, j, g))$$ (3.2)

This requires only $(i \times n \times k) + 1$ auxiliary variables to encode the constraint.

Each Group in Each Week has Exactly $p$ Players

Similarly, the CNF approach requires two set of clauses. The first set defines each group has at least $p$ players and the second set makes each group has at most $p$ players.

Our non-CNF encoding, again we use the COUNT ($\equiv$) operator to encode the constraint. The constraint for each group in each week has exactly $p$ players can be defined as:

$$\equiv p(G(i, 1, k), G(i, 2, k), ..., G(i, n, k))$$ (3.3)

Note that by setting the parameter of the COUNT gate as $p$, it ensures that each group has exactly $p$ players. This constraint requires only $w \times g$ auxiliary variables.

No Golfers Plays with Any Other Golfer Twice

The primary variables we have defined so far allow us to know exactly which golfers are scheduled to play in each match. For instance, $G(0, 0, 0)$ shows that golfer 0 plays in group 0 in week 0. Player $x$ plays with player $y$ when they intersect in week and group. Therefore the constraint $G(w, x, g) \land G(w, y, g)$ defines that player $x$ is playing with player $y$. So, if the two players, $x$ and $y$ only play once then only one of the instances where player $x$ playing with player $y$ can be true. This can be defined with a COUNT gate with $k = 1$.

$$\equiv 1(\forall i \forall k \; G(i, x, k) \land G(i, y, k))$$ (3.4)

If there are $n$ players, then there are a total of $\sum_{i=0}^{n-1} i = \frac{n^2 - n}{2}$ pairs of players. Each pair has $w \times g$ intersections, hence the total number of nodes produced is

$$\left(\frac{n^2 - n}{2} \times w \times g\right) + \frac{n^2 - n}{2}$$
3.4. Non-CNF SAT Benchmarks

Encoding Results

Table 3.1 shows the result of our encoding as compare to that of Gent and Lynce [50]

<table>
<thead>
<tr>
<th>Problem</th>
<th>CNF</th>
<th></th>
<th></th>
<th>Non-CNF</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td># Vars</td>
<td># Cls</td>
<td># Vars</td>
<td># Nodes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7-4-2</td>
<td>1456</td>
<td>8556</td>
<td>112</td>
<td>606</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9-5-2</td>
<td>3375</td>
<td>21665</td>
<td>180</td>
<td>1147</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5-4-4</td>
<td>4000</td>
<td>35032</td>
<td>320</td>
<td>2944</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11-6-2</td>
<td>6732</td>
<td>46026</td>
<td>264</td>
<td>1940</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7-6-3</td>
<td>9450</td>
<td>79965</td>
<td>378</td>
<td>3895</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13-7-2</td>
<td>12103</td>
<td>86751</td>
<td>364</td>
<td>3033</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6-5-5</td>
<td>13500</td>
<td>147950</td>
<td>750</td>
<td>10234</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7-7-3</td>
<td>14406</td>
<td>127302</td>
<td>441</td>
<td>5233</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5-8-3</td>
<td>14880</td>
<td>135780</td>
<td>360</td>
<td>4915</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3-6-6</td>
<td>15876</td>
<td>207054</td>
<td>648</td>
<td>12748</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15-8-2</td>
<td>20160</td>
<td>149912</td>
<td>480</td>
<td>4474</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6-7-4</td>
<td>21756</td>
<td>227402</td>
<td>672</td>
<td>10318</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3-8-5</td>
<td>24480</td>
<td>303260</td>
<td>600</td>
<td>13219</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5-7-5</td>
<td>28175</td>
<td>339185</td>
<td>875</td>
<td>16549</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17-9-2</td>
<td>31671</td>
<td>242541</td>
<td>612</td>
<td>6311</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4-7-6</td>
<td>32340</td>
<td>40013</td>
<td>1008</td>
<td>22729</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3-9-5</td>
<td>34020</td>
<td>43260</td>
<td>675</td>
<td>16669</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10-9-3</td>
<td>41310</td>
<td>388341</td>
<td>810</td>
<td>11995</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6-9-4</td>
<td>43740</td>
<td>484614</td>
<td>864</td>
<td>16858</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-10-3</td>
<td>44400</td>
<td>426435</td>
<td>720</td>
<td>11863</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19-10-2</td>
<td>47500</td>
<td>372630</td>
<td>760</td>
<td>8592</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3-9-6</td>
<td>48843</td>
<td>701811</td>
<td>972</td>
<td>28345</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5-10-4</td>
<td>49000</td>
<td>554140</td>
<td>800</td>
<td>17404</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4-8-7</td>
<td>63616</td>
<td>995876</td>
<td>1568</td>
<td>46484</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5-10-5</td>
<td>76250</td>
<td>991925</td>
<td>1250</td>
<td>33379</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4-9-7</td>
<td>88452</td>
<td>1419075</td>
<td>1764</td>
<td>58685</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Comparison of number of variables and clauses between CNF and Non-CNF encoding

3.4.3 Minimum Disagreement Parity Problem

The Minimal Disagreement Parity (MDP) Problem [74] is a well-known class of hard satisfiability problem. It is listed as challenge number two in the ten challenges in propositional reasoning by Selman [80]. Although this challenge was solved in 1998, see [139, 87] the solutions seem superficial because they are achieved by preprocessing the input before a SAT solver is applied to solve the problem. Indeed no SAT solver
can solve the 32 bits problem directly. Moreover, the problem remains an interesting
benchmark for testing local search solvers particularly because they are known to
be very hard for such solvers. The reason is because local search algorithms tend to
become trapped at a near-solution where a small subset of clause is never satisfied
simultaneously.

Informally the MDP problem is the following: given a set of sample input vectors and a set of sample parities, find the bits of the input vectors on which the parities
were computed. In the absence of noise this problem has a known polynomial
solution. In the presence of a moderate amount of noise (i.e., corruption of around
1/4 of the parity bits), however, there is reason to believe that the problem is quite
hard [74].

Problem Statement

Here we formally redefined the MDP problem as in [74]. In general, the parity of an
integer is its attribute of being even or odd. In binary representation, parity is the
digit “1” count modulo 2. Therefore 1010 has two 1s and hence has parity 2 mod 2,
which is 0. It is the parity of binary number that we considered in MDP problem.
In defining the MDP problem, the following definitions of operation will be used.

Definition 3.4.1 If \(a, b \in \{0, 1\}\) then \(a + b\) is the parity of \(a\) and \(b\).

Example: \(1 + 0 + 0 + 1 + 1\) has parity 1 (3 mod 2).

Definition 3.4.2 If \(a, b \in \{0, 1\}^n\) then \(a \times b = \sum_{i=1}^{n} a_i b_i\)

Example: \(10011 \times 01101 = (1)(0) + (0)(1) + (0)(1) + (1)(0) + (1)(1)\)
\[= 0 + 0 + 0 + 0 + 1\]
\[= \text{has parity 1 (1 mod 2)}\]

The minimal disagreement parity problem (MPD) is defined formally below:

Instance:
- Sample input: \(x_i \in \{0, 1\}^n\) for \(i = 1, ..., m\)
- Sample output: \(y_i \in \{0, 1\}\) for \(i = 1, ..., m\)
- Error tolerance: integer \(k\)

Find: a vector \(a \in \{0, 1\}^n\) such that the parity function represented by \(a\) disagrees
with the input of sample on at most \(k\) points, i.e., for which

\[|\{i : a \times x \neq y_i\}| \leq k\]
Here the $x_i$ represent $m$ sample inputs over $n$ Boolean variables, the $y_i$ the (potentially incorrect) value of the unknown parity function on each of the $m$ inputs, and $k$ represents a bound on the acceptable number of errors.

### Generating a MDP Problem

The MDP problem is generated as follow:

1. First, a random target vector $s \in \{0, 1\}^n$ is generated uniformly at random.
2. Next, a set of sample input $x_i \in \{0, 1\}^n$ are generated uniformly at random for $i = 1, \ldots, m$.
3. Finally, a set of “noise bits” $r_i \in \{0, 1\}^m$ are generated by randomly choosing exactly $k$ of these bits to be 1 and the rest are set to 0. Each sample output $y_i$ is then computed as $y_i = s \times x_i + r_i$.

The resulting instance is then given by the sample inputs $x_i$, the sample output $y_i$ and the error tolerance $k$.

**Example:** Let $m = 8$, $n = 4$ and $k = 2$

\[
\begin{align*}
x_1 &= \begin{bmatrix} 0 & 1 & 1 & 0 \end{bmatrix} \\
x_2 &= \begin{bmatrix} 1 & 1 & 0 & 0 \end{bmatrix} \\
x_3 &= \begin{bmatrix} 0 & 1 & 1 & 0 \end{bmatrix} \\
x_4 &= \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix} \\
x_5 &= \begin{bmatrix} 1 & 0 & 1 & 0 \end{bmatrix} \\
x_6 &= \begin{bmatrix} 1 & 0 & 1 & 0 \end{bmatrix} \\
x_7 &= \begin{bmatrix} 1 & 1 & 0 & 0 \end{bmatrix} \\
x_8 &= \begin{bmatrix} 0 & 1 & 1 & 1 \end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
x \times s &= \begin{bmatrix} 0 \end{bmatrix} \\
r &= \begin{bmatrix} 0 \end{bmatrix} \\
y &= \begin{bmatrix} 0 \end{bmatrix}
\end{align*}
\]

### Encoding MDP Problem as Non-CNF SAT

Encoding MDP as non-CNF SAT is almost similar to that of CNF encoding, except we maintain the encoding of XOR and use the AT MOST gate to determine the correctness of the parity function. The encoding processes are as follow:

1. First we generate $y_i = s \times x_i + r_i$ as discussed in previous section. We use $k = \frac{1}{8}$ just as is used in the DIMACS CNF version.
2. Once we have computed $y_i$, we remove $s$ and replace it with variable $a \in \{0, 1\}^n$. Essentially $a$ is a solution to this parity learning problem. Variables representing $a$ appear explicitly in the theory generated so one can be assured that the problem is always satisfiable.
### 3.5. Experimental Results

Our stochastic non-CNF solver (SNCNFS) is compared with WalkSAT [118] (an incomplete CNF solver), MiniSat [39] (a complete CNF solver), and NoClause [124] (a complete non-CNF solver) on several test suites. Since our solver is incomplete, we only considered formulas that are satisfiable. For the incomplete solvers (WalkSAT and SNCNFS) we used a maximum flips of 250,000 and repeated each test 100 times. The initial starting valuations were uniformly randomly generated. We used the Random strategy of WalkSAT most similar to the search strategy defined

---

Table 3.2: Comparison of CNF and non-CNF encoding of MDP problem

<table>
<thead>
<tr>
<th>Problem</th>
<th># Vars</th>
<th># Cls</th>
<th># Vars</th>
<th># Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>par8-1</td>
<td>350</td>
<td>1149</td>
<td>8</td>
<td>441</td>
</tr>
<tr>
<td>par16-1</td>
<td>1015</td>
<td>3310</td>
<td>16</td>
<td>1649</td>
</tr>
<tr>
<td>par32-1</td>
<td>3176</td>
<td>10277</td>
<td>32</td>
<td>6369</td>
</tr>
<tr>
<td>par8-1-c</td>
<td>64</td>
<td>254</td>
<td>8</td>
<td>36</td>
</tr>
<tr>
<td>par16-1-c</td>
<td>317</td>
<td>1264</td>
<td>16</td>
<td>66</td>
</tr>
<tr>
<td>par32-1-c</td>
<td>1315</td>
<td>5254</td>
<td>32</td>
<td>131</td>
</tr>
</tbody>
</table>

The trailing ‘c’ in the problem is the compressed version of the instance.
### 3.5. Experimental Results

#### Table 3.3: Comparative results for CNF solvers on hard random formulas

<table>
<thead>
<tr>
<th>r</th>
<th># Clause</th>
<th>% Succ</th>
<th>Mean Flips</th>
<th>Mean Time(s)</th>
<th>Mean Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.6</td>
<td>8327</td>
<td>100</td>
<td>153</td>
<td>0.0009</td>
<td>0.0600</td>
</tr>
<tr>
<td>1.8</td>
<td>9333</td>
<td>100</td>
<td>219</td>
<td>0.0014</td>
<td>0.4100</td>
</tr>
<tr>
<td>2.0</td>
<td>10388</td>
<td>100</td>
<td>700</td>
<td>0.0040</td>
<td>29.0000</td>
</tr>
<tr>
<td>2.2</td>
<td>11494</td>
<td>100</td>
<td>1975</td>
<td>0.0116</td>
<td>—</td>
</tr>
<tr>
<td>2.4</td>
<td>12508</td>
<td>100</td>
<td>9519</td>
<td>0.0406</td>
<td>—</td>
</tr>
<tr>
<td>2.6</td>
<td>13491</td>
<td>100</td>
<td>56517</td>
<td>0.3773</td>
<td>—</td>
</tr>
<tr>
<td>2.8†</td>
<td>14545</td>
<td>7</td>
<td>4543650</td>
<td>70.3112</td>
<td>—</td>
</tr>
<tr>
<td>3.0†</td>
<td>15577</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

— Max flips or max time exceeded.

above for SNCNFS. Since flips in WalkSAT are substantially faster than those in SNCNFS, whenever SNCNFS solved a larger percentage of problems we increased the maximum flips for WalkSAT to 10,000,000. This is to allow WalkSAT to have at least the same amount of time to solve the problem. The benchmarks where this occurred are marked as †. For the complete solvers, we aborted if no solution is found after running for two hours. In the tables, where solutions are found, we give the averages over the successful runs only.

#### 3.5.1 Hard Random Non-CNF Formula

The comparative results are shown in Table 3.3 and Table 3.4. For these problems the size of the non-CNF formula is not substantially smaller than that of the CNF form, so there is no advantage to the non-CNF solver in size. It is shown in the table as the value of \( r \) approaches 2.8, the instances become harder. WalkSAT has only 7% success rate in solving instances \( r = 2.8 \) and Minisat is not even capable of solving instances beyond \( r = 2.0 \). For the SNCNFS, on top of less number of flips required to solve the instance of \( r = 2.8 \), it also achieved a higher solving percentage (38%) compared to WalkSAT. NoClause performed poorly on the problem. This may give an indication that complete solvers are not good on this type of problem.

Clearly the the execution time of SNCNFS are much longer than WalkSAT for the same number of flips, illustrating the highly optimized implementation of CNF solver (WalkSAT). Obviously, by treating formula in CNF format, there is less computing resources required to determine the truth of a clause. Nevertheless, SNCNFS is capable of solving harder problems than WalkSAT, illustrating there is an advantage of treating the formula in the non-CNF form. For this class of problems the complete solvers are unable to tackle the more difficult cases.
3.5. Experimental Results

Table 3.4: Comparative results for non-CNF solvers on hard random formulas

<table>
<thead>
<tr>
<th>R</th>
<th># Nodes</th>
<th>% Succ</th>
<th>Mean Flips</th>
<th>Mean Time(s)</th>
<th>SNCNF S</th>
<th>NoClause</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.6</td>
<td>8743</td>
<td>100</td>
<td>434</td>
<td>0.2020</td>
<td>2.994</td>
<td></td>
</tr>
<tr>
<td>1.8</td>
<td>9801</td>
<td>100</td>
<td>2448</td>
<td>1.0364</td>
<td>22.121</td>
<td></td>
</tr>
<tr>
<td>2.0</td>
<td>10837</td>
<td>100</td>
<td>3256</td>
<td>1.7596</td>
<td>71.352</td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>11791</td>
<td>100</td>
<td>5282</td>
<td>3.2681</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>12749</td>
<td>100</td>
<td>19596</td>
<td>13.4205</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>2.6</td>
<td>13907</td>
<td>92</td>
<td>64783</td>
<td>51.0984</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>2.8†</td>
<td>14942</td>
<td>38</td>
<td>42775</td>
<td>37.3086</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>3.0†</td>
<td>15904</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td></td>
</tr>
</tbody>
</table>

— Max flips or max time exceeded.

3.5.2 Social Golfer Problem

Randomly generated formulas do not give much advantage to our non-CNF solver over existing CNF local search solvers. The reason is because the size of the generated formulas in CNF and in non-CNF are almost equivalent. The advantage of a non-CNF encoding is apparent when dealing with encoding of real life combinatorial problem. In this section, we compare the social golfers problem [61] in non-CNF SAT and in CNF format. The CNF encoding of social golfer problem is by Gent and Lynce [50]. In our implementation, we encode the social golfer as non-CNF SAT problem using the COUNT gate and achieved significant gain in term of problem size. Table 3.5 shows the result of our encoding as compare to that of Gent and Lynce [50]. The benefit is obvious when we have solved many instances that cannot be solved by CNF WalkSAT.

Table 3.5: Comparative results for solvers on social golfer problems

<table>
<thead>
<tr>
<th>Problem</th>
<th>% Succ</th>
<th>Mean Flips</th>
<th>Mean Time(s)</th>
<th>SNCNF S</th>
<th>Non-CNF</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-2-2†</td>
<td>4</td>
<td>288009</td>
<td>0.1968</td>
<td>100</td>
<td>0.0009</td>
</tr>
<tr>
<td>3-3-3†</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>100</td>
<td>0.0236</td>
</tr>
<tr>
<td>3-4-4†</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>100</td>
<td>0.97985</td>
</tr>
<tr>
<td>4-3-3†</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>100</td>
<td>0.025</td>
</tr>
<tr>
<td>4-4-4†</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>100</td>
<td>1.5638</td>
</tr>
<tr>
<td>5-4-4†</td>
<td>0</td>
<td>—</td>
<td>0.1</td>
<td>100</td>
<td>0.7875</td>
</tr>
<tr>
<td>5-5-5†</td>
<td>0</td>
<td>—</td>
<td>0.31</td>
<td>40</td>
<td>18.0805</td>
</tr>
<tr>
<td>6-5-5†</td>
<td>0</td>
<td>—</td>
<td>0.44</td>
<td>80</td>
<td>8.4278</td>
</tr>
</tbody>
</table>

— Max flips or max time exceeded.
3.6 Chapter Summary

We have introduced an incomplete non-clausal solver based on stochastic local search. This is the first work we are aware of which uses both negative and positive scores for evaluating the degree of “truthfulness” or “falsity” of a propositional formula. Our experiments demonstrate that on certain benchmarks, stochastic local search non-clausal solver can out perform existing incomplete CNF solver as well as complete CNF and non-CNF solvers. The results of our preliminary experiments are very promising.

It would be interesting to find more complex non-clausal benchmarks to exper-

---

3.5.3 Minimal Disagreement Parity Problem

The Minimal Disagreement Parity (MDP) Problem [74] is a well-known class of hard satisfiability problem.

The advantage of non-CNF encoding for these problems is that we can maintain the encoding of XOR and use the AT MOST gate to determine the correctness of parity function, hence reduces the problem size significantly. We compare the non-CNF encoding versus the standard CNF encoding. Table 3.6 shows the results of SNCNFS compared to WalkSAT and MiniSat (we cannot apply NoClause since it does not support the ATMOST gate). Although we are unable to beat MiniSat, SNCNFS can solve 16 bit instances which are beyond the capability of WalkSAT.\(^1\)

### Table 3.6: Comparative results for solvers on MDP problems

<table>
<thead>
<tr>
<th>Problem</th>
<th>% Succ</th>
<th>Mean Flips</th>
<th>Mean Time (s)</th>
<th>% Succ</th>
<th>Mean Time (s)</th>
<th>% Succ</th>
<th>Mean Flips</th>
<th>Mean Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>par8-1†</td>
<td>2</td>
<td>298872</td>
<td>0.149</td>
<td>100</td>
<td>0.02</td>
<td>100</td>
<td>3227</td>
<td>0.1782</td>
</tr>
<tr>
<td>par16-1†</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>100</td>
<td>0.1</td>
<td>90</td>
<td>57471</td>
<td>7.0971</td>
</tr>
<tr>
<td>par32-1†</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>par8-1-c</td>
<td>100</td>
<td>8516</td>
<td>0.0054</td>
<td>100</td>
<td>0.01</td>
<td>100</td>
<td>3123</td>
<td>0.0572</td>
</tr>
<tr>
<td>par16-1-c†</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>100</td>
<td>0.06</td>
<td>100</td>
<td>32284</td>
<td>1.1499</td>
</tr>
<tr>
<td>par32-1-c†</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

The trailing ‘c’ in the problem is the compressed version of the instance.
— Max flips or max time exceeded.

\(^1\)Systematic methods are known to work well on this class of problem. WalkSAT using the Novelty strategy can solve the compressed version of 16 bits instance, but not the uncompressed version [69].
3.6. Chapter Summary

iment on, almost all SAT benchmarks are presently in CNF. The advantage of a non-clausal solver should be more evident on large difficult benchmarks that appear in non-clausal form. There remains a great deal of scope to explore different strategies for selecting a neighborhood move. We could take into account the magnitude of the children scores in weighting the random choice of child to take. We could also extend the approach to generate a candidate set of variables to flip by keeping more than one child that can change the truth value, and then picking the variable whose flipping leads to the best overall score for the root node. We could add penalties to nodes, and learn penalties when we find ourselves in local minima (as in DLM [131]). Nevertheless, this requires in depth investigation as the implementation will not be straightforward. The reason is because the multi-level DAG tree in non-CNF formula requires a completely different way of projecting the weight as well as penalty from one level to another level and the positive and negative score makes the ordinary way of expressing weighted penalty as found in [131] become invalid. Finally, it would be worthwhile reimplementing the algorithm in Comet [102] which provides efficient and built in evaluation of invariants.
Chapter 4

University Course Timetabling Problem

Timetabling problems are continually attracting interest from researchers because they are challenging problems for Artificial Intelligence and Operation Research. They are problems of time-based planning and combinatorial optimization which tend to be solved with a cooperation of search and heuristics, which usually lead to satisfactory but sub-optimal solution. Over the years, various techniques have been developed to tackle these problems. Among the techniques, local search is one that attracted major interest due its ideal co-existence with the combinatorial optimization nature of timetabling problems. Some very good results have been reported in PATAT conference proceeding series [19, 16, 18, 17, 21] using local search techniques.

In this chapter we develop a weighted-penalty scheme (WPS) local search algorithm to solve course timetabling problems. The scheme resembles the Discrete Lagrangian Multiplication techniques used in [131, 130, 134, 135] except our WPS differs from Discrete Lagrangian methods in the way that we model our problems as discrete unconstrained problems. Our WPS method is a powerful technique whereby the search is guided by a multiplier that provides a force to determine the concentration search on difficult constraints. It is also a method that provides a mechanism to escape local minima hence can be considered as a meta-heuristic technique.

4.1 University Course Timetabling

Course timetabling is a multi-dimensional assignment problem in which events (individual meetings between students and teachers) are assigned to slots (classrooms and times) [26]. A set of constraints restricts the slot each event can take. The task
4.1. University Course Timetabling

is to find a feasible solution, i.e. an assignment of event-slot where all constraints are satisfied.

A typical university course timetabling requires assignment of $N$ number of courses over $D$ days where in each day there are $T$ time slots and $R$ number of rooms. Therefore, it can be viewed as a three dimension search space where a slot is an instance of Day ($d$), Time Slot ($t$) and Room ($r$). For example, the course CEB1114 - Algorithms and Data Structures is scheduled on Monday, 10 am -12 pm, in tutorial room 1.

The constraints in course timetabling can be divided into Hard and Soft constraints [22]. Hard constraints are constraints that must be rigidly enforced. Breaking a hard constraint results in an infeasible solution and must be repaired or rejected by the timetabling algorithm. Soft constraints on the other hand are desirable but not absolutely essential. It is usually impossible to avoid breaking at least some of them. The quality of the solution is rated by the ability of the algorithm to minimize soft constraints violations. While infeasible solutions should be rejected, intuitively a high quality solution is preferred among all the feasible solutions. Nevertheless, in practice, an infeasible solution that violates few hard constraints, but satisfies many soft constraints, may be preferred to a feasible solution but violates many soft constraints.

Constraints in course timetabling vary from one to the next institution but in general, they can be categorized as follow:

**Resource constraint** - An event may be assigned to a resource of a different type (for whatever reason). For example, a course must be scheduled in a particular room, or in a particular location (such as building), or the number of students in a room may not exceed the rooms capacity.

**Time constraint** - An event must be scheduled to a specific time (for whatever reason).

**Event constraint** - Common examples of this class of constraint are that two events must not be scheduled at the same time slot (no clash), or a set of events (such as exams) must be run simultaneously.

**Order constraint** - That an event must take place before another one.

**Spread constraint** - Events should be spread out in time. For example, no teacher should have classes consecutively.

**Preferences constraint** - These constraints are designed to produce more organised and convenient timetables. For example, a lecturer prefers to have all his lectures in three days, giving him two lecture free days.

**Continuity** - Any constraints whose main purpose is to ensure that certain features
of student timetables are constant or predictable. For example, lectures for the same course should be scheduled in the same room, or at the same time of day.

Each institution will apply some or all of these constraints. The exact form will be dependent on the institution, and some may be treated as soft constraints.

In addition to the course timetabling, there is also another class of timetabling problem faced by academic institution, that is the examination timetabling. Examination timetabling have many typical differences than that of course timetabling, but the most common differences are that classes are scheduled into more than one session in the timetable but examinations are scheduled into only one, and that room allocation is normally considered in a standard course timetabling problem but not in examination timetabling (rooms are normally allocated after the timetable has been generated) [95]. We have so far not known of any claim that one class of problem (course timetabling) is conclusively more difficult than the other class of problem (examination timetabling) and vice versa, but for either case, the difficulty is highly dependent on the number of constraints as well as the size of problem to be solved and the availability of resources. In this chapter, we will focus our discussion on course timetabling, typically course timetabling in a university.

### 4.2 Methods in Course Timetabling

A wide variety of approaches have been developed over the years to solve timetabling problems. Carter and Laporte [25; 26] grouped these approaches into four categories. These are: (1) sequential methods, (2) cluster methods, (3) constraint based methods, and (4) meta-heuristic methods.

#### 4.2.1 Sequential Approaches

Sequential approaches are one of the earliest approaches used to solve timetabling problems [24]. They use a constructive approach that orders events using domain heuristics and then assigns the events sequentially into valid time periods so that no events in the period are in conflict with each other. The timetabling problems are usually represented as graphs where events (courses) are represented as vertices, while conflicts between the events are represented by edges [34]. The ordering heuristic is usually determined via the attributes of the graph. Some commonly used ordering heuristics are:

- Largest conflict first: Events that have a large number of conflicts with other events are scheduled first. The rationale is that the events with a large number of conflicts are more difficult to schedule and so should be tackled first.
4.2. Methods in Course Timetabling

- Largest weight first: Instead of using the number of conflicts as the determining factor, weight is assigned to each node based on given criteria.

- Worst first: In each step of the timetable construction an event which has the smallest number of valid periods available for scheduling in the timetable constructed so far is selected.

The problem with this method is that it is greedy in nature, and therefore unable to see the problems that may be created for future allocation by the decisions already made. As the algorithm progresses it may move to a partial solution for which it is not possible to find a feasible solution, irrespective of any future allocations.

4.2.2 Cluster Approaches

In these methods events are grouped into a number of clusters, where any two events in a particular cluster do not conflict with each other. Then the algorithm will attempt to optimize the solution to satisfy the soft constraints. This approach is found in many examination timetabling literatures, see [141, 142, 44, 10, 86]. The main drawback is that the clusters of events are formed and fixed at the beginning of the algorithm and that may result in a poor quality timetable.

4.2.3 Constraint Programming Approaches

Constraint programming (CP) is a widely used approach to solve timetabling problems. The reason is because timetabling problems are naturally a constraint satisfaction problem (CSP). In a timetabling problem, each event has a variable whose domain will be the set of sessions in which this event is allowed to be scheduled. This is indeed a finite domain constraint satisfaction problem, where each variable has an associated finite set of domain value.

When solving timetabling problems using CP, the problems are first modeled as finite set CSP. Given a timetabling problem as CSP, we can apply build-in finite domain solver found in many CP tools such as CHIP[1][35], ILOG[62], ECLiPSe[136] to obtain the desired solution. The approach of CP is fundamentally different to the heuristic approaches discussed later, as it aims to provide an exact solution. Like sequential construction heuristics, constraint programming is a means of producing a solution. The difference between the two methods is that the aim of CP is solely feasibility, and attempt at optimisation is rarely made.

There have been many attempts to solve timetabling problems using CP technique, for example see [38, 60, 83, 57, 3] and the PATAT conference proceedings
The benefit of using a CP approach is that the declarative nature of CP languages allow programmers to express information in its natural conceptual form, free of procedural consideration. This means that programmers are only concerned with ‘what’ needs to be defined for a program and not ‘how’ it should be implemented.

Nevertheless, there are some drawbacks with this technique. CP appears to be a new paradigm in computer problem solving and for many applications. The distance between the model and the constraint program is still large, mainly due to the peculiarities of the host programming language and the lack of support for modeling. CP also performs poorly on optimisation problem as opposed to other local search rivals. Although constraints are convenient mechanism for defining relation among objects, their power leads to a weakness. As the generality of these algorithms increases, so does their run time. Thus to solve a variety of constraints both efficiently and correctly, a broad range of algorithms is necessary.

### 4.2.4 Meta-Heuristic Local Search Approaches

A typical local search technique model timetabling problems in relation to some objective function $f(x)$ whereby the goal is to minimize the objective function. The objective function is usually derived from the sum of violated constraints in a particular complete solution. A perfect solution is achieved when $f(x) = 0$. Equation 4.1 shows a typical objective function formulation.

$$\min f(x) = \sum_{i=1}^{n} g_i(x)$$

where $g_i(x) =$ score for constraint $i$, $\forall i \in \{1, 2, ..., n\}$

Local search algorithms start from a complete assignment and move from complete assignments to neighboring complete assignments in order to obtain a better solution (i.e. minimizes the objective function). The move heuristic can be simple (change the value of a variable or swap the value of variables) or involved more complicated strategy. Simple, problem-specific heuristic methods can produce fast and good result, but as the size and complexity increase, local search algorithms need to employ some form of techniques to escape local minima.

In the recent years, a variety of meta-heuristic techniques have been investigated for timetabling. Techniques such as simulated annealing [4], tabu search [66], iterated local search [90], genetic algorithms [5, 14], and memetic algorithms [20] have been used. Comprehensive compilation of the results of these techniques can be found in the Practice and Theory of Automated Timetabling conference proceeding.
4.3 Course Timetabling Benchmarks

In this thesis, we use the the timetabling specification used in [114]. It is a simplified version of a typical university course timetabling introduced by Ben Paechter\textsuperscript{1} to reflect aspects of Napier University’s real timetabling problem. It is also the benchmark used for the international timetabling competition 2003 [108] organized by the Metaheuristics Network [2]. The benchmark consists of a set of events to be scheduled over five days of nine hours each, a set of rooms in which events can take place, a set of students who attend the events, and a set of features satisfied by rooms and required by events. Each student attends a number of events and each room has a size. Each event has a duration of one hour. A feasible timetable is one in which all events have been assigned a timeslot and a room so that the following hard constraints are satisfied:

- no student attends more than one event at the same time;
- the room is big enough for all the attending students and satisfies all the features required by the event;
- only one event is in each room at any timeslot.

In addition, a candidate timetable is penalised equally for each occurrence of the following soft constraint violations:

- a student has a class in the last slot of the day;

\textsuperscript{1}School of Computing, Napier University
4.3. Course Timetabling Benchmarks

- a student has more than two classes consecutively;

- a student has a single class on a day.

In this benchmark, the soft constraints have been chosen to be representative of three different classes: the first one can be checked with no knowledge of the rest of the timetable; the second one can be checked while building a solution, taking into account the events assigned to nearby timeslots; and finally the last one can be checked only when the timetable is complete, and all events have been assigned a timeslot[114].

4.3.1 A Simple Timetabling Example

We define a reduced-size course timetabling problem. The problem will be used throughout the thesis when explaining our proposed timetabling algorithm. Let’s assume that the timetable is scheduled over two days where there are three slots in each day. There are three rooms and each has the features as shown in Table 4.1. There are ten events and ten students. Table 4.2 and Table 4.3 shows the features required by each event and the student-event subscription respectively.

<table>
<thead>
<tr>
<th>Room</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1$</td>
<td>$f_1$</td>
</tr>
<tr>
<td>$r_2$</td>
<td>$f_1, f_2$</td>
</tr>
<tr>
<td>$r_3$</td>
<td>$-$</td>
</tr>
</tbody>
</table>

Table 4.1: Room Features

<table>
<thead>
<tr>
<th>Event</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_1$</td>
<td>$f_1$</td>
</tr>
<tr>
<td>$e_2$</td>
<td>$f_1, f_2$</td>
</tr>
<tr>
<td>$e_3$</td>
<td>$-$</td>
</tr>
<tr>
<td>$e_4$</td>
<td>$-$</td>
</tr>
<tr>
<td>$e_5$</td>
<td>$f_1$</td>
</tr>
<tr>
<td>$e_6$</td>
<td>$f_2$</td>
</tr>
<tr>
<td>$e_7$</td>
<td>$-$</td>
</tr>
<tr>
<td>$e_8$</td>
<td>$-$</td>
</tr>
<tr>
<td>$e_9$</td>
<td>$f_1$</td>
</tr>
<tr>
<td>$e_{10}$</td>
<td>$f_1, f_2$</td>
</tr>
</tbody>
</table>

Table 4.2: Features required by each event
### 4.4 Timetabling Problem Modeling

The timetable is represented as a $2 \times 2$ matrix. The timeslots and slots are enumerated as a continuous sequence of integers. A slot is an instance of day $(d_i)$, timeslot $(t_i)$ and room $(r_i)$ hence will be represented as triplet,

$$ s_i = (d_i, t_i, r_i). $$

The integer-based slots can be computed as:

$$ s_i = r_i + (t_i - 1) \times R $$

where $1 \leq r_i < R$ and $1 \leq t_i < T$  

Note that the computation of slots does not require the instance of day, but day is included to assist in the modeling process.  

Given our simple course timetable in Section 4.3.1, the example of the timetable and its enumeration of slots is shown in Figure 4.1.

**Definition 4.4.1** \textit{feature}($e_i$) $= (f_1, f_2, ..., f_n)$ defines that event $e_i$ requires features $f_1, f_2, ..., f_n$. The total number of features is $n$.

**Definition 4.4.2** \textit{feature}($r_i$) $= (f_1, f_2, ..., f_n)$ defines that room $r_i$ has features $f_1, f_2, ..., f_n$. The total number of features is $n$.

**Definition 4.4.3** \textit{event}($e_i$) $= (a_1, a_2, ..., a_n)$ defines that event $e_i$ is taken by student $a_1, a_2, ..., a_n$. The total number of students is $n$.

**Definition 4.4.4** \textit{student}($a_i$) $= (e_1, e_2, ..., e_n)$ defines that student $a_i$ taking a series of events $e_1, e_2, ..., e_n$. The total number of course taken is $n$.

<table>
<thead>
<tr>
<th>Student</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>$e_1, e_2, e_7$</td>
</tr>
<tr>
<td>$a_2$</td>
<td>$e_1, e_2, e_3, e_9$</td>
</tr>
<tr>
<td>$a_3$</td>
<td>$e_3, e_4, e_7, e_{10}$</td>
</tr>
<tr>
<td>$a_4$</td>
<td>$e_4, e_5$</td>
</tr>
<tr>
<td>$a_5$</td>
<td>$e_4, e_6, e_8$</td>
</tr>
<tr>
<td>$a_6$</td>
<td>$e_1, e_2, e_7$</td>
</tr>
<tr>
<td>$a_7$</td>
<td>$e_3, e_6, e_9, e_{10}$</td>
</tr>
<tr>
<td>$a_8$</td>
<td>$e_5, e_8, e_9, e_{10}$</td>
</tr>
<tr>
<td>$a_9$</td>
<td>$e_2, e_{10}$</td>
</tr>
<tr>
<td>$a_{10}$</td>
<td>$e_5, e_6, e_8$</td>
</tr>
</tbody>
</table>

Table 4.3: Students subscription to events
4.4. Timetabling Problem Modeling

<table>
<thead>
<tr>
<th>Room</th>
<th>Day 1</th>
<th>Day 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$t_1$</td>
<td>$t_2$</td>
</tr>
<tr>
<td>$r_1$</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>$r_2$</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>$r_3$</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Figure 4.1: Example of empty timetable slots

**Definition 4.4.5** $\text{capacity}(r_i) = k$ indicates that the capacity of room $r_i$ is $k$.

**Definition 4.4.6** An assignment is defined as $\text{slot}(e_i) = s_i$ such that event $e_i$ is assigned to slot $s_i$.

**Definition 4.4.7** The solution to the problem is a set $P$ of $\text{slot}(e_i)$ which satisfies all the constraints.

Having established the model of our timetabling problem, we can define the constraints as follows:

Constraint 1: No two events are assigned the same slot. [hard constraint]

$$\forall e_1, e_2 : e_1 \neq e_2 \Rightarrow \text{slot}(e_1) \neq \text{slot}(e_2)$$

(4.4)

Intuitively, it means if event $e_i$ has been assigned slot $s_i$, no event other than $e_i$ can be assigned to slot $s_i$. This is indeed an alldifferent constraint for the event-slot pairs.

Constraint 2: A student should not attend more than one event at the same time. [hard constraint]

$$\forall a \forall e_i \forall e_j \forall s_i \forall s_j \forall t_i \forall t_j$$

$$\text{student}(a) = (e_i, e_j),$$

$$\text{slot}(e_i) = s_i, \text{slot}(e_j) = s_j,$$

$$s_i = (\_ , t_i , \_), s_j = (\_ , t_j , \_),$$

$$t_i \neq t_j$$

(4.5)

Intuitively, it means if student $n$ takes event $e_i$ and $e_j$, then the slot for $e_i$ and $e_j$ must not have the same time slot. This is indeed an alldifferent constraint for time slot.
Constraint 3: Room has the features required by the event. [hard constraint]
\[ \forall e \forall r, \quad \text{slot}(e) = \langle \_, r \rangle \Rightarrow \text{feature}(e) \subseteq \text{feature}(r) \] (4.6)

Constraint 4: The room is big enough for the total number of students. [hard constraint]
\[ \forall e \forall r, \quad \text{slot}(e) = \langle \_, r \rangle \Rightarrow |\text{event}(e)| \leq \text{capacity}(r) \] (4.7)

Constraint 5: A student should not have a class in the last slot of the day. [soft constraint]
\[ \forall e, \forall a \in \text{event}(e), \quad \text{slot}(e) = \langle \_, t, \_ \rangle \Rightarrow t \mod T \neq 0, \]
where \( T \) is the total slots per day
(4.8)

Constraint 6: A student has more than two classes consecutively. [soft constraint]
\[ \forall a \forall t_i \forall t_j, \forall e_i, e_j \in \text{student}(a) \]
\[ \text{slot}(e_i) = \langle t_i, \_ \rangle, \text{slot}(e_j) = \langle t_j, \_ \rangle \Rightarrow t_i \neq t_j - 1 \] (4.9)

Constraint 7: A student should not have a single class on a day. [soft constraint]
\[ \forall a \forall d, \exists e_i, e_j \in \text{student}(a) \]
\[ \text{slot}(e_i) = \langle d, \_, \_ \rangle, \text{slot}(e_j) = \langle d, \_, \_ \rangle, e_i \neq e_j \lor \]
\[ \neg \exists e_i, \text{slot}(e_i) = \langle d, \_, \_ \rangle \] (4.10)

4.5 Solution Representation

The timetabling problem is represented with two arrays of objects. The first array is the array of event objects, \( \epsilon \) of length \( E \) (total event) where event object \( i \) corresponds to event \( i \) and is in the position \( i \) of the event array. The second array is the array of student objects, \( \alpha \) of length \( A \) (total student). For each event object, there is a linked list pointing to the list of students taking the event. In addition, there is another linked list of features required by the event. Every event object has
4.6 Searching for Solution

fields for room, day, timeslot and slot. For each student object, there is a linked list pointing to the list of events taken by the student.

Similar to [114], the room assignments are not part of the explicit representation, rather a generator randomly assigns room to each event and a matching algorithm ensures that the room has the feature required by the event and is sufficient in size.

4.6 Searching for Solution

The process of searching for solution has two phases. The first phase is to search for a feasible solution, i.e. a solution that satisfies all the hard constraints. The second phase is to optimize the solution to minimize soft constraint violations.

4.6.1 Searching for Feasible Solution

We use a weighted penalty scheme (WPS) method of local search in searching for an initial feasible timetable. This method resembles the Discrete Lagrangian Method (DLM) [133] for solving discrete constraint optimization problem except that our WPS modeled problem in discrete unconstrained formulation whereas problems in DLM are modeled as discrete constraint optimization problem.

Another form of meta-heuristic that is close to our WPS is Guided Local Search (GLS) [128, 127]. GLS is the generalisation of GENET [138, 32] which sits on top of any local search algorithm to penalize undesirable solutions (local minimas). The penalty acts like a weighting function that will make the local minimum more costly than the surrounding search space, where these features are not present, thus guiding the local search algorithm out of the local minimum. The primary different between GLS and WPS is that GLS is built on top of local search. For that, GLS requires definition of the features to be penalized and therefore depends on the type of problem. WPS on the other hand is built into the local search itself and therefore has the pure focus to change the penalty (instead of penalizing the feature). This makes WPS independent from the type of problem. Similar to GLS, WPS also provides a mechanism to dynamically determine the importance of a constraint. For certain constraints that are very difficult, the multiplier increases their importance forcing the algorithm to give priority to the satisfiability of these constraints. In addition, the weighted penalty can provide a breakout force when search is stuck in local minima.
4.6. Searching for Solution

**Figure 4.2: WPS: Escaping local minima**

**Penalty Formulation**

We define our penalty function $f(x)$ as

$$f(x) = \sum_{i=1}^{n} g_i(x) + \lambda_i c_i(x)$$

where $g_i(x) = $ score for hard constraints

$\lambda_i = $ weight factor for $g_i(x)$

$$c_i(x) = \begin{cases} 
1 & \text{if } g(x) > 0 \\
0 & \text{if } g(x) = 0 
\end{cases}$$

This definition gives a weighted penalty scheme for our local search algorithm. By controlling the weight factor $\lambda_i$, we can control the degree of importance of a particular constraint. $\lambda_i$ are positive integers. The larger the value of $\lambda_i$, the larger the value of the penalty function. Similar constraint can carry different penalty value as a result of $\lambda_i$. This provides a powerful force for search to escaping local minima. For instance, in Figure 4.2 (a), the search is stuck in local minima. At this point $\lambda_i = 0$. As the value of $\lambda_i$ increases from 0 to 3 (Figure 4.2 (b), (c), (d)), it pushes the penalty value to be greater than that of its neighbors, hence leading the search out of local minima.
Neighborhood structure

We define two variations of move. \( N_1 \) defines a move to an empty location. \( N_2 \) defines a swap between two event (i.e. exchange of two non-empty location). Having defined our timetabling model as in 4.2, it allows us to define several neighborhood strategies using the moves defined above. The neighborhood strategies are:

- **M1** – Move from one slot to another slot, either by \( N_1 \) or \( N_2 \).
- **M2** – Move from one timeslot to another timeslot, either by \( N_1 \) or \( N_2 \).
- **M3** – Move from one room to another room, either by \( N_1 \) or \( N_2 \).
- **M4** – Move from one day to another day, either by \( N_1 \) or \( N_2 \).

M1 is the type of move that affects the day, timeslot and room. M2 is the type of move that only affects the day and the timeslot. M3 is the type of move that only affects the room and M4 only affects the day. All these moves are useful when defining our move heuristic functions.

Move Heuristic

The search is guided by the following move heuristics.

**H1** - Move that minimizes event-slot violations (constraint 1).

By restricting moves to only an empty slot, this will minimize event-slot violation.

**H2** - Move that minimizes event-student violations (constraint 2).

Event-student violations are the result of clashing of timeslot. Therefore, by instituting moves that only involve the timeslot, it will reduce the event-student violation (without affecting event-room violations).

\[
 s_i = (d_i, t_i, r_i) \rightarrow (d_i, t_j, r_i)
\]

where \( t_i \neq t_j \)

**H3** - Move that minimizes event-room violations (constraint 3 and 4).

Event-room violations are the result of incompatible room. Therefore, by instituting moves that only involve the room, it will reduce the event-room violation (without affecting event-student violations).

\[
 s_i = (d_i, t_i, r_i) \rightarrow (d_i, t_i, r_j)
\]

where \( r_i \neq r_j \)
H4 - Move that minimizes the overall objective function.

This involves free movement to any of the slot that minimizes the objective function.

The algorithm

The general WPS algorithm for solving course timetabling problem is shown in Algorithm 4.1.

**Algorithm 4.1 Weighted Penalty Scheme Timetabling Algorithm**

1. Procedure WPS(T)
2. \( s := \text{GenerateInitialSolution}(T) \)
3. \( s^* := s \)
4. For \( k = 0 \) To MaxTries Do
5. \( h := \text{Heuristic}(H1, H2, H3, H4) \)
6. \( s' := \text{Move}(s, h) \)
7. If \( f(s') \leq f(s) \) Then
8. \( s^* := s' \)
9. \( s := s' \)
10. Else
11. \( \text{IncreaseMultiplier}(s) \)
12. End If
13. If Constraint(s) == 0 Then
14. Break
15. End If
16. End For
17. Return \( s^* \)
18. End WPS

The algorithm starts by generating an initial solution (line 2). The initial solution is a random complete assignment of event-slot. At this point, the initial solution is the best solution, thus it is remembered at \( s^* \) (line 3). Next, the algorithm enters into the search loop and is repeated for MaxTries attempts. At each loop, the search first determine the type of heuristic move \( H1, H2, H3 \) or \( H4 \). Then it performs one of the move strategy \( M1, M2, M3 \) or \( M4 \) depending on the type of heuristic chosen. For instance if the heuristic is \( H2 \), then the move is restricted to \( M2 \), or if the heuristic is \( H3 \) then the move is restricted to \( M3 \). Moreover, we can combine several heuristics together at each iteration.

If the move results in a better or equivalent solution (based on the score), the move is accepted, otherwise, the multiplier for the events that involved in the move is increased by 1. Accepting equivalent solutions (i.e. different solution with similar score) is known as a sideways move [117] and it was shown earlier in Selman et al.
4.7. Implementation

[117] that sideways moves can result in significant improvements to local search. The loop terminates if all the constraints are satisfied, or when the loop exceeds MaxTries.

4.6.2 Searching for Quality Solution

As we have mentioned earlier, the quality of a timetabling solution is measured by the number of soft constraint violations. In our implementation, we did not give much focus on improving the quality of solution as our aim is to solve hard constraint using WPS. Furthermore, it is difficult to incorporate WPS side by side with soft and hard constraints due to the reason that if weight is to be given to them, then the weight would increase their importance irrespective of whether they are soft or hard constraints. In this case, we cannot concentrate the search to give precedence to satisfying the hard constraint, thus it can result in a solution that may satisfy many soft constraints, but is not guaranteed to be feasible. In our implementation, once a feasible solution is found, we simply resort to greatest-descent search. Using this method, we simply accept moves that equal (sideways) or improve the current score.

4.7 Implementation

We implemented our weighted-scheme local search algorithm for course timetabling problems in C++ using the technology of invariants as discussed in Chapter 2. The score of a solution can be easily obtained by summing up the penalty value for constraint violations. Indeed we need to maintain an invariant of \( v = g(x) \) where \( g(x) \) is the sum of constraint violations. Our One-Way-Solver will appear handy in this situation where the invariant can be implemented using the in_list data structure via the makeSum feature. The makeSum invariant will allow quick and efficient implementation of summing up the value of constraint violations and propagates updates incrementally when the penalty values change.

Furthermore, for constraint such as Constraint 1: No events are assigned the same slot, and Constraint 2: A student should not attend more than one event at the same time, are indeed an alldiff constraint. We can use the makeAllDifferent feature in in_list to return the penalty value of the alldifferent list. If the constraints are satisfied, the makeAllDifferent will return 0 penalty score, otherwise a score equivalent to the number of violations as discussed in Section 2.2.3.
4.8 Experiment Results

We tested our algorithm on benchmark data provided by Rossi-Doria et al. [114]. The complete set of data can be downloaded from http://iridia.ulb.ac.be/~msampels/ttnn.data. All the instances in the benchmark have perfect solutions, i.e. solutions with no constraint violations, hard or soft. It contains three classes of instances of different sizes called small, medium and large. They have the attributes as shown in Table 4.4.

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Small</th>
<th>Medium</th>
<th>Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of events</td>
<td>100</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>Number of rooms</td>
<td>5</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Number of features</td>
<td>5</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>Approximate feature per room</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Percent feature use</td>
<td>70</td>
<td>80</td>
<td>90</td>
</tr>
<tr>
<td>Number of student</td>
<td>80</td>
<td>200</td>
<td>400</td>
</tr>
<tr>
<td>Max events per student</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Mas student per event</td>
<td>20</td>
<td>50</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 4.4: Attributes of benchmark instances

The experiments were tested on a PC with Intel Centrino 1.6 Mhz with 758 Mb of RAM. We follow the time frame used by Rossi-Doria et al. [114], i.e. 90 seconds for small instances, 900 seconds for medium instances and 9000 seconds for large instances. For every instance, we ran 10 independent trials and took the average result. We tested the instances with two variants of move. The first variant uses moves that only minimize the overall objective function (i.e. H4). The intuition behind this attempt is to test our WPS algorithm using moves that are independent of problem-specific knowledge, hence called “naive move.” Naive move provides a fair comparison with the techniques used in [114]. Since there is no problem specific knowledge involved in the search process, consequently the outcome is purely the result of the metaheuristic. The second variant of moves are guided by the move heuristics defined earlier (i.e. H1, H2, H3 and H4). The intuition is to test the effect of adding problem specific knowledge to our WPS algorithm. We call this attempt as “heuristic guided moves.”

Table 4.5 shows the results of our WPS on all the problem instances. The results show the time required to reach feasible solution and the score achieved after the preset amount of time (i.e. 90s for small instances, 900s for medium instances and 9000s for large instances). The score is obtained by running the final result on the standard solution checker provided by [114].

We compare our results using only the naive move with the results from the
4.8. Experiment Results

<table>
<thead>
<tr>
<th>Problem</th>
<th>Feasible Time (s)</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>small01</td>
<td>0.023</td>
<td>7</td>
</tr>
<tr>
<td>small02</td>
<td>0.043</td>
<td>8</td>
</tr>
<tr>
<td>small03</td>
<td>0.018</td>
<td>6</td>
</tr>
<tr>
<td>small04</td>
<td>0.012</td>
<td>7</td>
</tr>
<tr>
<td>small05</td>
<td>0.04</td>
<td>4</td>
</tr>
<tr>
<td>medium01</td>
<td>6.6206</td>
<td>368</td>
</tr>
<tr>
<td>medium02</td>
<td>7.4689</td>
<td>395</td>
</tr>
<tr>
<td>medium03</td>
<td>11.4717</td>
<td>410</td>
</tr>
<tr>
<td>medium04</td>
<td>7.7546</td>
<td>379</td>
</tr>
<tr>
<td>medium05</td>
<td>12.1852</td>
<td>364</td>
</tr>
<tr>
<td>large01</td>
<td>357.0834</td>
<td>948</td>
</tr>
<tr>
<td>large02</td>
<td>119.4579</td>
<td>862</td>
</tr>
</tbody>
</table>

Table 4.5: Summary of Results for All Timetabling Problem Instances

techniques used by Rossi-Doria et al. [114]. In brief, Rossi-Doria et al. [114] tested the same instances with five different meta-heuristic techniques, i.e. Ant Colony Organization (ACO), Genetic Algorithm (GA), Iterated Local Search (ILS), Simulated Annealing (SA) and Tabu Search (TS). Figure 4.3 and Figure 4.4 show the results of WPS on small and medium instances respectively and Figure 4.5 and Figure 4.6 show the result of WPS on large01 and large02 instances respectively. In comparison with the techniques in [114]. Similar to other techniques, WPS manage to achieve 100% of feasible solution in all of these problem instances. When comparing the score for soft constraint violation, WPS is not so impressive as compared to other techniques. This is most likely due to the fact that our WPS only applies to hard constraints. When a feasible solution is found, the algorithm simply resorts to greatest-descent to minimize soft constraint violation.

Figure 4.7 and 4.8 show the percentage of invalid solutions (i.e. no feasible solution found after preset time frame) for large instances. WPS has achieved 100% feasible solution for both instances whereas other techniques perform poorly either on large01 or large02 problem or on both of them. Comparison for the score of soft constraint violation is not possible because the result is not published by Rossi-Doria et al. [114]. These results are very promising as our WPS is far superior to the other techniques with regard to searching for feasible solution.

The results have also shown that the move heuristic does not persistently guarantee better results (although in many cases it does). In the medium instances (Figure 4.9), the move heuristic adds significant improvement to finding the feasible solution. Conversely, in large instances (Figure 4.10), the naive move outperforms heuristic move. The most likely reason for this behavior is as heuristics are generally greedy
and based on rule of thumb, they do not always work. The consequence of applying a heuristic into local search is highly problem-dependent hence is influenced by the structure of the problem.

### 4.9 Chapter Summary

We have developed a course timetabling algorithm using weighted penalty scheme (WPS) local search technique. We have shown that WPS is superior to the techniques like Ant Colony Organization (ACO), Genetic Algorithm (GA), Simulated Annealing (SA), Iterated Local Search (ILS), and Tabu Search (TS) in terms of searching for feasible solution. We did not apply WPS onto soft constraints due to the reason that if weight is to be given to them, then the weight would increase their importance to a point that they would eventually behave like a hard constraints. Nevertheless, we have shown that WPS has its own unique strength. Further research is required to find out how WPS can be applied to soft constraints. As the results of WPS on hard constraints are very promising, we believe that similar success can be replicated if we can successfully incorporate WPS to tackle soft constraints.
Figure 4.4: Result of soft constraint violation for medium instances

Figure 4.5: Result of soft constraint violation for large01 instances
4.9. Chapter Summary

Figure 4.6: Result of soft constraint violation for large02 instances

Figure 4.7: Percentage of invalid solution for large01
4.9. Chapter Summary

Figure 4.8: Percentage of invalid solution for large02

Figure 4.9: Comparison for Feasible Time between Naive and Heuristic Move for Medium Instances
Figure 4.10: Comparison for Feasible Time between Naive and Heuristic Move for Large Instances
Chapter 5

Conclusions

This thesis describes work on local search techniques for constraint problems. Local search has become the subject of intense research in recent years giving birth to many innovative tools and techniques in solving constraint problems. Many industries have since benefited from the progress in the field as it helps industries to address critical situations and to become more reactive and cost-effective. This chapter draws the conclusions from the research results obtained, and looks at some future work in relation to the work done.

5.1 Invariants

There is no argument that local search is one of the most important paradigms for solving constraint problems. The majority of the research work done in the local search community is about the low-level implementation aspects of local search algorithm. Neighborhoods, local moves, heuristics and meta-heuristics are among the key issues in local search. While these issues are crucial, there are other aspects of local search that are equally important. From the implementation standpoint, we need a unifying theme to present local search to the general public. While the GUI communities have rich libraries to describe various aspects of user interface, the networking communities are also not lacking in their SOCKET library. The challenge lies at our capability to present local search in the similar fashion.

In this thesis, we have proposed a series of programming libraries known as One-Way-Solver. One-Way-Solver combines the aspects of declarative and imperative programming to ease the process of implementing local search. What is more important is that we want to clear the preconception that local search is a collection of heterogeneous techniques lacking a unifying theme. Our One-Way-Solver shows that there is a possibility of designing local search algorithms in high-level language...
structure that offers modeling, search abstractions, reuse, separation of concerns and modularity. This is the point of benefit for local search from software engineering standpoint.

Future work on these topics has the potential to deliver significant improvements in term of performance and modeling capability of the language. Several attempts have shown promising progress in the field, for instance the introduction of COMET by Michel and Hentenryck [102] in 2001, is the first modeling language for local search that uses constraints like those found in constraint programming languages to describe and control local search. Nevertheless, much improvement is still required. Local search modeling appears to be a new paradigm in computer problem solving and for many applications. The distance between the model and the constraint programs is still large, mainly due to the peculiarities of the host programming language and the lack of support for modeling. Although constraints are convenient mechanism for defining relation among objects, their power leads to weaknesses. As the generality of these algorithms increases, so does their run time. In addition, most of the practical problems require some form of problem-specific information to be incorporated into the search process. It is still unclear how problem-specific information can be incorporated into the search without loosing the generality of the language.

5.2 Non-CNF SAT

In this thesis, we have also introduced a non-CNF SAT solver using local search technique. To our knowledge, this is the first most comprehensive work on non-CNF SAT solver particularly using local search technique. Earlier implementations of non-CNF solvers are mainly extension of clausal solver to take formula in free form, but our solver has included the implementation of almost all the known Boolean gates.

Non-CNF SAT solving is gaining popularity in recent years as the progress in CNF-based SAT solver though without any sign of slowdown seems to be in the stage of plateau. Many researchers have pointed out that non-CNF SAT solving will be the next top ten new challenges for SAT [137]. Our work in this thesis has produced promising results in the area and show that there is huge potential in solving SAT problem in non-CNF format, particularly using local search techniques. Much work remains to be done as there are still many things that are not well understood about non-CNF SAT solving. For example, currently the heuristic strategy design is still an art instead of a science. It is very hard to explain why a certain heuristic
strategy works better than others. Furthermore, we still do not understand why certain problem can be solved better in non-CNF format compared to their CNF counterpart and some problems are better on complete solvers as compared to their incomplete counterpart. By getting a better understanding of the solving process, the advantage of a non-clausal solver should be more evident. This knowledge is a vital guide for us in choosing the right solver for the right problem.

5.3 Timetabling

In this thesis, we have also developed a weighted penalty scheme (WPS) local search technique and applied our WPS onto university course timetabling problem. WPS method is a new breed of local search meta-heuristic technique where the technique is simple and straightforward. Guided by a multiplier, the search can be directed to determine the concentration search as well as to escape local minima. We have shown that on our timetabling benchmarks, WPS has outperformed several local search meta-heuristic techniques.

Further research is required to better understand the behavior of WPS particularly in solving optimization problems. In addition, there are other potential applications of WPS which can be discovered by conducting more theoretical as well as empirical research on different classes of constraint problems.

In conclusion, the work in this thesis demonstrates the theory and application of local search methods for constraint problems. It is our belief that there is a bright future for local search techniques and there is plenty of room for improvement. It is our hope that this work has provided some useful insights in our pursue to understand and benefit from this unique class of problem solving technique.
Bibliography


Minerva Access is the Institutional Repository of The University of Melbourne

Author/s:
Muhammad, Muhammad Rafiq Bin

Title:
Local search methods for constraint problems

Date:
2008-02

Citation:

Publication Status:
Unpublished

Persistent Link:
http://hdl.handle.net/11343/39367

File Description:
Local search methods for constraint problems

Terms and Conditions:
Terms and Conditions: Copyright in works deposited in Minerva Access is retained by the copyright owner. The work may not be altered without permission from the copyright owner. Readers may only download, print and save electronic copies of whole works for their own personal non-commercial use. Any use that exceeds these limits requires permission from the copyright owner. Attribution is essential when quoting or paraphrasing from these works.