Towards Effective Parameter-Free Clause Weighting Local Search for SAT

by

Abdelraouf Ishtaiwi

BSc in Computer Science (1997), Al-Isra University, Jordan

Masters of Information and Communication Technology, Griffith University, Australia (2002)

A thesis submitted in fulfillment
of the requirements of the degree of
Doctor of Philosophy

Institute for Integrated and Intelligent Systems
Faculty of Engineering and Information Technology
Griffith University, Queensland
Australia

September, 2007

Abstract

Recent research has shown that it is often preferable to encode real-world problems as propositional satisfiability (SAT) problems, and then solve them using general purpose SAT solvers. However, most SAT solvers require the tuning of parameters in order to obtain optimum performance. Tuning these parameters usually takes a considerable amount of time, and even to achieve average performance can require many runs with many different parameter settings. In this thesis we investigate various ways to improve the overall performance of local search solvers via new techniques that do not employ parameters and therefore take considerably less time for experimentation. A summary of our main contributions is as follows:

- Firstly, we investigate possible ways to improve the current state-of-the-art for dynamic local search SAT techniques. This includes the development of a neighbourhood weight transfer mechanism (part of this study was published at the 2005 Principles and Practice of Constraint Programming Conference, CP-05).

- Secondly, we improve the neighbourhood weight transfer mechanism with an adaptive heuristic that alters the process of weight transference during the search. We show that our new algorithm has the best performance in comparison to the current state-of-the-art SAT techniques on a range of real-world problems (part of this study was published at the 2006 Principles and Practice of Constraint Programming Conference, CP-06).

- Thirdly, we investigate the performance of the neighbourhood weight transfer mechanism in combination with a preprocessing technique. We conclude that the preprocessing technique improves the performance of the new algorithm over a variety of structured problems, including planning and graph colouring problems. However, we found that preprocessing had no effect when used on unstructured, randomly generated problems (part of this study was published at the 2006 Principles and Practice of Constraint Programming conference,

CP-06).

- Fourthly, we investigate a natural extension of the new approach to handle over-constrained unweighted MAX-SAT problems. To evaluate this we conducted an extensive experimental study to compare the performance of the new approach with the current state-of-the-art techniques that also do not require any parameter tuning (part of this work has been accepted for publication at the 2007 Australian Joint Conference on Artificial Intelligence, AI-07).

- Finally, we extend the weight transfer mechanism to handle over-constrained problems with hard and soft constraints and compare this approach with a previously best performing local search technique.

Statement of Originality

This work has not previously been submitted for a degree or diploma to any university. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made in the thesis itself.

Abdelraouf Ishtaiwi

September, 2007

Contents

List of Figures

List of Tables

Acknowledgements

This work was not possible in any way without the support and encouragement of my supervisors Dr John Thornton and Professor Abdul Sattar. I am grateful for all the discussion and the guidance by my supervisors that kept me on the right track.

Also I would like to dedicate this work to my wife Raghda and my son Mohammed who are the joy of my life. I would like also to thank my family back in Jordan, my dad Mohammed, my mother Huda, my brother Osama, and my sisters. Special thanks to my uncle Yousef and uncle Fathe whose support was one of the reasons that I could come to do PhD.

Finally, I would like to thank all my colleagues and friends for their help, kindness and helpful comments during the process of completing this thesis and to gratefully acknowledge the financial assistance provided by Griffith University's Institute for Integrated and Intelligent Systems (IIIS) Research Centre and by National ICT Australia (NICTA). NICTA is funded by the Australian Government's *Backing Australia's Ability* initiative, and through the Australian Research Council.

Chapter 1

Introduction

In this chapter, we briefly review the concept of a constraint satisfaction problem (CSP) and of a propositional satisfiability (SAT) problem. We then describe the research problems addressed in this thesis and our motivations to carry out this research. Finally, we provide a summary of the main contributions of the thesis and an outline of the remaining chapters.

1.1 Background

Many problems in artificial intelligence (AI) and computer science can be formulated as constraint satisfaction problems (CSPs), including computer vision, scheduling, temporal reasoning, graph colouring, planning and the satisfiability problem (SAT). The constraint satisfaction paradigm provides a simple, but expressively rich methodology to represent complex relationships among objects in the world. Such relationships can then be exploited to find solutions efficiently. Solving CSPs has been one of the major fields of research in AI for the last three decades.

A constraint is a generic term, it can be any arbitrary relationship over objects often represented as variables, e.g. predicates, fuzzy relations, algebraic equations, etc. The solution task is to find values for variables that satisfy all the constraints of a given problem. In general a constraint satisfaction problem can be defined as a triple (X, D, C):

$$\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$$
$$\mathcal{D} = \{d_1, d_2, \ldots, d_n\}$$
$$\mathcal{C} = \{c_1, c_2, \ldots, c_n\}$$

where *X* represents a finite set of variables $x_1, \ldots x_n$, *D* is a set of finite domains for each variable in X and *C* is a finite set of constraints on an arbitrary subset of variables in X. The task of solving a CSP is to find an assignment of values $d_1, d_2, \ldots, d_n$ to variables in X that satisfies all the constraints $c_1, c_2, \ldots, c_n$.

Assume that our task is to colour the world map with only a limited number of colours and on the condition that any two adjacent countries should be assigned with different colours. To formulate this as a CSP we define the variables to be the countries. The domain of each variable is a set of colours (e.g. green, blue, and red) and the constraints are to assign all pairs of variables with distinct colours (domain values) if they are neighbours (adjacent).

In Figure 1.1, the letters A, B, C and D represent four different countries and the letters b, g, r stand for the colours blue, green and red (the domain values that can be assigned to the countries).

A CSP can have many solutions. For the above problem a possible solution is as follows:

A = blue, B = green, C = red and D = green

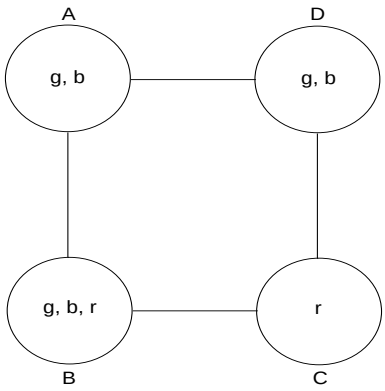


Figure 1.1: Example of graph colouring problem with 4 countries and 3 colours

Often, we encounter situations in which there is no consistent assignment of values for each variable, meaning a problem cannot be solved. These problems known as *over-constrained*. For example, in Figure 1.2 another constraint has been added to represent that country B and D are adjacent. Now, it becomes impossible to find a consistent assignment for all four countries, i.e. either B is blue and D is green, or B is green and D is blue. In either case A cannot be assigned a consistent value.

CSPs generally require sophisticated search methods to solve them efficiently. Broadly speaking, there are two classes of methods available: complete or *systematic* search and incomplete *stochastic local search* (SLS). Systematic search typically consists of a backtracking method plus

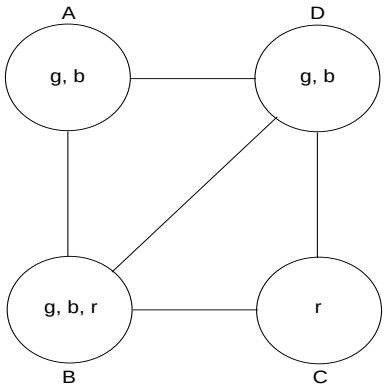


Figure 1.2: An over-constrained graph colouring problem

some performance enhancing heuristics such as forward checking, intelligent backtracking, etc. Stochastic local search is based on the iterative repair method, plus clever heuristics to get out of the local minima. A general consensus has emerged in recent years that SLS methods are useful when backtracking methods become unable to solve large sized problems within a reasonable time bound.

1.1.1 The Propositional Satisfiability Paradigm

The propositional satisfiability (SAT) is an interesting example of constraint satisfaction problems. A propositional formula can be represented in conjunctive normal form (CNF): $\mathcal{F} = \bigwedge_m \bigvee_n l_{mn}$ in which each l_{mn} is a literal (propositional variable or its negation), and each disjunct $\bigvee_n l_{mn}$ is a clause [43]. Each literal can be either true or false. The problem is to find an assignment that satisfies $\mathcal{F}$.

The SAT problem is known to be the mother of the class of NP complete problems [12]. An efficient solution for SAT can be translated into efficient solutions for a wide range of hard computer science problems. SAT is the first computational problem to be shown as a NP-complete problem. Later it was used as the fundamental benchmark to prove other equivalent problems to be NP-Complete via mapping a polynomial encoding of these problems into SAT [13].

Therefore, both encoding problems as SAT and developing efficient techniques for SAT solving have become major research activities in the field in recent past[3, 44, 45, 86]. One of the key reasons that makes SAT more preferable over other NP-complete problems is its conceptual simplicity. Thus, the task of designing and implementing efficient algorithms for SAT solving is much more easier than it is for other NP-Complete problems. SAT also inherits a rich experience

of well studied deductive systems which have been developed by the automated theorem proving community for the last many years.

1.1.2 Local Search for SAT

Stochastic Local Search (SLS) is a successful technique for the solution of many large size problems where systematic search based approaches fail to succeed. A local search algorithm for SAT starts with a complete but inconsistent assignment (a combination of all variable instantiations that satisfies a subset of clauses), then tries to iteratively improve the consistency of the current solution by flipping the value of a selected variable. The selection of the variable to flip can be random or based on some domain dependent or independent guiding principle. The major limitation of local search techniques is that they are typically *incomplete* and hence neither guarantee to eventually find a solution for a given problem nor prove the problem to be unsatisfiable.

The effectiveness of local search techniques can be significantly improved by using additional heuristics and by using parameters to guide and control the process of searching for better cost solutions. These parameters are often domain dependent and need manual tuning. This tuning process requires significant time and effort before the actual search commences. Therefore, all other things being equal, parameter free algorithms are always preferred.

1.2 Research Problems

The importance of solving SAT problems efficiently is clear, especially large sized problems that cannot be solved by backtracking-based search techniques. This motivated us to look into further into developing new stochastic local search methods. The efficiency and robustness of local search methods in solving large combinatorial problems is well known in the optimisation field. In this thesis, we focus on clause weighting approaches to solve SAT problems. Our preliminary investigations have revealed that *developing an effective parameter-free clause weighting local search method* is an important research issue that has significant practical implications.

Existing local search methods generally have explicit parameters that are used to control the search fully or partially. As we have already discussed, tuning these parameters is hard and time consuming, and yet necessary to achieve acceptable performance. Although there are several local search techniques that can automatically adapt their parameter settings, their performance still falls well short of the best hand-tuned techniques. Therefore, it is an important research challenge

to develop a self-adjusting or parameter free algorithm that can match or even outperform some of the better parameter dependent SAT solvers.

There have been recent efforts in improving local search methods for SAT solving by using resolution as a preprocessor. This led to highly impressive results in SAT2005 competition for SAT solving. We therefore decided to investigate how combining resolution-based preprocessing and parameter-free local search can further improve the state-of-the-arts in the field.

Many real world problems are over-constrained. The extension of the best SAT solvers to handle over-constrained problems is therefore of great practical importance. To this end we address the issues of applying local search to over-constrained MAX-SAT problems and to over-constrained problems involving hard and soft constraints.

The current research addresses these problems via the development of a new approach to clause weighting aimed at reducing or eliminating the dependence of clause weighting on sensitive, problem dependent parameters. Existing weighting algorithms crucially depend on when and/or by how much clause weights are reduced during the search. In our approach, we eliminated such decisions by allocating a fixed amount of weight for a search and by only allowing a clause to increase weight by reducing the weight on another clause - thereby combining weight increments and decrements into a single redistribution step. This produces the central thesis of the current research: *that a clause weighting approach to SAT solving based on redistributing a fixed amount of weight can outperform existing parameter-based and parameter adapting clause weighting algorithms when the option of hand-tuning parameters is ruled out.* We argue that the ruling out of manual parameter tuning is necessary if local search algorithms are to be viable and competitive for use in practical commercial applications.

1.3 Research Contributions

The main contributions of this thesis are:

- The introduction of a new weight redistribution SAT solver called Divide and Distribute Fixed Weights (DDFW). The main idea behind DDFW came from observing the behaviour of existing clause weighting local search SAT solvers. In weighting strategies, an important factor that affects the performance of the solver is performing weight reduction after a certain number of weight increases (depending on the heuristic used). In our new approach, the idea was to transfer weight from satisfied to unsatisfied clauses instead of doing increasing

weight in one step and decreasing in another. We combined these two steps into a single action of redistribution to produce a new algorithm that no longer requires the tuning of a weight reduction parameter. This technique achieved promising results when compared to the state-of-the-art SAT local search solvers and was published in CP 2005.

- The evaluation of DDFW. This includes the identification of the best solver for a number of well-known problems from real-world domains.

- The enhancing of DDFW by adapting one of its default-valued parameters to produce DDFW$^+$. This was motivated by our observation that using different values for this parameter frequently effects the performance of the new method. Firstly, we performed extensive experiments to make sure that different values for the parameter are producing different levels of performance. Then a mechanism was successfully implemented to adapt the parameter. This produced one of the best performing techniques for solving SAT problems without the need for manual parameter tuning and was published in CP 2006.

- The enhancement and evaluation of DDFW and DDFW$^+$ by the addition of a resolution-based preprocessor. This includes the identification of the best combination of preprocessor and solver for a number of well-known problems from real-world domains.

- The extension of our new solver to successfully handle over-constrained unweighted MAX-SAT problems. We observed that, by recording the best cost found while searching for optimum (or near optimum) solutions, our new solver could be extended to competitively handle an important class of over-constrained (MAX-SAT) optimisation problems. A short version of this study has been accepted for publication Australian AI 2007.

- The development and evaluation of a new weight distribution approach to handling over-constrained problems with hard and soft constraints and the evaluation of this approach in comparison with another state-of-the-art SAT solver, *TWO-LEVEL*.

1.4 Thesis Outline

The remainder of this thesis is organised as followed: in the next chapter we define the main concepts of propositional satisfiability and present a general literature review on SAT solving, focusing on techniques used in the remainder of the thesis.

In Chapter 3, we investigate a new method that combines the addition and reduction of weights into a single step. As a result a new DDFW algorithm is proposed and empirically tested. This study strongly supports that the new method is among the best state-of-the-art methods in dealing with clause weights for SAT.

Next, in Chapter 4, we extend the work of Chapter 3 by introducing an adaptive mechanism that is used with DDFW to gain further improvements. We also study the effect of preprocessing on the performance of the best local search techniques in comparison with the new algorithm.

In Chapter 5, we present and evaluate our extension of DDFW to handle unweighted MAX-SAT problems and develop a new DDFW variant to handle problems with hard and soft constraints.

Finally, in Chapter 6, we summarise the general conclusions and research contributions of this thesis and discuss possible avenues of future work.

Chapter 2

Background

2.1 Introduction

This chapter covers the areas of propositional satisfiability that are pertinent to the thesis. First, we introduce and define the concept of a propositional satisfiability (SAT) problem. Then, we discuss the current state-of-the-art in systematic and local search solving techniques. Finally, we give an informal categorisation of the different local search approaches and discuss the use of parameters to optimise local search performance.

2.1.1 Propositional Satisfiability Problems

The propositional satisfiability (SAT) problem forms an important subset of the range of problems that can be represented as a CSP. However, although SAT problems can be solved using more general CSP approaches, in some domains the evidence suggests that translating CSPs into propositional satisfiability (SAT) and solving them with existing SAT solvers can be more efficient [35, 90]. These efficiencies can partly be explained by the nature of the SAT system, where boolean variables are used instead of more complex non-boolean CSP structures. Also, in SAT, constraints are represented uniformly in clausal forms that allow the creation of simple and efficient clause processing algorithms. This often means that designing and evaluating solvers to resolve SAT problems is considerably easier than doing so in the CSP sphere.

A propositional formula can be represented in conjunctive normal form (CNF) as follows: $\mathcal{F} = \bigwedge_m \bigvee_n l_{mn}$ where each l_{mn} is a literal (a propositional variable or its negation), and each disjunct $\bigvee_n l_{mn}$ is a clause. Each literal can be either true or false. The problem is to find an assignment that satisfies $\mathcal{F}$. Given that SAT is at the core of the class of NP complete prob-

lems, efficiently solving this problem has much wider implications for many computer science problems.

Although there are alternative SAT problem representations (such as disjunctive normal form), in practice, SAT problems have been represented primarily as CNF formulae. Hence, we only consider CNF SAT problems in the remainder of the thesis.

2.1.2 Mapping CSP-into-SAT

In general, real world problems can be formulated into a SAT CNF structure after an intermediate step that first transforms the problem into a binary CSP (consisting of only binary constraints) and then encoding the binary CSP into SAT. Examples of encoding CSPs into SAT are :

- conflict-based encodings [35, 90].
- logarithm-based encodings [35, 90].
- support-based encodings [24].

Such encodings greatly extend the applicability of SAT solving techniques making them competitive with existing native CSP approaches [90, 58].

2.1.3 The MAX-SAT Problem

While the SAT problem has many important applications, it is the case that many, if not most, real-world problems are *over-constrained.* If we express such problems using the SAT formalism, then the search task is no longer to find whether a satisfying assignment exists, but to find an assignment that maximises the number of true clauses (or equivalently minimises the number of false clauses). This problem is known as the unweighted MAX-SAT problem and has many applications in such areas as scheduling and the processing of Bayesian networks [81].

The following sections of the chapter discuss possible methods and approaches for solving CSP and SAT problems. We discuss CSP as well as SAT techniques because of the significant cross-over between the two areas, with SAT solvers regularly being used to solve CSPs and because both areas overlap considerably in terms of the kinds of techniques that are employed.

2.2 Systematic Techniques

Systematic techniques incrementally attempt to extend a partial solution (that specifies consistent values for a subset of variables) toward a complete solution, by repeatedly choosing a value for another variable consistent with the values in the current partial solution. Deciding which domain value to assign next is typically achieved using heuristic information gathered from the search process [59]. When no value consistent with the current partial solution can be found, the search reaches a *dead end* and a previous assignment is retracted (i.e. the search *backtracks*) [17].

Systematic methods can be divided into three main categories:

- Consistency-enforcing algorithms.
- Intelligent backtracking algorithms.
- Variable and value ordering heuristics.

2.2.1 Consistency-Enforcing Algorithms

Before applying a backtracking technique to solve a CSP problem, a consistency enforcing algorithm can be used to pre-check the consistency between the variables and their domain values [21]. Enforcing consistency helps the backtracking technique by eliminating domain values that cannot form part of a consistent solution. For example, in Figure 1.1, country B can never be coloured red and be consistent with country C, as C only has one red domain value available. The three main types of consistency-enforcing methods are node-consistency, arc consistency, and path consistency [18].

2.2.2 Backtracking Algorithms

The backtracking algorithm was first described more than a century ago and since then has been reintroduced several times [6], producing a range of sophisticated backtracking methods. Current intelligent backtracking techniques can be divided into two categories based on the way they tend to prevent or deal with dead-ends: look-back and look-ahead methods. The search for consistency in *look-back* algorithms is done by checking the already instantiated past variables with the domain values of the variable that is about to be instantiated, so that the current partial solution is consistently extended. Backjumping [47], conflict-directed backjumping [60], backmarking [23]

are all examples of look back schemes. Alternatively, look ahead algorithms search for consistency by checking each current potential instantiation against the possible values of the yet to be instantiated future variables. Forward checking [31] and maintaining arc consistency (MAC) [64] are examples of algorithms that follow the look ahead scheme.

2.2.3 Variable and Value Ordering Heuristics

Variable Ordering

Experimental studies have shown that the ordering in which variables are chosen for instantiation can have a substantial impact on the complexity of a backtracking search [84, 22, 27, 11, 6]. The ordering may be either *static*, in which the order of the variables is specified before the search begins, and is not changed thereafter, or *dynamic*, in which the choice of the next variable to be considered at any point depends on the current state of the search. Several variable ordering heuristics have been developed and analysed, the most common ones being based on the "first-fail" principle. In this approach, the variable with the fewest possible remaining domain values is selected for instantiation [18]. The reasoning is that assigning the variables with the least choice first will result in a smaller overall search tree, as inconsistencies will be detected earlier.

Value Ordering

Once the decision is made to instantiate a variable, it may have several domain values available. Again, the order in which these values are considered can have substantial impact on the time to find the first solution [74]. This is an advantage if it biases the search to explore branches that lead to a solution earlier than branches that result in dead ends. In the ideal case, if the CSP has a solution, and if a correct value is chosen for each variable, then a solution can be found without any backtracking.

2.2.4 Solving SAT Problems with Systematic Search

In Davis and Putnam's 1960 paper [16], a new DP algorithm was proposed to solve SAT CNF problems in a complete and systematic manner. DP works by exploiting a process known as *resolution*. Resolution [14, 62, 63] is a rule of inference originated from classical logic and was the first major step in turning classical deduction, *modus ponens*, into computation. This rule is defined as follows: from $\{l\} \cup \Sigma_1$ and $\{\bar{l}\} \cup \Sigma_2$, where Σ_1 and Σ_2 are sets of literals (or

Algorithm 1 ComputeResolvents($\mathcal{F}$)

```
1: for each clause c_1 of length ≤ 3 in F do
2:   for each literal l of c_1 do
3:     for each clause c_2 of length ≤ 3 in F s.t. l̄ ∈ c_2 do
4:       Compute resolvent r = (c_1 \ {l}) ∪ (c_2 \ {l̄})
5:       if r is empty then
6:         return unsatisfiable
7:       else
8:         if r is of length ≤ 3 then
9:           F := F ∪ {r}
10:        end if
11:      end if
12:    end for
13:  end for
14: end for
```

disjunctions of literals) and l is an atom, we can conclude $\Sigma_1 \cup \Sigma_2$ that is called the *resolvent* of the two clauses. Here, l is the *atom resolved upon* and the process is called *resolution*.

For all literals that appear positive in some clauses and negative in others, the resolution process can be applied as defined above. For example the clause $(a3 \vee a4 \vee \overline{a5})$ is the resolvent of the clauses $(\overline{a2} \vee a3 \vee a4)$ and $(a2 \vee a3 \vee \overline{a5})$. The resulting resolvent can then be used to generate further resolvents. For example consider the following formula in CNF:

1 $(a1 \vee a2 \vee a3)$

2 $(\overline{a1} \vee a2 \vee a3)$

3 $(a2 \vee a3)$ resolvent of clauses 1 and 2

4 $(a1 \vee \overline{a2} \vee a3)$

5 $(\overline{a1} \vee \overline{a2} \vee a3)$

6 $(\overline{a2} \vee a3)$ resolvent of clauses 4 and 5.

7 $\{\}$ resolvent of clauses 3 and 6.

Clause 7 is an empty clause that is generated by applying resolution on clauses 3 and 6, which means the formula has no solution and can not be satisfied.

As we shall discuss later, resolution can also be used as a preprocessing step to simplify a formula before trying to find a satisfying assignment. Such a procedure is illustrated in Algorithm in 1 and is used in the successful modern SAT solver Satz ([49], [4]).

The DP algorithm was further improved by Davis, Logemann and Loveland [15], and became popularly known as the DPLL method. A DPLL SAT solver employs an organised backtracking process to search the (exponentially-sized) space of variable assignments, looking for satisfying

assignments. This approach enhances the backtracking algorithm by the use of the following rules at each step:

- Unit Propagation:

  If a clause is a unit clause, i.e. it contains only a single unassigned literal, it can be satisfied by assigning the value that makes this literal true. Thus, no selection is needed. In practice, this frequently leads to deterministic cascades of units that eliminate large regions of the search space.

- Pure Literal Elimination:

  If a propositional variable occurs with only one polarity in the formula, it is called pure. Pure literals can always be assigned in a manner that makes all clauses containing them true. Thus, these clauses do not constrain the search and can be deleted.

Modern SAT solvers (developed in the last ten years) such as zChaff [52] augment the fundamental DPLL search algorithm with a range of additional heuristics including conflict driven clause learning, non-chronological backtracking, variable state independent decaying sum scoring of literals VSIDS [57], a two literal watching scheme for boolean constraint propagation, adaptive branching and random restarts. These extra features have helped modern complete SAT solvers to solve much larger problems than was possible for the original DPLL approach [37]. For example, zChaff (which implements the well known Chaff algorithm [57]) was the winner on the industrial and handmade instances in the SAT2002 competition [73] and was the best complete solver in the industrial category in the 2004 SAT competition [48].

The SAT Competitions

The SAT competition first started in the year 2002 as part of the annual international SAT conference.[1] This competition has played a significant role in the development of modern SAT solvers, both by giving a public forum where the various different solvers can be objectively compared and by defining sets of benchmark problems on which researchers have subsequently concentrated.

In addition, the rules of the SAT competition do not allow the hand-tuning of parameters in advance, and the actual problem instances are not made available until after the competition. This has led the SAT community to concentrate on developing techniques that either produce robust

[1] See http://www.satcompetition.org/

Algorithm 2 Hill-Climbing($\mathcal{F}$)

```
1:  A ← a randomly generated truth assignment of F
2:  while true do
3:    if A satisfies F then
4:      return A
5:    else
6:      select variable v which, if flipped, maximises the number of satisfied clauses
7:      if flipping v does not increase the current number of satisfied clauses then
8:        return no satisfying assignment found
9:      else
10:       update A with the flipped value of v
11:     end if
12:   end if
13: end while
```

performance with default parameter settings or can automatically adapt parameter settings during the search. This emphasis of the SAT competition against manual parameter tuning has acted as a strong motivator in the development of new more robust algorithms and has influenced the work in the current thesis. Comprehensive comparisons from the SAT competition have shown that complete search algorithms generally perform better than local search algorithms on the *structured* problems from the industrial and handmade domains, whereas local search approaches have the better performance on *random* problem instances. This difference in performance is largely related to the ability of the recent DPLL algorithms to identify and exploit the structure inherent in the non-random problem instances. However local search techniques still have the better performance on a number structured real-world problems taken from the other major SAT benchmark problem suites [77]: SATLIB (http://www.satlib.org) and DIMACS (http://dimacs.rutgers.edu/).

2.3 Local Search Techniques for SAT

Local search techniques have become some of the most successful and powerful algorithms for solving a range of CSP and SAT problems. Algorithms in this category share the same general template: a complete but inconsistent initial starting point is chosen by randomly instantiating all the variables in the search space, then the solution cost is iteratively improved by changing single instantiations until the search reaches a desired cost value or is timed out.

The simplest form of local search is known as greedy hill-climbing (see Algorithm 2). Here, the basic idea is to always move to the best available state in the local neighbourhood of possible moves. The weak point of a hill-climbing local search is that it will get stuck in the first local

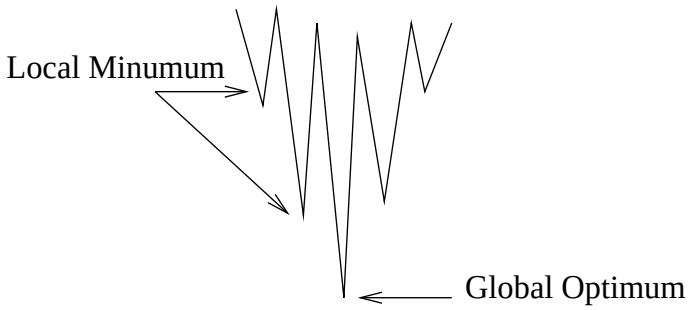


Figure 2.1: Local minima and global optima

minimum that it encounters (see Figure 2.1). In other words, if it cannot find a domain value that improves the cost of the current solution, the algorithm will stop searching and so cannot be guaranteed to find an optimal solution. Escaping local minima has become a major concern for researchers interested in applying local search for solving CSPs. This has resulted in the development of a number of new techniques that can be divided into the following four categories [76]:

1) **Restart Strategies:** algorithms that restart when encountering local minima or after a certain number of moves.

2) **Stochastic Escape Strategies:** algorithms that allow cost increasing moves (according to a fixed or dynamically adjusted probability).

3) **Memory Strategies:** algorithms that remember previous moves or solutions and so avoid moves that lead back to previously visited solutions.

4) **Weighting Strategies:** algorithms that change the cost topography by dynamically adjusting the cost of violated clauses.

2.3.1 Restart Strategies

The simplest approach to handling a local minimum is to restart the search (by randomly instantiating all variables) whenever a search runs out of cost improving moves. This is known as a *random restart* strategy [68]. Alternatively, in a *fixed iteration restart* strategy, the search does not wait until encountering a local minimum, instead it restarts after a specific number of moves (for example, see GSAT in Figure 3). The number of moves executed before restarting can also be specified from the search behaviour. For example, if we allow the search to take equal cost moves (i.e. moves that leave the overall cost of the solution unchanged) then we can use a strategy that will restart either when no equal or improving cost moves are available or after a fixed number of

Algorithm 3 GSAT($\mathcal{F}$, $MaxFlips$, $MaxTries$)

```
1: for i ← 1 to MaxTries do
2:    A ← a randomly generated truth assignment of F
3:    for j ← 1 to MaxFlips do
4:       if A satisfies F then
5:          return A
6:       else
7:          select variable v which, if flipped, maximises the number of satisfied clauses
8:          update A with the flipped value of v
9:       end if
10:   end for
11: end for
12: return no satisfying assignment found
```

equal cost moves have been taken.

GSAT

GSAT is a randomised hill-climbing algorithm for solving propositional satisfiability problems. It was first introduced by Selman and Kautz in 1992 [70] and is an early example of a successful restart strategy. GSAT starts with an arbitrarily generated truth assignment and proceeds by changing (flipping) the assignment of the variable that produces to the greatest number of satisfied clauses. This flipping is repeated until a solution is found that satisfies all clauses or until a pre-set number of flips ($MaxFlips$) is reached for each try. This procedure is then repeated until a maximum number of tries is reached (see $MaxTries$ in Algorithm 3).

2.3.2 Stochastic Escape Strategies

Stochastic escape techniques avoid being trapped in local minima by adding a random element to the move selection heuristic that allows cost increasing moves [76]. For example, simulated annealing accepts moves according to a random probability P as follows [46]:

1. At each step randomly select a neighbour state of the current state.
2. If the current state cost $>$ the neighbour state cost or $P >$ temperature T then move to the neighbour state.
3. Reduce the value of T according to a cooling schedule such that $T > 0$.

The intuition is that by being less greedy in the short term, simulated annealing can escape from local minima or plateaus more quickly, and hence benefits in the long run.

Selman, Kautz & Cohen in 1994 [69] noted that the high level of greediness in GSAT quickly attracts it towards local minima. In subsequent work, they extended GSAT with a random walk heuristic to produce GWSAT. In each search step, GWSAT randomly picks a variable from a false clause within a fixed *walk* probability wp and flips it regardless its effect on the overall cost. Otherwise, the GSAT heuristic is performed. By performing a random walk in this manner, the search can make unpredictable cost increasing moves that allow it to escape from local minima.

WalkSAT

Both GSAT and GWSAT consider all the variables that appear in the false clauses when trying to take a move. In 1994, Selman, Kautz & Cohen [69] first introduced a new WalkSAT approach to variable selection with the WSAT algorithm. In 1997, the WalkSAT family of algorithms was further elaborated by McAllester, Selman & Kautz [53]. In WalkSAT the variable selection is done in two stages: firstly, a false clause is randomly selected. Secondly, with probability wp a variable in the clause is randomly selected to be flipped, otherwise the variable in the clause which (if flipped) causes the maximal increase of the number of satisfied clauses is selected (see Algorithm 4). The WalkSAT algorithm differs from GWSAT in the first step where a single false clause is randomly chosen from the false clauses list, although the random walk heuristic is similar in both algorithms.

Algorithm 4 WalkSAT($\mathcal{F}, MaxFlips, MaxTries$)

```
1:  for i ← 1 to MaxTries do
2:      A ← a randomly generated truth assignment of F
3:      for j ← 1 to MaxFlips do
4:          if A satisfies F then
5:              return A
6:          else
7:              randomly select an unsatisfied clause c
8:              if within probability wp then
9:                  randomly select a variable v within clause c
10:             else
11:                 greedily select a variable v within clause c
12:             end if
13:             update A with the flipped value of v
14:         end if
15:     end for
16: end for
17: return no satisfying solution found
```

Novelty and R-Novelty

The Novelty algorithm [53] was introduced in 1997. It escapes from local minima (and sometimes avoids encountering local minima) by ordering variables within a randomly selected false clause according to the increase in total number of satisfied clauses when flipped. If the best variable in this ordering is not the most recently flipped, it is always flipped; otherwise, the second-best cost flip is selected with probability p. In the same work [53], McAllester *et al.* introduced another Novelty variant, R-Novelty. This algorithm follows the Novelty heuristic, but uses a more complex decision scheme that considers the size of the difference in score between the best and the second best scoring flips.

The Adaptive Mechanism (AdaptNovelty$^+$)

In 2002, a new WalkSAT method appeared known as AdaptNovelty$^+$ [36] which proposed a technique for dynamically adjusting the Novelty noise parameter p. AdaptNovelty$^+$ consists of two distinct heuristics:

1. The Novelty$^+$ heuristic developed in [34]. In this paper, Hoos demonstrated the incompleteness of Novelty by devising satisfiable instances for which Novelty could not find solutions. Hoos proposed a solution that relaxes the level of determinism of Novelty via the use of a WalkSAT random walking step. The resulting Novelty$^+$ algorithm will randomly flip a variable within a selected clause according to a probability wp, otherwise Novelty continues. The value of wp is generally fixed at 0.01.

2. The main feature of AdaptNovelty$^+$ is the mechanism for adapting the noise parameter. This idea emerged from earlier work in [53] and consists of increasing the amount of noise in situations of search stagnation (this is related to the work on Reactive Tabu Search [2] and Iterated Local Search [29]). In AdaptNovelty$^+$, the noise parameter p is increased by an amount $p := p + (1 - p)\cdot \phi$ when it is judged that the search has *stagnated*. Stagnation is recognised when there has been no improvement in the overall solution cost for the last $\theta \cdot m$ search steps. Conversely, noise is decreased by an amount $p := p - p \cdot \phi/2$ when an improvement in the overall cost is detected. While it may appear that parameter p is adapted at the expense of introduced three additional parameters, in practice these other parameters work robustly with the following fixed values: $\theta = 1/6$, $\phi = 0.2$ and $m =$ number of clauses in the problem instance. This robust performance was demonstrated when AdaptNovelty$^+$

won the random problem category of the SAT2004 competition [48]. In addition, a variant of AdaptNovelty$^+$ known as R+AdaptNovelty$^+$ (or ranov), that employs the preprocessing resolvent step discussed in Section 2.2.4, won random problem category of the SAT2004 competition [72].

G^{2}WSAT

Despite the fact that the deterministic variable selection of Novelty is the main difference and reason why it can solve many hard problems significantly better than other WalkSAT variants (such as WalkSAT/G, WalkSAT/Tabu, and WalkSAT/B and WalkSAT/SKC [53]), it can still get into infinite loops and fail to find a solution on certain problems [34, 50]. As discussed previously, this drawback was first addressed and solved by Novelty$^+$ [34].

Afterwards, a more diversified heuristic, analogous to the one used in HSAT [25] (discussed in 2.3.3) was proposed by Li & Huang [50]. The intuition behind this new heuristic was to further weaken the high level of determinism in Novelty, as follows: with a probability p of diversification, the least recently flipped variable is selected for flipping, otherwise the search continues as Novelty.

G^{2}WSAT also differs from Novelty in selecting flips from the set of all literals in the currently false clauses (as did GSAT) rather than limiting its choice to the literals in a randomly selected clause. The resulting gradient-based greedy WalkSAT (G^{2}WSAT) algorithm finished second in the random SAT category in the 2005 SAT competition [72].

2.3.3 Memory-based Strategies

Tabu Search

Tabu search (TS) [28] is one of the most successful local search meta heuristics used in the operations research community, outperforming many other approaches for hard problems like job shop scheduling [85], the quadratic assignment problem and graph colouring.

The strategy of a TS algorithm is to escape local minima by *remembering* the points in the solution space it has visited in the past and not allowing instantiations that lead to these previously visited points. In terms of space and time requirements, it is impractical to store and retrieve the complete instantiation of every point visited during a search. Instead a TS stores features of the most recently visited solutions that then define *tabu* partial solutions. Typically these features

are represented in a list of the most recently changed domain values - with the length of the list representing the length of the system's memory.

In spite of its efficiency on some problems, TS algorithm performance is significantly affected by the length of the tabu list, which generally has to be tuned to suit particular problem instances. As with AdaptNovelty$^+$ there has been work on dynamically adapting the tabu list length according to the degree of stagnation of the search [2]. In the SAT domain, a tabu search implementation was recently shown to outperform Novelty in [91]. TS has also been applied to solving MAX-SAT problems in [75].

HSAT and HWSAT

HSAT was proposed in [25]. It employs memory when breaking ties between equal cost flips by choosing the least recently flipped variable. If the choice is between variables that were never flipped in the current try, it uses an arbitrary ordering to choose between them. HSAT also follows GSAT in selecting flips from the set of all false clauses rather than randomly selecting a single false clause. Due to the high level of determinism in selecting variables, HSAT was observed to be strongly attracted to local minima [26]. The authors therefore extended HSAT to reduce the level of determinism by adding the random walk heuristic of GWSAT. The resulting algorithm, known as HWSAT [26], achieves a similar improvement in comparison with HSAT as GWSAT does in comparison to GSAT.

2.3.4 Weighting-based Strategies

The performance of a local search algorithm is affected dramatically by how greedy the method is. The more the greedy the method, the faster the descent in the search space. However, greediness will also guide a search into the nearest local minimum.

The main idea behind a weighting strategy is to allow unattractive moves whenever that is necessary. For example, if the current number of false clauses is one and if flipping any variable in that clause will cause two other clauses to become false then the search is in local minimum. In this situation, a weighting algorithm will increase the weight on the false clause until it becomes greater than combined weight of the clauses that will become false when that clause is satisfied.

In [56], the Breakout algorithm was one of the first algorithms to use weights to deal with local minima. The algorithm initially assigns a unit of weight to each problem clause and then proceeds to greedily select the flip that most reduces the summed weighted cost of the false clauses. Each

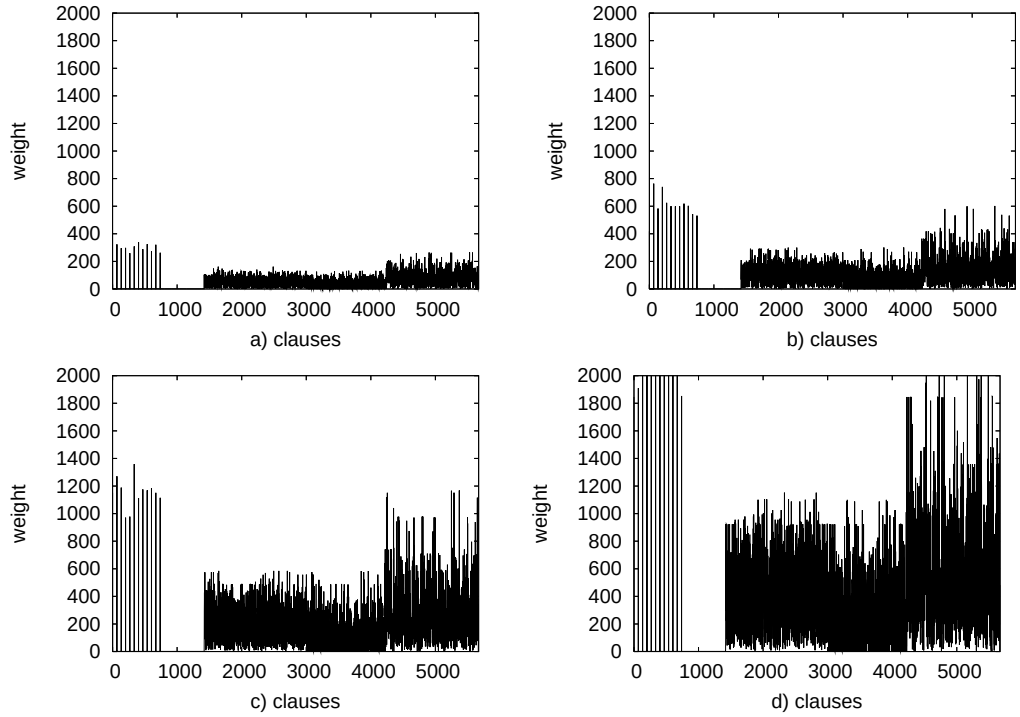


Figure 2.2: Weight distributions at various times, a) = 1 seconds, b) = 2 seconds, c) = 4 seconds and d) = 8 seconds on a single run of the Breakout algorithm applied to the ais12.cnf SATLIB all interval series problem (see www.satlib.org)

time a local minimum appears, the weights of the unsatisfied clauses are additively increased by 1. The search continues until a solution is found or it is timed out. Regardless of its simplicity and drawbacks, the idea of the Breakout algorithm has opened a large avenue for researchers which has resulted in producing several of the best known local search algorithms for solving SAT problems.

The main problem with Breakout is that the weight growth is unlimited. This may result in delaying the search from reaching a nearby solution because as larger and larger weights accumulate on each clause, each new weight increment has less and less effect. The figures in 2.2 illustrate Breakout's weight growth at various times during the search.

The clause weighting algorithm proposed in [68] is similar to the Breakout algorithm except that the increase of the weights takes place after each try rather than when a local minimum is encountered. A more general Guided Local Search (GLS) weighting algorithm was proposed in [88]. Here, the idea is to assign penalty values to unwanted features appearing in the solution space. Those values are modified according to a utility function μ whenever a local minimum is encountered forcing the search to visit another area in the search space.

Although the Breakout, clause weighting and the GLS algorithms increase weights differently, they all allow unlimited weight growth. The following subsections describe algorithms where the weights are also decremented at some stages during the search.

Discrete Lagrangian Method

The weight decrease mechanism for SAT first appeared in the Discrete Lagrangian Method (DLM) in [71]. The standard DLM algorithm uses a similar variable selection heuristic as in GSAT/Tabu. At each search step, a *non-tabu* variable is selected if it will reduce the sum of weights of the unsatisfied clauses. False clause weights are incremented additively after reaching a fixed number of flat (non-cost improving) moves (controlled by a θ_1 parameter). Another parameter is used to deterministically reduce the weights, after a fixed number of weight increases (controlled by a θ_1 parameter).

Later, a more complicated heuristic was implemented within the basic DLM to efficiently solve some specific DIMACS benchmark problem instances [2] such as the Hanoi towers and the parity learning problems [92, 93]. These enhanced DLM variants were shown to perform better than the standard DLM and WalkSAT/SKC algorithms. However, their optimal performance depends on the correct settings of a large number of parameters which are *non-robust* across the applied problem domains [67, 79].

GLSSAT

In 1999 GLS for SAT (GLSSAT)[54, 55] was evolved from the original GLS algorithm [87, 89]. GLSSAT starts by initialising all clause weights of the given problem to 0. Then, at each search step, the variable is selected that will reduce the overall weighted cost the most when flipped. Breaking ties is done in favour of the least recently flipped variable. If there is no cost reducing move, the least recently flipped *flat* variable is selected (flipping a flat variable leaves the weighted cost unchanged). After $smax$ (a fixed number of consecutive flat moves) if no solution was found, GLSSAT increases the weights of all current false clauses by 1.

In addition GLSSAT implements a weight reduction heuristic: if after increasing the weights, the maximal clause weights exceeds a preset $pmax$ limit, all the clause weights are reduced multiplicatively by a preset value $pdecay < 1$. This extra weight reduction mechanism played an important role in GLSSAT's improved performance in comparison with WalkSAT/SKC [55].

[2] http://dimacs.rutgers.edu/

However, more recent work suggests that GLSSAT is generally outperformed by other weighting algorithms such as ESG, SAPS and PAWS [67, 41, 79, 38].

Smoothed Descent and Flood

In 2000, Schuurmans & Southey developed a new and fully *multiplicative* weighting algorithm known as Smoothed Decent and Flood (SDF) [65]. SDF differs significantly from DLM in the following attributes and heuristics:

- In every local minimum, weights of the current false clauses are multiplied by a fixed factor.
- After each clause weight multiplication, the weights of all false clauses are normalised such that the difference in the sum of weights $\triangle w(x)$ remains fixed relative to a constant number δ.
- While the weights of false clauses are being maintained in the previous two points, weights of all satisfied clauses are reduced towards their average weight by a factor $(1 - \rho)$.
- Tie breaking among the same cost moves is done by taking into account the number of satisfied clauses that will result, otherwise the number of true literals is considered.

The resulting SDF algorithm could significantly outperform Novelty$^+$ on many benchmark problems in terms of search steps. However, SDF requires the use of floating point arithmetic for multiplicative weight updates which makes it slower relative to additive algorithms such as DLM. Also, SDF performs weight reduction at each local minimum, unlike DLM and GLSSAT where weight reduction is is only performed after a certain number of weight increases. The combined effects of these factors makes the time performance of SDF significantly slower than DLM [66].

Exponentiated Subgradient Optimisation (ESG)

ESG [67] follows a standard subgradient optimisation search with two main phases: the *weighted search* phase, and the *weight update* phase. The weighted search phase consists of a series of greedy variable flips where a variable is chosen from the set of all unsatisfied clauses that when flipped will lead to a maximal reduction in the total weight of unsatisfied clauses (breaking ties randomly). When there are no such variables, with probability η , a variable is randomly chosen and flipped from the set of all variables appearing in unsatisfied clauses. Otherwise, the weighted search phase is terminated and the weight update phase takes place. This phase consists of two

stages: first, the scaling stage, in which the weights of satisfied clauses are multiplied by α_{sat} and the weights of unsatisfied clauses are multiplied by α_{usat}. Secondly, the smoothing stage, in which the weight ω of each clause is pulled down towards the mean, calculated by the formula $\omega \leftarrow \omega \cdot \rho + (1 - \rho) \cdot \varpi$ where ϖ is the average of all clause weights after scaling, and ρ is a parameter fixed between zero and one. ESG generally shows competitive results in comparison with DLM (at least in terms of flip counts) when applied to solving SAT problems. However, although the work on ESG concluded that multiplicative weight updates produce better performance than additive updates this was later contradicted by [79].

Scaling and Probabilistic Smoothing (SAPS and RSAPS)

From the work on ESG, it was observed in [41] that ESG's weight update stage is much more expensive than the weighted search stage (in terms of time). Consequently, a new algorithm was developed called scaling and probabilistic smoothing (SAPS) [41] based on two key ideas: (1) The scaling operation is limited to the unsatisfied clauses only and (2) The expensive weight update procedure is only preformed occasionally with a certain probability P_{smooth} instead of after each weighted search stage. SAPS uses four parameters which control the algorithm behaviour: the multiplicative weight increase factor α, the multiplicative weight decrease factor ρ, the random walk probability, $\omega\rho$ and the smoothing probability P_{smooth}. In the same work on SAPS it was conjectured that the P_{smooth} parameter has the major impact on the overall performance of SAPS and that the other three parameters could be set to their default values (α = 1.3, ρ = 0.8 and $\omega\rho$ = 0.01). On this basis it was decided to develop an algorithm to adapt the P_{smooth} parameter based on the AdaptNovelty+ heuristic (see section 2.3.2) as follows: if the overall cost has not improved over the last $c \cdot vf$ flips (where c is the number of clauses and vf is the number of variable flips), the smoothing probability P_{smooth} is reduced. On the other hand, the moment that the number of unsatisfied clauses is reduced below the value at the last change of the smoothing probability, P_{smooth} is increased. This P_{smooth} adapting variant of SAPS is known as RSAPS or Reactive SAPS. SAPS and RSAPS outperformed ESG for all problems instances that appeared in [41]. Although the RSAPS results were encouraging, a carefully tuned SAPS still gives a better performance especially when applied to a larger problem instances, with regard to both the number of solutions found, and the speed in which a solution is reached.

Pure Additive Weighting Scheme (PAWS)

After the success of SAPS in reducing the time complexity imposed by the weighting update strategy in ESG and the use of a multiplicative weighting strategy, a study was conducted [79] in which a try was made to answer some questions arising from the work on SAPS. For instance, was the success of SAPS due to the multiplicative weighting? How does SAPS perform when applied to solve larger DIMACS benchmark problems? And was it because of the efficient implementation of SAPS that it could perform better than DLM and ESG? To answer these questions a new pure additive weighting scheme (PAWS) was developed [79]. PAWS was implemented in the SAPS source code with the multiplicative weighting strategy changed to additive. Then, both PAWS and SAPS were applied to solving a combination of medium and large problem sets chosen from the SATLIB and DIMACS benchmarks problem sets. In detail, PAWS works as follows (see Algorithm 5): firstly the weights of all clauses are initialised to one and all variables are randomly instantiated. Searching for a solution involves flipping the variable from the unsatisfied clauses that will maximally increase the number of satisfied clauses (breaking ties randomly). When a such variable does not exist, a zero cost flat move is selected with a fixed probability of 15%, otherwise a weight update stage is invoked. In this stage all the unsatisfied clause weights are incremented by 1. This process continues until the search reaches a pre-set Max_{inc} parameter number of weight increases, then the weights of all clauses are reduced by 1 (down to a minimum weight of 1). In summary, PAWS differs from SAPS in four aspects: (1) The use of additive rather than multiplicative weight adjustments (2) The use of flat moves rather than only allowing weighted cost reducing moves (3) Deterministic reduction of the weights rather than reducing the weights probabilistically (4) Multiple inclusion: with PAWS if a best cost variable flip appears in n false clauses then it will appear n times in the move list $\mathcal{L}$ (this increases the probability it will be picked when costs are tied).

Significantly, PAWS only uses a single Max_{inc} parameter which gives an advantage over the previous algorithms like SAPS, DLM, ESG, and SDF. However PAWS is sensitive to the setting of Max_{inc} which requires a significant number of trials to optimally tune the parameter. In general, PAWS is the simplest and yet one of the fastest state-of-art local search algorithms for solving large and hard SAT problems.

Algorithm 5 PAWS($\mathcal{F}$, $MaxTime$, Max_{inc})

```
1: A ← a randomly generated truth assignment of F
2: set the weight w_i for each clause c_i ∈ F to 1
3: while time < MaxTime do
4:    if A satisfies F then
5:       return A
6:    else
7:       select list L of literals causing the greatest reduction in weighted cost Δw when flipped
8:       if (Δw < 0) or (Δw = 0 and probability ≤ 15%) then
9:          update A by flipping a literal randomly selected from L
10:      else
11:         increase the weight of each false clause by 1
12:         if number of times clause weights increased % Max_inc = 0 then
13:            reduce the weight of each clause with weight > 1 by 1
14:         end if
15:      end if
16:   end if
17: end while
18: return no satisfying solution found
```

2.3.5 General Categorisation Criteria for Local Search

The best local search-based heuristics, such as SAPS, PAWS, DLM, and Novelty$^+$, are efficient and useful in many practical applications, but are also very sensitive to the settings of their own internal parameters. The common scheme used for testing the performance of a given algorithm is to firstly enter a parameter setting stage where multiple runs with different parameter values are tried to find a (near) optimal setting. Afterwards, the actual testing starts by applying the algorithm to solve each problem instance many times. Time spent on these runs is averaged for each instance and used to judge the quality of the algorithm.

In practice, the efficiency of an algorithm is judged on the time it takes to solve a problem. Time spent searching for optimum parameter values is generally not counted in this assessment which can therefore underestimate the true effort involved - especially for algorithms like SAPS where several interacting sensitive parameters are involved. There are several possible solutions to the above problem. Firstly, parameter tuning could be expressly disallowed, as in the SAT competitions, thereby stimulating the development of more efficient algorithms that do not require tuning of their parameters. Secondly, the process of parameter tuning could be standardised such that the effort of tuning the parameters of the given algorithm is added to the actual running time when applied to solving a problem. Thirdly, a general parameter prediction model could be used to predict the possible optimum (or near optimum) parameter values. Recent work [40] shows

that parameter values can be calculated based on the prediction of the run-time distribution using a combination of machine learning techniques. However this approach requires considerable preliminary effort to discover the correct prediction model.

In this thesis we are specifically interested in algorithms that do not depend on the setting of parameters to reach to their optimum (or at least acceptable) performance. This is motivated by the consideration that in most realistic situations it is not practical to have to solve a problem multiple times to discover the best parameter settings and then solve it again to find the best possible solution time.

In the following we split local search techniques into two main categories based on the parameter dependence for each algorithm:

- *Parameter-dependent solvers:* Solvers in this category require manual tuning of parameters to achieve competitive performance, i.e. to the extent that a poor choice of parameter values can turn a trivial problem into something intractable. A second feature of these algorithms is that optimal parameter settings vary widely across problem classes and sizes so that no generally effective default settings are available. Included in this category are the WalkSAT algorithms that are dependent on the setting of their noise parameters (including the Novelty variants), the clause weighting algorithms (SAPS, PAWS, DLM, SDF and ESG) that are dependent on parameters to control the process of weight updates and all tabu search algorithms that require the setting of the tabu list length.

- *Parameter-independent solvers:* This category is made up of two subcategories: A) Solvers with no parameters (or fixed value parameters) that give reasonably robust performance regardless of problem size and structure, and B) Solvers with adapted parameters, i.e. that are dynamically adjusted during the search. So far very few local search algorithms are independent of the use of parameters. AdaptNovelty$^+$ and RSAPS are two successful attempts to adapt the parameters used in the original Novelty$^+$ and SAPS algorithms. However, these adaptive algorithms are still unable to match the performance of tuned versions of their underlying algorithms. For instance, a tuned version of Novelty$^+$ can perform twice as well as its variant AdaptNovelty$^+$ [36]. Similarly, a tuned SAPS can perform significantly better than RSAPS.

Clearly, the ideal is to develop a local search technique that can dispense with parameter tuning while still achieving comparable levels of performance to the best tuned algorithms. Failing

that, it would be a major step forward to improve on the performance of the existing parameter independent solvers.

2.4 Summary

The chapter started by introducing the the areas of propositional satisfiability testing which will be relevant to the thesis. Included here was a subsection that discussed over-constrained SAT problems and their importance (as these problems are directly linked to the work of the coming chapters). Then we briefly described the most well-known systematic solving techniques, followed by a more in-depth analysis of local search approaches. As part of this analysis, we categorised local search solvers into memory-based strategies, restart-based strategies, stochastic escape-based strategies and weighting-based strategies. Finally, we discussed the parameter dependency issue using two divisions: parameter-dependent and parameter-independent solvers.

Chapter 3

Neighbourhood Clause Weight Redistribution in Local Search for SAT

3.1 Introduction

Since the development of the Breakout heuristic [56], clause weighting local search algorithms have been intensively investigated, and continually improved [9, 19]. However, the performance of these algorithms remained inferior to their non-weighting counterparts [53], until the development of weight smoothing heuristics [93, 66, 41, 79]), which currently represent the state-of-the-art for paramter-tuned SLS methods on SAT problems.

The work of this chapter has evolved out of work on recently developed clause weighting algorithms: Scaling and Probabilistic Smoothing (SAPS) [41] and the Pure Additive Weighting Scheme (PAWS)[79]. SAPS, in turn, evolved out of SDF [66] and ESG [67], retaining the multiplicative weighting framework of the earlier methods, but gaining considerable efficiencies from smoothing weights periodically (rather than at each local minimum) as discussed in the previous chapter. PAWS uses a similar approach to SAPS, but updates weights additively rather than multiplicatively. Additive weighting was previously used in DLM [93], but PAWS dispenses with DLM's plateau searching heuristic and parameters, instead using a random flat move heuristic described in Chapter 2.

In a recent study [79], PAWS was shown to outperform SAPS over a range of structured and random SAT problems, particularly on larger instances. The superiority of PAWS was narrowed down to the use of additive rather than multiplicative weight updates, and indicated that the weight distinctions made by multiplicative schemes are generally less effective. However, SAPS was still better on some problems, and while PAWS was significantly better overall, the performance of the two algorithms could be considered as similar, with much of PAWS' advantage coming from

efficiencies gained from using integer updates. Hence, we can consider SAPS and PAWS as two versions of the same underlying weighting strategy: increase weights on false clauses in a local minimum, then periodically reduce weights according to a problem specific parameter setting. A key weakness of PAWS and SAPS is that their performance depends on problem specific parameter tuning. This issue was addressed in RSAPS [41] by applying the same adaptive noise mechanism that was introduced in AdaptNovelty+ [36].

The question addressed in the current chapter is whether there are alternative weighting schemes that can produce further performance gains in the SAT domain. In particular, we are interested in *weight redistribution* schemes, that move around a fixed quantity of weight between clauses. Such an approach offers the advantage of not explicitly reducing weights, thereby avoiding considerable computational overhead, and also avoiding the need for a problem specific weight reduction parameter. Secondly, we are interested in exploiting structural information contained in the weight distributions between neighbouring clauses. As adding weight to a clause can only immediately affect those clauses with which it shares a variable, it appears promising to connect weighting decisions with the relative level of weight on neighbouring clauses. We combine both weight redistribution and consideration of neighbourhood relationships in the Divide and Distribute Fixed Weights (DDFW) algorithm, which implements weight redistribution between neighbouring clauses.

In the remainder of the chapter we provide further background on SAPS and PAWS and the evolution of clause weighting algorithms. We then introduce DDFW in more detail, and provide an empirical comparison between the various clause weighting approaches. We also consider RSAPS and AdaptNovelty+ in our empirical study. From this we identify a significant class of problems for which DDFW has considerably better performance, and conclude that weight redistribution and neighbourhood strategies could be the key to a next generation of structure exploiting and problem specific parameter-free local search SAT solvers.

3.2 Clause Weighting Techniques for SAT

Clause weighting local search algorithms for SAT follow the basic procedure of repeatedly flipping single literals that produce the greatest reduction in the sum of false clause weights. Typically, all literals are randomly initialised, and all clauses are given a fixed initial weight. The search then continues until no further cost reduction is possible, at which point the weight on all unsatisfied clauses is increased, and the search is resumed, punctuated with periodic weight

reductions.

Existing clause weighting algorithms differ primarily in the schemes used to control the clause weights, and in the definition of the points where weight should be adjusted. Multiplicative methods, such as SAPS, generally adjust weights when no further improving moves are available in the local neighbourhood. This can be when all possible flips lead to a worse cost, or when no flip will improve cost, but some flips will lead to equal cost solutions. As multiplicative real-valued weights have much finer granularity, the presence of equal cost flips is much more unlikely than for an additive approach (such as DLM or PAWS), where weight is adjusted in integer units. This means that additive approaches frequently have the choice between adjusting weight when no improving move is available, or taking an equal cost (flat) move.

Despite these differences, the three most well-known clause weighting algorithms (DLM [93], SAPS [41] and PAWS [79]) all share a similar structure in the way that weights are updated:[1] Firstly, a point is reached where no further improvement in cost appears likely. The precise definition of this point depends on the algorithm, with DLM expending the greatest effort in searching plateau areas of equal cost moves, and SAPS expending the least by only accepting cost improving moves. Then all three methods converge on increasing weights on the currently false clauses (DLM and PAWS by adding one to each clause and SAPS by multiplying the clause weight by a problem specific parameter $\alpha > 1$). Each method continues this cycle of searching and increasing weight, until, after a certain number of weight increases, clauses weights are reduced (DLM and PAWS by subtracting one from all clauses with weight > 1 and SAPS by multiplying all clause weights by a problem specific parameter $\rho < 1$). SAPS is further distinguished by reducing weights probabilistically (according to a parameter P_{smooth}), whereas DLM and PAWS reduce weights after a fixed number of increases (again controlled by parameter).[2] PAWS is mainly distinguished from DLM in being less likely to take equal cost or flat moves. DLM will take up to θ_1 consecutive flat moves, unless all available flat moves have already been used in the last θ_2 moves. PAWS does away with these parameters, taking flat moves with a fixed probability of 15%, otherwise it will increase weight. Fig 3.1 represent a general categorisation for the clause weight algorithms applied to solving SAT domains.

As SAPS has already been shown to outperform DLM on a range of SAT problems [41], and

[1] Additionally, in Chapter 2 we discussed a fourth clause weighting algorithm, GLSSAT [54], uses a similar weight update scheme, additively increasing weights on the least weighted unsatisfied clauses and multiplicatively reducing weights whenever the weight on any one clause exceeds a predefined threshold.

[2] In more recent work [82], it was concluded that SAPS works equally as well using deterministic smoothing.

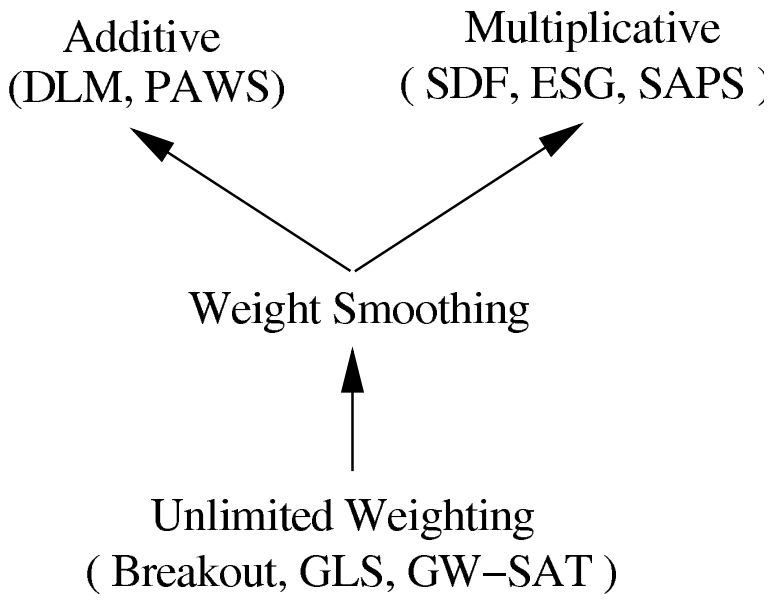


Figure 3.1: A general categorisation of clause weight algorithms for SAT

PAWS has been shown to have better time performance on the same problem set [79], we decided to include PAWS in our empirical study and to base the development of our new approach on the PAWS source code. However, it should be noted that specialised versions of DLM (using a distance penalty) still have state-of-the-art performance on some of the largest and most difficult SAT instances [93]. Also, as SAPS retained superior flip performance on several of the benchmarks in our empirical study, we have included SAPS results for comparison purposes.

However, determining parameter settings manually is a difficult task and can be very time-consuming. As discussed in Chapter 2, RSAPS is a variant of SAPS that has self-tuning mechanism (automatic parameter setting). Its performance has also been shown to be competitive with ESG and SAPS, at least on some problems [41]. Another self-tuning algorithm, Novelty$^+$ (introduced in Chapter 2), belongs to the WalkSAT family of algorithms. These approaches are based on an iterative search process that in each step randomly selects a currently unsatisfied clause, and chooses a variable in this clause to flip. The different strategies of variable selection give rise to different WalkSAT algorithms. The performance and behaviors of WalkSAT algorithms depend on the setting of a noise parameter that controls the search process. AdaptNovelty+ is known to be the best algorithm in this family that relies on adaptive noise mechanism. Indeed, this algorithm was turned out to be best performer at 2004 SAT competition.

3.3 Divide and Distribute Fixed Weights

The Divide and Distribute Fixed Weights (DDFW) algorithm introduces two new ideas into the area of clause weighting algorithms for SAT. Firstly, it evenly distributes a fixed quantity of weight across all clauses at the start of the search, and then escapes local minima by *transferring weight from satisfied to unsatisfied clauses*. The existing state-of-the-art clause weighting algorithms have all divided the weighting process into two distinct steps: i) increasing weights on false clauses in local minima and ii) decreasing or normalising weights on all clauses after a series of increases, so that weight growth does not spiral out of control. DDFW combines this process into a single step of weight transfer, thereby dispensing with the need to decide when to reduce or normalise weight. In this respect, DDFW is similar to the predecessors of SAPS (SDF [66] and ESG [67]), which both adjust *and* normalise the weight distribution in each local minimum. Because these methods adjust weight across all clauses, they are considerably less efficient than SAPS, which normalises weight after visiting a series of local minima.[3] DDFW escapes the inefficiencies of SDF and ESG by only transferring weights between pairs of clauses, rather than normalising weight on all clauses. This transfer involves selecting a single satisfied clause for each currently unsatisfied clause in a local minimum, reducing the weight on the satisfied clause by an integer amount and adding that amount to the weight on the unsatisfied clause. Hence DDFW retains the additive (integer) weighting approach of DLM and PAWS, and combines this with an efficient method of weight redistribution, i.e. one that keeps all weight reasonably normalised without repeatedly adjusting weights on all clauses.

DDFW's weight transfer approach also bears similarities to the operations research subgradient optimisation techniques discussed in [67]. In these approaches, Lagrangian multipliers, analogous to the clause weights used in SAT, are associated with problem constraints, and are adjusted in local minima so that multipliers on unsatisfied constraints are increased and multipliers on satisfied constraints are reduced. This *symmetrical* treatment of satisfied and unsatisfied constraints is mirrored in DDFW, but not in the other SAT clause weighting approaches (which increase weights and then adjust). However, DDFW differs from subgradient optimisation in that weight is only transferred between pairs of clauses and not across the board, meaning less computation is required.

[3] Increasing weight on *false* clauses in a local minimum is efficient because only a small proportion of the total clauses will be false at any one time.

3.3.1 Exploiting Neighbourhood Structure

The second and more original idea developed in DDFW, is the exploitation of neighbourhood relationships between clauses when deciding which pairs of clauses will exchange weight.

We term clause c_i to be a neighbour of clause c_j, if there exists at least one literal $l_{im} \in c_i$ and a second literal $l_{jn} \in c_j$ such that $l_{im} = l_{jn}$ where l_{im} is the m^{th} literal of c_i and l_{jn} is the n^{th} literal of c_j. Furthermore, we term c_i to be a *same sign* neighbour of c_j if the sign of any $l_{im} \in c_i$ is equal to the sign of any $l_{jn} \in c_j$ where $l_{im} = l_{jn}$. From this it follows that each literal $l_{im} \in c_i$ will have a set of same sign neighbouring clauses $C_{l_{im}}$. Now, if c_i is false, this implies all literals $l_{im} \in c_i$ evaluate to false. Hence flipping any l_{im} will cause it to become true in c_i, and also to become true in all the same sign neighbouring clauses of l_{im}, i.e. $C_{l_{im}}$. Therefore, flipping l_{im} will *help* all the clauses in $C_{l_{im}}$, i.e. it will increase the number of true literals, thereby increasing the overall level of satisfaction for those clauses. Conversely, l_{im} has a corresponding set of opposite sign clauses that would be *damaged* when l_{im} is flipped.

The reasoning behind the DDFW neighbourhood weighting heuristic proceeds as follows: if a clause c_i is false in a local minimum, it needs extra weight in order to encourage the search to satisfy it. If we are to pick a neighbouring clause c_j that will donate weight to c_i, we should pick the clause that is most able to pay. Hence, the clause should firstly already be satisfied. Secondly, it should be a same sign neighbour of c_i, as when c_i is eventually satisfied by flipping l_{im}, this will also raise the level of satisfaction of l_{im}'s same sign neighbours. However, taking weight from c_j only increases the chance that c_j will be helped when c_i is satisfied, i.e. not all literals in c_i are necessarily shared as same sign literals in c_j, and a non-shared literal may be subsequently flipped to satisfy c_i. The third criteria is that the donating clause should also have the largest store of weight within the set of satisfied same sign neighbours of c_i

The intuition behind the DDFW heuristic is that clauses that share same sign literals should form alliances, because a flip that benefits one of these clauses will always benefit some other member(s) of the group. Hence, clauses that are connected in this way will form groups that tend towards keeping each other satisfied. However, these groups are not closed, as each clause will have clauses within its own group that are connected by other literals to other groups. Weight is therefore able to move between groups as necessary, rather than being uniformly smoothed (as in existing methods). The idea is that an unsatisfied clause gains weight from another clause within the same group in such a way that the overall level of satisfaction of each group remains the same or is improved. The principle is one of causing the least damage to the existing solution by taking

weight from a clause that can most easily afford to have its importance reduced. This is in contrast to other existing weighting techniques that treat all clauses the same by reducing weight across the board.

Fig 3.2 illustrates a general framework for DDFW. Initially (after the complete random assignment of the problem variables), clauses are divided into satisfied and unsatisfied groups. The satisfaction status of a clause could be changed during the search after some step/s (flips), i.e. some satisfied clauses may become unsatisfied and vice versa. Then, when the search is judged to have reached a local minimum, the weights are transferred among the clauses. In every local minimum, the weights are repeatedly transferred until the search finds a weighted cost improving move and escapes.

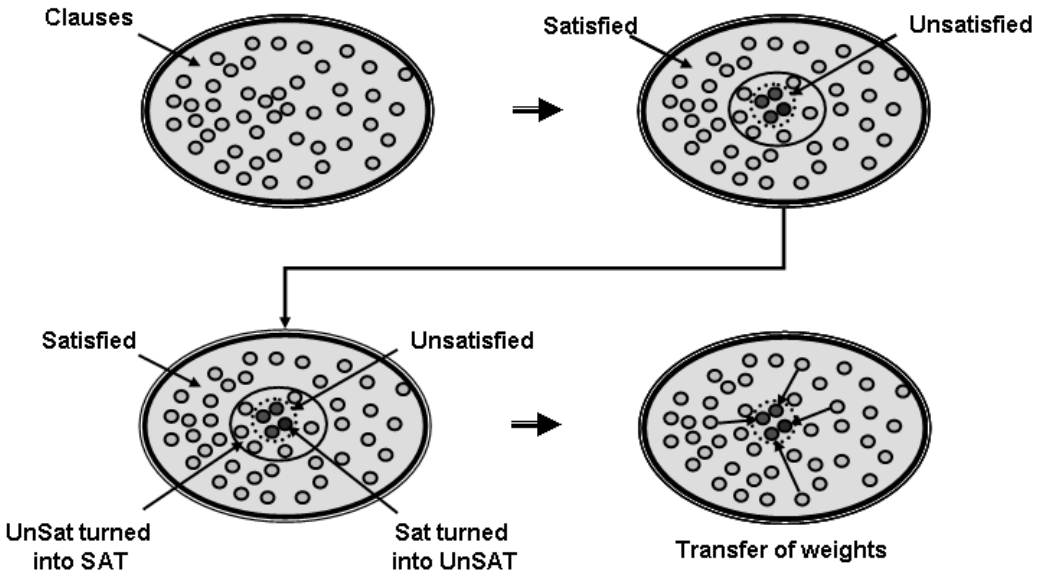


Figure 3.2: Illustration of the DDFW general framework

To the best of our knowledge the only other SAT local search techniques to exploit neighbourhood relationships were [9] and [61]. These approaches used opposite sign relationships to generate new clauses by resolution, and so are not directly related to the work on DDFW.

3.3.2 Implementation Details

The pseudocode for DDFW is shown in Algorithm 6. DDFW was implemented directly into the PAWS source code[4] that in turn was implemented into the original version of SAPS. Hence, all three techniques share the same efficient flip update architecture and data structures, allowing us

[4] http://www.int.gu.edu.au/ johnt/paws.tar

Algorithm 6 DDFW($\mathcal{F}$, $MaxTime$, W_{init})

```
1:  A ← a randomly generated truth assignment of F
2:  set the weight w_i for each clause c_i ∈ F to W_init;
3:  while time < MaxTime do
4:    if A satisfies F then
5:      return A
6:    else
7:      select list L of literals causing the greatest reduction in weighted cost Δw when flipped
8:      if (Δw < 0) or (Δw = 0 and probability ≤ 15%) then
9:        update A by flipping a literal randomly selected from L
10:     else
11:       for each false clause c_f do
12:         select a satisfied same sign neighbouring clause c_k with maximum weight w_k;
13:         if w_k < W_init then
14:           randomly select a clause c_k with weight w_k ≥ W_init;
15:         end if
16:         if w_k > W_init then
17:           transfer a weight of 2 from c_k to c_f;
18:         else
19:           transfer a weight of 1 from c_k to c_f;
20:         end if
21:       end for
22:     end if
23:   end if
24: end while
25: return no satisfying solution found
```

to make undistorted runtime comparisons.

DDFW uses the same flip selection approach as PAWS. This is for two main reasons: firstly, PAWS and DDFW both use additive (integer) weighting schemes and hence face a cost surface that is more likely to contain equal cost flips. This is in contrast to multiplicative methods like SAPS, where real valued weights create finer weight distinctions and make it unlikely that any two flips will evaluate to the same cost. All three methods will automatically accept flips that reduce the weighted false clause cost. However, in situations where the best available flips leave the cost unchanged, the choice is either to keep on searching in the expectation that another cost reducing flip will be discovered later, or to immediately increase weight on false clauses. PAWS already successfully navigates such situations by taking equal cost flips with a probability of 15%, otherwise false clause weights are increased by one. DDFW follows the same procedure to decide *when* to increase false clause weights, but differs in the way that the weights are adjusted. The second reason that DDFW follows PAWS' flip selection heuristic is so that differences between

Algorithm 7 PAWS($\mathcal{F}$, $MaxTime$, Max_{inc})

```
1:  A ← a randomly generated truth assignment of F
2:  set the weight w_i for each clause c_i ∈ F to 1
3:  while time < MaxTime do
4:    if A satisfies F then
5:      return A
6:    else
7:      select list L of literals causing the greatest reduction in weighted cost Δw when flipped
8:      if (Δw < 0) or (Δw = 0 and probability ≤ 15%) then
9:        update A by flipping a literal randomly selected from L
10:     else
11:       increase the weight of each false clause by 1
12:       if number of times clause weights increased % Max_inc = 0 then
13:         reduce the weight of each clause with weight > 1 by 1
14:       end if
15:     end if
16:   end if
17: end while
18: return no satisfying solution found
```

the two approaches can be entirely explained in terms of the weight update scheme.

For comparison purposes, we reproduce the PAWS pseudocode from Chapter 2 in Algorithm 7. Here Max_{inc} determines the number of times PAWS increases weight before performing a weight reduction. Max_{inc} is a problem dependent parameter that must be carefully tuned to obtain optimum performance. As discussed in [79], one of the significant contributions of PAWS was to reduce the number of necessarily tunable parameters down to one (SAPS and DLM both have three parameters that have proved sensitive in the SAT domain). An important challenge in the SAT local search community is to develop a parameter free heuristic that can equal the state-of-the-art performance of SAPS, DLM or PAWS (for instance see the work on RSAPS [41]).

Although DDFW does away with the PAWS Max_{inc} parameter, its performance is influenced by the amount of weight that is initially given to each clause. This amount is set by the value of W_{init} (see Algorithm 6). In effect, W_{init} controls the proportion of weight redistributed between clauses, i.e. a larger W_{init} causes a proportionally smaller redistribution of weight, and vice versa, as the amount of weight redistribution remains fixed. While W_{init} does affect performance, the range of settings on the problem set used in the empirical study was only from $2 \ldots 20$ in steps of one, in comparison to Max_{inc}'s range of $3 \ldots 74$. In addition, DDFW's performance proved less sensitive to changes in W_{init} (in comparison to PAWS or SAPS) and performed fairly robustly with a setting of 8.

There are two further details in the implementation of DDFW that proved important. Firstly, the weight on a clause is not allowed to fall below $W_{init} - 1$ (see Figure 3.4). Hence, if the situation arises where no neighbouring same sign clause has sufficient weight to give to a false clause, then a non-neighbouring clause with sufficient weight is chosen randomly. Secondly, if the donating clause has a weight greater than W_{init} then it donates a weight of two, otherwise it donates a weight of one (see Figure 3.3).

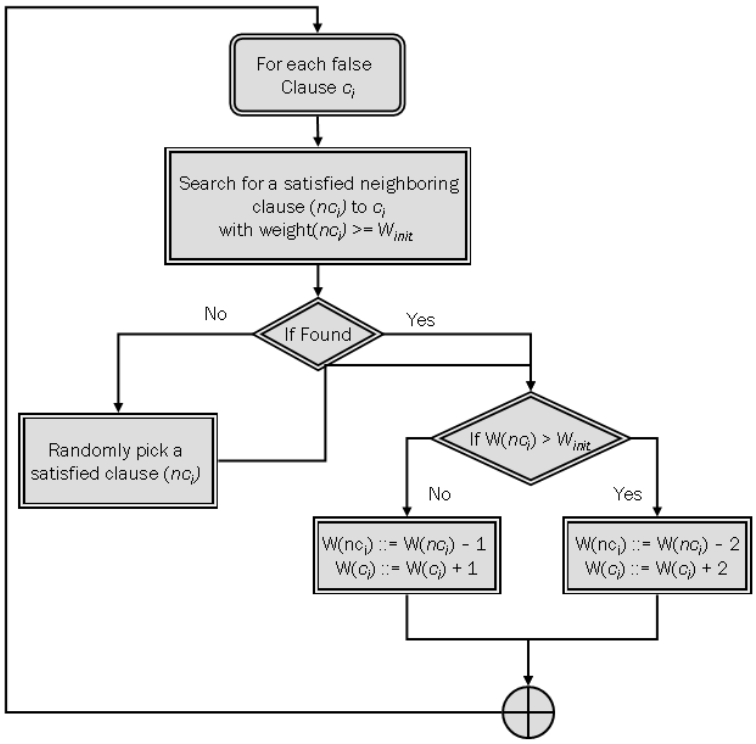


Figure 3.3: DDFW flow chart

3.4 Results and Analysis

To illustrate the performance characteristics of DDFW we decided to perform two empirical studies. Firstly, while developing DDFW, we noticed that the initial cost parameter W_{init} remained fairly robust across many problem instances. This inspired us to compare the default performance of DDFW ($W_{init} = 8$) with the RSAPS and AdaptNovelty$^+$ parameter adapting algorithms. This study is designed to show the utility of DDFW when parameter tuning is not practical. Secondly we compared DDFW with a tuned W_{init} against tuned versions of SAPS and PAWS, as these two algorithms represent the state-of-the-art in clause weighting local search for SAT. The idea here was both to see how DDFW stands in relation to the other clause weighting techniques and to

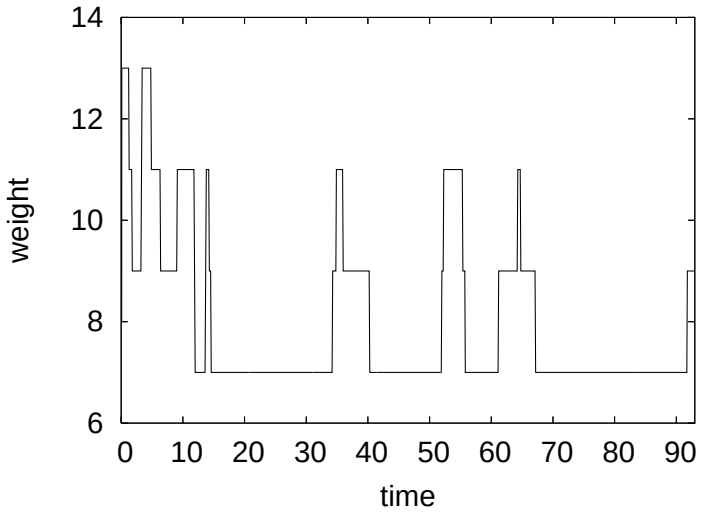


Figure 3.4: Weight distribution for a single clause during the first 93 seconds of DDFW execution when solving a parity 16 problem from the SATLIB library.

evaluate the effect of W_{init} on DDFW's performance.

3.4.1 Comparison with RSAPS and AdaptNovelty$^+$

Table 3.1 shows the results for DDFW ($W_{init} = 8$), RSAPS and AdaptNovelty$^+$ on the original problem set used to evaluate SAPS [41]. This set is made up of SAT-encoded blocks world (bw) and logistics planning instances, SAT-encoded flat graph colouring problems (flat), critically constrained uniform random-3-SAT instances (uf), and SAT-encoded all-interval-series problems (ais). These instances have already been extensively studied, with comparative results published for WSAT, Novelty+, DLM, ESG, SAPS and PAWS [65, 67, 41, 79].

To test performance on larger problems, Table 3.2 shows results for the extended problem set used in the original PAWS study [79], made up of the SATLIB bw-large.d blocks world problem, the median and hardest DIMACS 16-bit parity learning problems (par16), a selection of larger uf problems and a range of critically constrained multivalued SAT-encoded random binary CSPs (where v = the number of CSP variables, d = the domain size, and c = the constraint density). In both tables, a -med or -hard suffix indicates that the median or hardest problem was selected from a larger problem set.

All experiments in this section were performed on a Dell machine with a 3.1GHz CPU and 1GB memory running Linux. Problems with a mean flip count of less than one million were tested on 1,000 runs, otherwise tests were over 100 runs, with all runs having a 20 million flip

cut-off for RSAPS and DDFW, except 50v15d40c, which used 50 million (AdaptNovelty$^+$ was allowed twice the flip cut-offs of the other algorithms as it performs flips approximately twice as fast). Means and medians are reported for all runs, including those that reached their flip limit.

		Success	Time(secs)		Flips	
Problem	Method	% runs	median	mean	median	mean
bw_large.a	RSAPS	100	0.01	0.01	2,660	3,543
	AdaptNovelty$^+$	100	0.01	0.01	9,914	12,690
	DDFW	100	**0.00**	**0.00**	1,842	2,323
bw_large.b	RSAPS	100	0.06	0.09	28,830	39,439
	AdaptNovelty$^+$	100	0.13	0.19	153,858	221,015
	DDFW	100	**0.04**	**0.05**	11,582	16,862
bw_large.c	RSAPS	84	19.85	24.86	3,355,668	4,203,079
	AdaptNovelty$^+$	61	7.43	7.50	6,080,786	6,139,377
	DDFW	100	**0.49**	**0.74**	78,417	118,343
logistics.c	**RSAPS**	100	**0.01**	**0.01**	7,832	9,541
	AdaptNovelty$^+$	100	0.08	0.10	104,946	135,048
	DDFW	100	0.67	0.70	289,353	336,127
flat100-med	RSAPS	100	0.00	0.01	10,181	14,523
	AdaptNovelty$^+$	100	0.00	0.00	9,791	14,090
	DDFW	100	**0.00**	**0.00**	4,954	6,901
flat100-hard	**RSAPS**	100	**0.02**	**0.03**	25,685	35,443
	AdaptNovelty$^+$	100	0.03	0.05	89,579	130,250
	DDFW	100	0.03	0.04	26,900	36,440
flat200-med	RSAPS	100	0.13	0.18	138,254	192,376
	AdaptNovelty$^+$	100	**0.08**	**0.12**	201,187	289,196
	DDFW	100	0.11	0.14	84,991	114,718
flat200-hard	RSAPS	78	5.04	5.17	5,391,192	5,523,884
	AdaptNovelty$^+$	37	4.32	3.41	10,000,000	7,893,681
	DDFW	100	**0.99**	**1.51**	766,147	1,169,003
uf100-hard	RSAPS	100	0.00	0.00	3,642	5,592
	AdaptNovelty$^+$	100	**0.00**	**0.00**	3,954	5,503
	DDFW	100	0.00	0.00	10,182	13,503
uf250-med	RSAPS	100	0.01	0.02	11,847	16,189
	AdaptNovelty$^+$	100	**0.00**	**0.00**	11,768	15,948
	DDFW	100	0.02	0.02	14,576	19,048
uf250-hard	**RSAPS**	100	**0.18**	**0.27**	179,287	260,527
	AdaptNovelty$^+$	97	1.09	1.55	1,842,282	2,607,267
	DDFW	100	0.65	0.90	466,887	689,116
uf400-med	RSAPS	100	0.13	0.20	114,512	166,586
	AdaptNovelty$^+$	100	0.11	0.15	180,304	246,879
	DDFW	100	**0.06**	**0.07**	37,155	49,704
uf400-hard	RSAPS	100	4.07	4.96	3,222,372	3,930,879
	AdaptNovelty$^+$	45	12.30	9.03	20,000,000	14,628,570
	DDFW	100	**0.57**	**0.83**	376,448	555,404
ais10	**RSAPS**	100	**0.02**	**0.04**	18,600	28,979
	AdaptNovelty$^+$	99	2.00	2.62	1,447,718	1,895,154
	DDFW	100	1.10	1.34	313,841	368,749

Table 3.1: Default DDFW results for the original SAPS problem set (see [41]). The best performing technique on each problem is indicated in bold.

Table 3.1 shows DDFW has the better local search time performance on all the bw_large prob-

		Success	Time(secs)		Flips	
Problem	Method	% runs	median	mean	median	mean
bw_large.d	RSAPS	4	109.00	105.60	10,000,000	9,685,554
	AdaptNovelty$^+$	18	19.96	18.14	10,000,000	9,089,274
	DDFW	**100**	1.31	1.52	123,876	141,145
f800-med	RSAPS	16	15.2	14.0	10,000,000	9,210,799
	AdaptNovelty$^+$	100	**0.25**	**0.40**	400,533	637,733
	DDFW	100	0.97	1.20	513,983	624,569
f800-hard	RSAPS	8	15.50	14.80	10,000,000	9,547,070
	AdaptNovelty$^+$	72	3.67	3.70	5,779,569	5,834,230
	DDFW	**100**	2.81	3.98	1,371,652	1,940,812
f1600-med	RSAPS	0	timed out	timed out	timed out	timed out
	AdaptNovelty$^+$	95	1.88	2.60	2,647,123	3,667,484
	DDFW	**100**	2.30	3.44	809,927	1,216,190
f1600-hard	RSAPS	0	timed out	timed out	timed out	timed out
	AdaptNovelty$^+$	71	14.20	16.00	19,570,985	22,092,279
	DDFW	**100**	13.60	17.30	4,708,978	6,169,660
par16-med	**RSAPS**	**84**	52.80	59.80	81,295,331	92,022,550
	AdaptNovelty$^+$	49	53.30	37.30	80,000,000	55,992,729
	DDFW	62	30.00	36.20	18,283,060	31,675,190
par16-hard	**RSAPS**	**71**	19.90	20.60	22,812,746	23,646,125
	AdaptNovelty$^+$	21	26.30	23.30	40,000,000	35,412,958
	DDFW	48	45.70	33.30	20,000,000	17,954,673
30v10d80c	RSAPS	100	0.15	0.21	75,788	108,651
	AdaptNovelty$^+$	100	**0.01**	**0.01**	9,971	13,625
	DDFW	100	1.52	2.90	259,788	461,527
30v10d40c	RSAPS	100	0.12	0.18	70,212	101,054
	AdaptNovelty$^+$	100	**0.02**	**0.03**	21,033	30,498
	DDFW	100	2.86	5.86	624,106	1,289,211
50v15d80c	RSAPS	47	60.60	44.90	10,000,000	7,401,787
	AdaptNovelty$^+$	100	**0.45**	**0.59**	187,247	243,750
	DDFW	100	130.00	206.00	4,978,679	7,543,275
50v15d40c	RSAPS	3	57.70	56.40	10,000,000	9,771,653
	AdaptNovelty$^+$	**100**	121.12	169.55	10,866,838	14,848,547
	DDFW	56	578.76	618.37	10,740,827	12,840,382

Table 3.2: Default DDFW results for the PAWS harder problem set (see [79]). The best performing technique on each problem is indicated in bold.

lems and the flat200-hard, uf400-med, uf400-hard and uf250-hard problems (7 out of the 14 problems) making it the best overall approach for these benchmarks. Of the other problems, RSAPS was better on logistics.c, flat100-hard, uf250-hard and ais10 (4 out 14) and AdaptNovelty$^+$ was better on flat200-med, uf100-hard and uf250-med (3 out of 14). In addition, the DPLL algorithms were better than local search on the bw_large.b, flat100-hard and flat200-hard problems.

Table 3.2 shows mixed results with DDFW dominating on the bw_large.d problem and all random 3-SAT f problems except f800-med (where AdaptNovelty$^+$ has the edge). RSAPS has the advantage on both parity 16 problems and AdaptNovelty$^+$ wins on all four binary CSP problems. However, Figure 3.5 shows that DDFW has the more consistent success rates over the whole large

problem set, with an average rate of 87.82% compared to 75.09% for AdaptNovelty$^+$ and 39.36% for RSAPS.

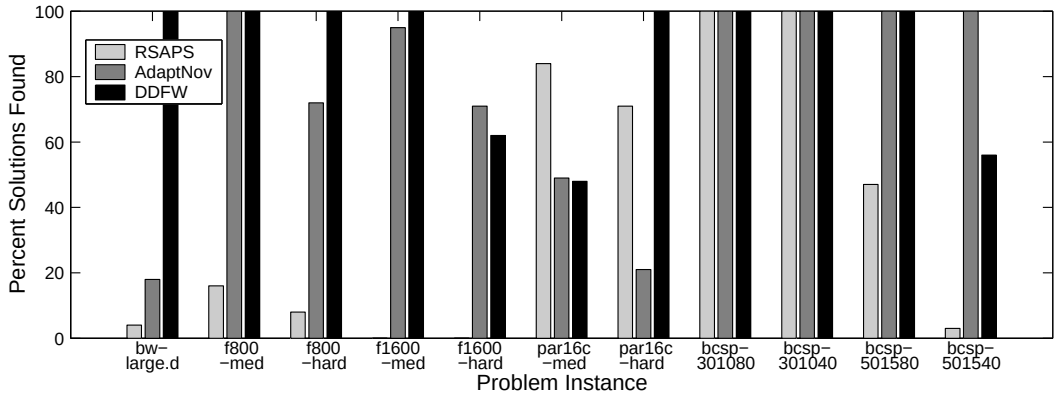


Figure 3.5: Percentage of solutions found by DDFW, RSAPS and AdaptNovelty$^+$

3.4.2 Comparison with PAWS and SAPS

Tables 3.3 and 3.4 show the results for the same two problem sets considered in the previous section (with the addition of the two most difficult DIMACS graph colouring problems, g125.17 and g250.29, in Table 3.4). This time we compared DDFW with a tuned W_{init} parameter against tuned versions of SAPS and PAWS (parameter settings are shown in the parameters column of each table). All experiments in this section were performed on a Sun supercomputer with 8 $\times$ Sun Fire V880 servers, each with 8 $\times$ UltraSPARC-III 900MHz CPU and 8GB memory per node, using the same cut-off rules as in the previous section. In tables 3.3 and 3.4, the Wilcoxon values give the probability that the null hypothesis $A \geq B$ is true, where A is the distribution of flips (indicated by an 'f') or run-times (indicated by a 't') that has the *smallest* rank-sum value, and B has the next smallest rank-sum. We record P-values against distribution A and take $P < 0.05$ to indicate that A is significantly less than B, marking such results with '*'. The Wilcoxon rank sum test provides a reliable estimate of statistically significant differences between non-normal distributions, and so is ideal for comparing local search performance (for more details see [79]). In addition, we provide a time comparison with two state-of-the-art complete solvers, Satz [49] and zChaff [57], in the DPLL column (only the best performing algorithm is reported for each problem). These figures are provided to indicate those problems that are beyond the reach of complete search.

The results show that a tuned DDFW is very good on the blocks world planning problems

			Success	Time(secs)		Flips		Wilcoxon	DPLL
Problem	Method	Parameters	%	median	mean	median	mean	$\frac{t:time}{f:flips}$	secs
bw_large.a	SAPS	P:6 α:1.30 ρ:0.80	100	0.01	0.01	2,184	2,824		0.01
	PAWS	Max_{inc}:34	100	0.01	0.01	2,518	3,235		
	DDFW	W_{init}:2	100	0.01	0.01	1,479	1,800	$0.1338t$ $*0.0000f$	
bw_large.b	SAPS	P:5 α:1.30 ρ:0.80	100	0.20	0.26	34,488	45,335		*0.01
	PAWS	Max_{inc}:50	100	0.16	0.21	33,480	45,501		
	DDFW	W_{init}:7	100	0.08	0.11	11,335	15,788	$*0.0000t$ $*0.0000f$	
bw_large.c	SAPS	P:7 α:1.10 ρ:0.60	100	17.63	26.45	1,366,319	2,103,352		0.53
	PAWS	Max_{inc}:5	100	4.74	6.87	798,389	1,181,032		
	DDFW	W_{init}:4	100	1.00	1.34	91,668	122,444	$*0.0000t$ $*0.0000f$	
logistics.c	SAPS	P:5 α:1.30 ρ:0.90	100	0.04	0.05	6,954	8,436		0.08
	PAWS	Max_{inc}:∞	100	0.02	0.03	5,229	6,771	$*0.0000t$	
	DDFW	W_{init}:3	100	0.03	0.04	4,934	6,224	$*0.0000f$	
flat100-med	SAPS	P:5 α:1.30 ρ:0.40	100	0.01	0.01	5,415	7,460		0.01
	PAWS	Max_{inc}:13	100	0.01	0.01	6,402	8,628		
	DDFW	W_{init}:6	100	0.01	0.01	4,536	6,662	$*0.0006t$ $*0.0000f$	
flat100-hard	**SAPS**	P:6 α:1.30 ρ:0.80	100	0.04	0.06	21,965	31,812	$0.4064t$ $*0.0010f$	*0.01
	PAWS	Max_{inc}:46	100	0.04	0.06	26,065	36,178		
	DDFW	W_{init}:4	100	0.06	0.08	22,375	31,443		
flat200-med	SAPS	P:5 α:1.30 ρ:0.40	100	0.12	0.17	57,411	83,558		0.12
	PAWS	Max_{inc}:10	100	0.10	0.13	48,990	67,781	$*0.0000t$ $*0.0002f$	
	DDFW	W_{init}:6	100	0.22	0.34	59,407	86,107		
flat200-hard	SAPS	P:5 α:1.30 ρ:0.40	100	4.86	6.38	3,173,188	3,397,088		*0.03
	PAWS	Max_{inc}:74	99	4.34	5.90	2,354,944	3,224,432		
	DDFW	W_{init}:8	100	0.32	2.40	706,533	943,515	$*0.0000t$ $*0.0000f$	
uf100-hard	**SAPS**	P:5 α:1.30 ρ:0.80	100	0.01	0.01	2,857	4,250	$0.2460t$ $*0.0444f$	0.01
	PAWS	Max_{inc}:40	100	0.01	0.01	3,282	4,579		
	DDFW	W_{init}:2	100	0.06	0.08	17,283	25,106		
uf250-med	SAPS	P:6 α:1.30 ρ:0.40	100	0.01	0.02	4,895	7,050		1.25
	PAWS	Max_{inc}:11	100	0.01	0.01	3,795	5,040	$*0.0000t$ $*0.0000f$	
	DDFW	W_{init}:2	100	0.02	0.03	5,939	8,515		
uf250-hard	**SAPS**	P:5 α:1.30 ρ:0.70	100	0.41	0.56	160,710	223,593	$*0.0437t$ $*0.0000f$	0.32
	PAWS	Max_{inc}:18	100	0.52	0.79	213,393	320,273		
	DDFW	W_{init}:3	100	0.71	1.03	194,769	279,684		
uf400-med	SAPS	P:5 α:1.30 ρ:0.20	100	0.12	0.17	42,514	61,159		57.81
	PAWS	Max_{inc}:9	100	0.08	0.10	28,601	38,363	$*0.0000t$ $0.2633f$	
	DDFW	W_{init}:4	100	0.11	0.16	28,222	39,867		
uf400-hard	SAPS	P:5 α:1.30 ρ:0.20	100	2.06	4.01	744,592	1,446,987		178.92
	PAWS	Max_{inc}:8	100	1.71	2.28	699,892	929,791		
	DDFW	W_{init}:12	100	1.79	2.08	666,149	686,707	$0.3842t$ $0.4011f$	
ais10	SAPS	P:4 α:1.30 ρ:0.90	100	0.06	0.10	11,708	18,085	$*0.0000f$	0.06
	PAWS	Max_{inc}:52	100	0.06	0.09	13,661	19,594	$0.4243t$	
	DDFW	W_{init}:4	100	0.14	0.20	13,516	19,109		

Table 3.3: Tuned DDFW results for the original SAPS problem set. The best performing local search technique on each problem is indicated in bold. A '*' in the DPLL column indicates that a DPLL technique had the best performance for the associated problem.

with up an order of magnitude of improvement. It is also good on logistics.c (another planning problem) with significantly better flips (but not time). Hence we have an indication that neighbourhood weighting is especially suited for planning problems. There are further signs of good

performance on the flat graph colouring problems (with a small advantage on flat100-med and a $\times 3$ improvement on flat200-hard). DDFW also does well on the uf400-hard and f1600-hard problems but the Wilcoxon tests do not show significance. A further run-time distribution (RTD) analysis [33] shows that PAWS has the initially better performance on these problems, but as the RTDs cross, DDFW has the better long-term performance (see Figure 3.6). On the remainder of the problems DDFW produced reasonable results, often matching SAPS, but performing relatively badly on some of the larger instances. More specifically, DDFW was unable to solve the two large graph colouring problems, g125.17 and g250.29 (where g250.29 has 454,622 clauses). In these cases, we conjecture that the large numbers of clauses and variables involved may have rendered the DDFW weight transfer relatively ineffective as a method of guidance. However, as DDFW has outstanding performance on the large blocks word problems (bw_large.d has 131,973 clauses), we cannot conclude that size alone is the determinant of performance. Rather, as DDFW exploits neighbourhood relationships, it is reasonable to assume that certain kinds of neighbourhood relationships will favour weight redistribution. The characterisation of these relationships presents an interesting direction for future research.

Overall PAWS has the better time performance on 14 of the 27 test problems, DDFW is better on 8 problems and SAPS on 5. PAWS therefore remains the stronger algorithm when parameter is allowed, but DDFW does perform better than SAPS, and if we consider success rates as the determining factor, the gap between PAWS and DDFW becomes narrower (12 versus 10).[5] However, DDFW does have the superior performance when parameter tuning is not allowed. This is evidenced by it outperforming AdaptNovelty$^+$ and RSAPS in the current study and by the poor performance of a default valued PAWS in the recent SAT competitions where it has been consistently outperformed by AdaptNovelty$^+$.

Finally, although tuning DDFW's W_{init} parameter does not increase its performance beyond that of a tuned PAWS, it does significantly improve DDFW's performance (over a default W_{init} setting of 8) on most problems. In particular, large improvements were observed on the logistics, ais, parity and binary CSPs. This can be observed by comparing the flip counts of DDFW between tables 3.1 and 3.3 and between tables 3.2 and 3.4 (note the times are not directly comparable between these pairs of tables as the results were obtained on different machines).

[5] Although DDFW did not have the best average time performance on the parity 16 problems or the 50v15d40c binary CSP, it did have the best success rates.

			Success	Time(secs)		Flips		Wilcoxon	DPLL
Problem	Method	Parameters	%	median	mean	median	mean	$\frac{t:time}{f:flips}$	secs
bw_large.d	SAPS	P:5 $\frac{\alpha:1.05}{\rho:0.80}$	100	29.22	37.87	1,868,733	2,398,205		2.01
	PAWS	Max_{inc}:4	100	7.07	10.87	903,962	1,432,780		
	DDFW	W_{init}:3	100	1.75	1.95	105,482	126,499	$*0.0000t$ $*0.0000f$	
f800-med	SAPS	P:5 $\frac{\alpha:1.25}{\rho:0.10}$	100	0.59	0.91	169,562	263,105		timed
	PAWS	Max_{inc}:9	100	0.26	0.36	82,392	115,451	$*0.0000t$ $*0.0000f$	out
	DDFW	W_{init}:15	100	1.44	1.80	287,990	379,971		
f800-hard	SAPS	P:5 $\frac{\alpha:1.25}{\rho:0.30}$	100	4.88	6.12	1,414,621	1,754,017		timed
	PAWS	Max_{inc}:10	100	2.58	3.18	897,696	1,087,076	$*0.0000t$ $*0.0011f$	out
	DDFW	W_{init}:15	100	3.85	5.80	786,419	1,202,206		
f1600-med	SAPS	P:5 $\frac{\alpha:1.25}{\rho:0.30}$	99	3.06	5.94	693,385	1,303,941		timed
	PAWS	Max_{inc}:10	100	0.98	1.74	284,591	548,332	$*0.0000t$ $*0.0000f$	out
	DDFW	W_{init}:17	100	2.89	3.60	488,273	602,977		
f1600-hard	SAPS	P:5 $\frac{\alpha:1.25}{\rho:0.30}$	94	30.62	34.34	6,499,140	7,777,980		timed
	PAWS	Max_{inc}:11	96	11.94	18.86	3,000,027	5,019,099		out
	DDFW	W_{init}:17	100	13.34	16.80	2,140,507	2,689,019	$0.4892t$ $0.4703f$	
par16-med	SAPS	P:7 $\frac{\alpha:2.00}{\rho:0.25}$	88	12.32	20.98	5,589,195	7,720,965		*1.52
	PAWS	Max_{inc}:36	97	5.32	8.77	2,646,531	4,496,763	$*0.0001t$ $*0.0004f$	
	DDFW	W_{init}:2	100	12.20	17.10	4,341,378	6,064,188		
par16-hard	SAPS	P:4 $\frac{\alpha:1.40}{\rho:0.90}$	86	13.78	18.71	6,454,597	9,725,495		*0.57
	PAWS	Max_{inc}:40	98	6.79	9.51	3,379,909	4,809,418	$*0.0000t$ $*0.0000f$	
	DDFW	W_{init}:40	100	21.10	29.50	7,380,532	10,280,304		
g125.17	SAPS	P:5 $\frac{\alpha:1.20}{\rho:0.05}$	97	55.59	81.65	2,772,017	4,187,750		timed
	PAWS	Max_{inc}:4	100	7.91	10.89	596,447	841,063	$*0.0000t$ $*0.0000f$	out
	DDFW		0	timed out	timed out	timed out	timed out		
g250.29	SAPS	P:6 $\frac{\alpha:1.15}{\rho:0.10}$	90	100.14	219.92	563,389	2,622,915		timed
	PAWS	Max_{inc}:4	100	19.73	21.89	275,782	315,937	$*0.0000t$ $*0.0000f$	out
	DDFW		0	timed out	timed out	timed out	timed out		
30v10d80c	SAPS	P:6 $\frac{\alpha:1.30}{\rho:0.10}$	100	0.06	0.08	8,661	12,299		0.26
	PAWS	Max_{inc}:7	100	0.04	0.06	7,576	10,633	$*0.0000t$ $0.0066f$	
	DDFW	W_{init}:2	100	1.01	0.70	49,577	72,142		
30v10d40c	**SAPS**	P:6 $\frac{\alpha:1.25}{\rho:0.50}$	100	0.08	0.12	13,716	19,711	$0.4072t$ $*0.0101f$	0.02
	PAWS	Max_{inc}:7	100	0.08	0.12	15,926	22,422		
	DDFW	W_{init}:2	100	0.92	1.42	80,793	122,629		
50v15d80c	SAPS	P:6 $\frac{\alpha:1.20}{\rho:0.10}$	100	1.81	2.92	119,552	186,552		timed
	PAWS	Max_{inc}:5	100	1.44	1.85	128,837	168,402	$*0.0062t$ $0.3574f$	out
	DDFW	W_{init}:70	100	9.73	14.25	285,958	388,727		
50v15d40c	**SAPS**	P:5 $\frac{\alpha:1.25}{\rho:0.25}$	99	96.84	149.05	7,579,338	11,748,482	$0.2535t$ $0.0735f$	timed
	PAWS	Max_{inc}:6	98	121.12	169.55	10,866,838	14,848,547		out
	DDFW	W_{init}:400	100	409.37	529.76	8,740,827	10,840,382		

Table 3.4: Tuned DDFW results for the PAWS harder problem set. The best performing technique on each problem is indicated in bold. A '*' in the DPLL column indicates that a DPLL technique had the best performance for the associated problem.

3.5 Summary

In summary, DDFW represents a powerful general purpose SAT solver for problem domains where extensive parameter tuning is not practical. The work on DDFW also represents a first step in the development of a weight redistribution approach to clause weighting, and shows a simple

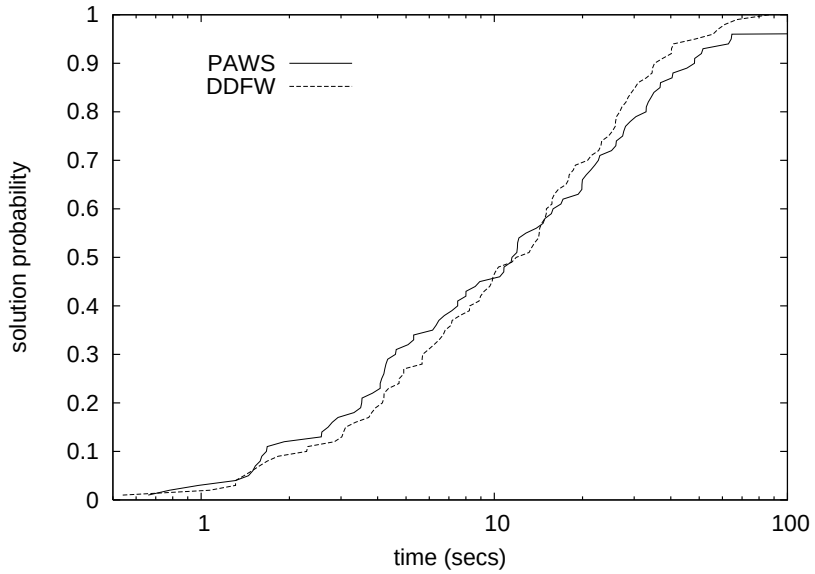


Figure 3.6: Run-Time Distribution for DDFW and PAWS on the f1600-hard 3SAT problem

way that neighbourhood structure can be used to guide weight redistribution decisions.

In comparison to AdaptNovelty+ and RSAPS, we have shown that DDFW performs significantly better on a large set of problems and in general is able to solve a larger number of problem instances. DDFW introduces a new direction to SAT clause weighting research. Current clause weighting algorithms follow a two stage process of weight smoothing and differ mainly on whether the smoothing is done additively or multiplicatively. In contrast, DDFW combines the weight increases and decreases into one step by redistributing weights among the clauses in a local minimum. Fig 3.7 shows a new categorisation of clause weighting techniques algorithms and illustrates how DDFW has extended the area.

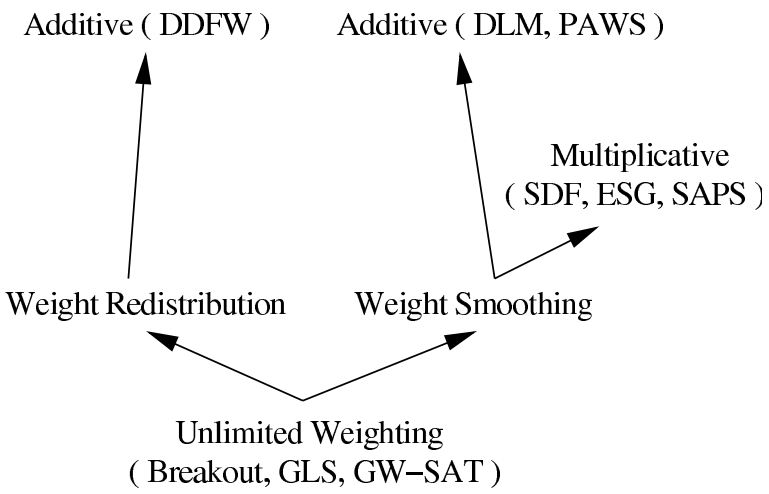


Figure 3.7: Clause weight algorithms for SAT with the new weight redistributing branch

Chapter 4

Enhancing DDFW's Performance

4.1 Introduction

A key weakness of clause weighting dynamic local search (DLS) approaches is that their performance depends on problem specific parameter tuning. This issue was partly addressed in the development of a reactive version of SAPS (RSAPS [41]) which used a similar adaptive noise mechanism to that used in AdaptNovelty$^+$ [36]. Nevertheless, as the 2005 International SAT competition (SAT2005) has shown, DLS algorithms, including RSAPS, have not proved competitive with the best SLS techniques when they are constrained to use fixed parameter values. This is explained by the sensitivity of the control parameters and by the lack of a sufficiently effective adaptive mechanism to adjust these parameters to specific problem instances.

As previously discussed in Chapter 3, DDFW's approach is to redistribute weight from satisfied to unsatisfied clauses in each local minimum, unifying the increase and decrease phases of weight control into a single action. This means there is no requirement for a problem specific parameter to decide when weight is to be reduced. In addition, DDFW only alters weights on those clauses that are false in a local minimum and an equal number of satisfied clauses. This makes it more efficient than earlier weight smoothing algorithms that also performed smoothing at each local minimum, but did so by adjusting weight on all the clauses in the problem (e.g. SDF [67]).

However, DDFW still has a parameter (W_{init}) whose setting can effect performance by varying the amount of weight that is initially given to each clause (see Chapter 3). The existence of such a parameter implies that DDFW could benefit from an adaptive mechanism to vary the amount of weight that is distributed according to the dynamic search conditions.

Also in 2005, it was shown that the performance of various local search (including weighting-

based) techniques can be significantly improved by the addition of a resolution-based preprocessing phase [1]. This work initially produced the winning algorithm in the SAT2005 satisfiable random problem category, R+AdaptNovelty$^+$. However, in the subsequent paper [1], the largest performance gains were obtained for clause weighting algorithms solving *structured* problem instances. Here R+AdaptNovelty$^+$ was convincingly outperformed by a R+RSAPS and a *tuned* version of R+PAWS on a range of quasigroup existence problems and standard structured SAT benchmarks.

The question we address in the current chapter is which SLS SAT algorithm should be preferred in situations where parameter tuning is impractical and we have no other information that could guide us in choosing a particular approach. As this is exactly the situation we would expect to find in many real world applications, we take the relevance and importance of this question to be self evident. While the initial work on DDFW showed that a fixed parameter version was able to outperform AdaptNovelty$^+$ and RSAPS on a range of random and structured SAT benchmarks, the question still remains whether the performance of DDFW can be further improved by incorporating a similar adaptive mechanism to that used by AdaptNovelty$^+$ and RSAPS to control the W_{init} parameter.

It also remains unanswered whether such an adaptive version of DDFW could derive enough benefit from resolution-based preprocessing to outperform the existing resolution-based versions of R+AdaptNovelty$^+$ or R+RSAPS. In addition, in SAT2005 a new SLS algorithm was introduced, G^{2}WSAT [51], which went on to win the silver medal in the random category of the competition. This algorithm has subsequently been improved and it too has yet incorporate a resolution-based preprocessor.

As a result of these considerations, our specific aim in the remainder of the chapter is to introduce an adaptive resolution-incorporating version of DDFW (called R+DDFW$^+$) and to compare it with the three other most promising general purpose local search SAT solvers, namely R+AdaptNovelty$^+$, R+RSAPS and an enhanced R+G^{2}WSAT. On the basis of an empirical study that covers a range of problems from SAT2005, the quasigroup existence domain and the SATLIB benchmark library, we conclude that R+DDFW$^+$ has the best overall performance of these methods, and that it derives significant benefits from its new adaptive mechanism.

4.2 Clause Weighting for SAT

As we have already stressed, the performance of clause weighting algorithms remains sensitive to the settings of their problem specific parameters (this has been shown in detail in [77]). While this sensitivity is also a problem for the non-weighting algorithms of the WalkSAT family, it has been somewhat counteracted by the use of heuristics that adapt parameter settings during the course of the search. The most successful of these algorithms, AdaptNovelty$^+$, works by adapting a noise parameter that controls whether a move is selected randomly or deterministically [36]. In simplified terms, the likelihood of making a random choice is increased the longer the search continues without achieving an improvement in the objective function. A similar scheme was added to SAPS, producing reactive SAPS or RSAPS [41]. However, adapting SAPS was not as successful as adapting Novelty, for, while a tuned SAPS generally produces better performance than a tuned Novelty+, RSAPS has not been able to reach the consistent performance AdaptNovelty$^+$ in the recent SAT competitions. One reason for this may be that SAPS requires the setting of *three* parameters to achieve its best performance, while RSAPS only adapts one of these parameters. Similarly, DLM requires the setting of at least three parameters before producing its best performance. In contrast, PAWS (like Novelty) only requires the tuning of a single parameter, but to date no successful heuristic has been discovered that can automatically adapt this value.

More recently, work has concentrated on *learning* empirical hardness models in order to predict the best parameter settings for SAPS [39]. This approach requires a set of training instances that are repeatedly solved by SAPS using different parameter settings. After this training phase, parameter settings can be generated for previously unseen instances taken from the same problem class. Results from this work are encouraging and could be generally applied to other local search algorithms. However, the weakness is that training is required on a representative test set before good predictions can be produced. It remains to be seen whether a general model can be devised that can predict good parameter settings for the SAT domain as a whole. In the meantime, if we are limited to solving problems from an undisclosed problem distribution and if manual parameter tuning is ruled out of court, then the best available clause weighting algorithm is DDFW.

4.2.1 Adapting DDFW

The new feature introduced in this chapter is the development of an adaptive mechanism that alters the total amount of weight that DDFW distributes according to the degree of stagnation in the search. This DDFW$^+$ heuristic is detailed in lines 12-27 of Algorithm 8. Previously

Algorithm 8 DDFW^{+}($\mathcal{F}$, $MaxTime$)

```
1:  A ← a randomly generated truth assignment of F
2:  set the weight w_a of each clause c_a ∈ F to 2;
3:  set the minimum m to the number of false clauses c_f ∈ F;
4:  set counter i to 0 and boolean b to false;
5:  while time < MaxTime do
6:    if A satisfies F then
7:      return A
8:    else
9:      select list L of literals causing the greatest reduction in weighted cost Δw when flipped
10:     if (Δw < 0) or (Δw = 0 and probability ≤ 15%) then
11:       update A by flipping a literal randomly selected from L
12:       if number of false clauses < m then
13:         set counter i to 0 and minimum m to the number of false clauses;
14:       else
15:         increment counter i by 1;
16:         if i ≥ number of literals in F then
17:           set counter i to 0;
18:           if b is false then
19:             increase the weight w_a of each clause c_a ∈ F by 1;
20:             set boolean b to true;
21:           else
22:             set the weight w_s of each satisfied clause c_s ∈ F to 2;
23:             set the weight w_f of each false clause c_f ∈ F to 3;
24:             set boolean b to false;
25:           end if
26:         end if
27:       end if
28:     else
29:       for each false clause c_f ∈ F do
30:         select a satisfied same sign neighbouring clause c_n with maximum weight w_n;
31:         if w_n < 2 then
32:           randomly select a clause c_n with weight w_n ≥ 2;
33:         end if
34:         if w_n > 2 then
35:           transfer a weight of 2 from c_n to c_f;
36:         else
37:           transfer a weight of 1 from c_n to c_f;
38:         end if
39:       end for
40:     end if
41:   end if
42: end while
43: return no satisfying solution found
```

DDFW would have initialised the weight of each clause to W_{init} (which was fixed at 8). Now this initialisation value is set at two in line 2 of Algorithm 8, but can be altered during the search as follows: if the search executes a consecutive series of i flips without reducing the total number of false clauses, where i is equal to the number of literals in the problem, then the amount of weight on each clause is increased by one in the first instance. However, if after increasing weights, the search enters another consecutive series of i flips without improvement, then it will reset the weight on each satisfied clause back to two and on each false clause back to three. The search then continues to follow each increase with a reset and each reset with an increase. In this way a long period of stagnation will produce oscillating phases of weight increase and reduction, such that the total weight can never exceed 3 times the total number of clauses $c_a \in \mathcal{F}$ plus the total number of false clauses $c_f \in \mathcal{F}$ (see Figure 4.1).

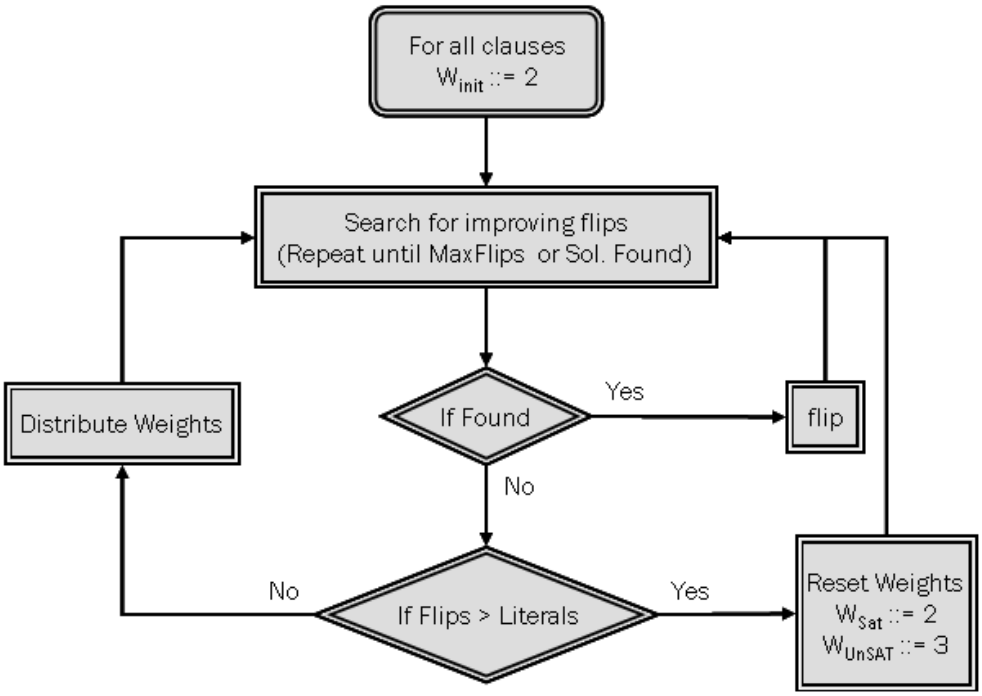


Figure 4.1: DDFW$^+$ flow chart

The reasoning behind this adaptive heuristic is based on the results from Chapter 3. These results showed that manually adjusting DDFW's original parameter W_{init} has a noticeable effect runtime performance, and that on several problems the default value of eight was not optimal. This is further illustrated in Figure 4.2, which shows that on the flat200-hard problem in (a) $W_{init} = 8$ is near optimal whereas on the bw_large.d problem in (b) $W_{init} = 2$ is the better choice (if we consider the underlying trend). We conjectured that we could circumvent the need

to initialise the clauses with more weight at the start of the search by allowing context sensitive weight increases during the search. Hence we developed a stagnation measure, much like the measures used in AdaptNovelty and RSAPS, that injects extra weight when no cost improvement occurs and made the frequency of this injection depend on the size of the problem. The unusual feature of the DDFW$^+$ heuristic is that the search will only effect one increase after which, if stagnation is observed again, the weights are reset. This reset mechanism was adopted after a series of empirical trials that tested various combinations of weight increase and decrease phases. Our main difficulty was to keep the weight growth within bounds and we could find no decrease scheme that worked well across a wide range of problems without requiring a further problem dependent parameter (which would obviously defeat the purpose of the study). We therefore settled on a simple reset strategy that places a strict limit on weight growth and avoids adding an additional parameter.

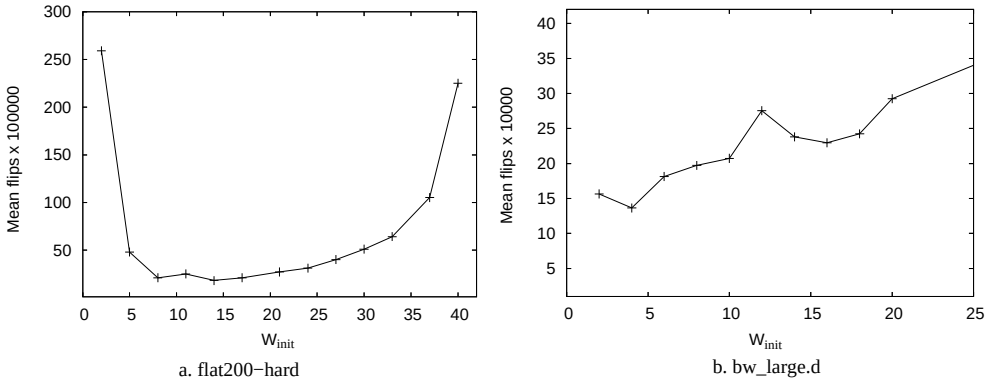


Figure 4.2: Flip performance of DDFW for various settings of the W_{init} parameter

4.3 Resolution-Based Preprocessing

As discussed in the introduction, significant performance benefits have been gained by preprocessing a problem using resolution before starting a search. This result is already well-known in the complete search community, where Satz [49] uses a restricted resolution procedure, adding resolvents of length ≤ 3, as a preprocessor before running the complete backtrack search. The same procedure has now been added to AdaptNovelty$^+$, PAWS, RSAPS and WalkSAT [1], and there is empirical evidence to suggest that clause weighting algorithms in particular benefit from this approach when solving structured real-world problems.

As previous discussed in Chapter 2, resolution itself is a rule of inference widely used in automated deduction [62, 16, 63]. In the present chapter, as in [1], we implement the Satz resolution

Algorithm 9 ComputeResolvents($\mathcal{F}$)

```
1: for each clause c_1 of length ≤ 3 in F do
2:   for each literal l of c_1 do
3:     for each clause c_2 of length ≤ 3 in F s.t. l̄ ∈ c_2 do
4:       Compute resolvent r = (c_1 \ {l}) ∪ (c_2 \ {l̄})
5:       if r is empty then
6:         return unsatisfiable
7:       else
8:         if r is of length ≤ 3 then
9:           F := F ∪ {r}
10:        end if
11:      end if
12:    end for
13:  end for
14: end for
```

process shown in Algorithm 9 (which reproduces Algorithm 1 from Chapter 2) as follows: when two clauses of a CNF formula have the property that some variable x_i occurs positively in one and negatively in the other, the resolvent of the clauses is a disjunction of all the literals occurring in the clauses except x_i and $\overline{x_i}$. For example, the clause $(x_2 \vee x_3 \vee \overline{x_4})$ is the resolvent for the clauses $(\overline{x_1} \vee x_2 \vee x_3)$ and $(x_1 \vee x_2 \vee \overline{x_4})$ and is added to the clause set. The new clauses, provided they are of length ≤ 3, can in turn be used to produce other resolvents. The process is repeated until saturation. Duplicate and subsumed clauses are deleted, as are tautologies and any duplicate literals in a clause. It is worth noting that this resolution phase takes polynomial time.

4.4 Experimental Evaluation

As the resolution process is encapsulated in a preprocessing phase, it can be added to an existing SAT solver as a separate module, leaving the original solver unaltered. In our experimental study we added this preprocessing phase (as defined in Algorithm 9) to DDFW, DDFW$^+$, RSAPS, AdaptNovelty$^+$ and G^{2}WSAT, producing R+DDFW, R+DDFW$^+$, R+RSAPS, R+AdaptNovelty$^+$ and R+G^2-WSAT. Of these algorithms, R+RSAPS and R+AdaptNovelty$^+$ have already been entered into SAT2005 and reported in [1].[1] However, R+DDFW, R+DDFW$^+$ and R+G^{2}WSAT are new algorithms whose performance has yet to be reported.[2] We chose to compare DDFW with R+AdaptNovelty$^+$ and R+G^{2}WSAT because these two algorithms were the gold and silver medal winners in the SAT2005 satisfiable random category competition and achieved the best overall lo-

[1] AdaptNovelty$^+$ and RSAPS are available as part of the UBCSAT solver from http://www.satlib.org/ubcsat/

[2] G^{2}WSAT is available at http://www.laria.u-picardie.fr/%7Ecli/g2wsat2005.c. This latest version is described by the authors as generally more than 50% faster than the version entered in SAT2005.

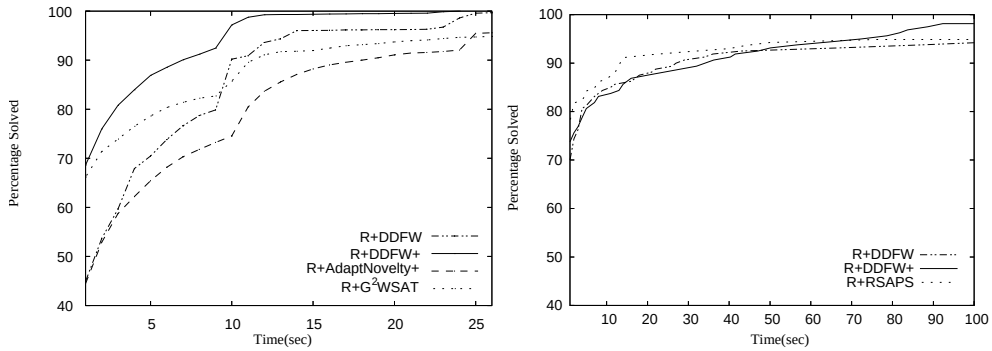
cal search results in terms of the number of problems solved [72]. We chose R+RSAPS because it was the best performing clause weighting algorithm in the competition. Together, therefore, these three algorithms can lay claim to being the state-of-the-art for general purpose local search SAT solving when manual parameter tuning is disallowed.

To evaluate the relative performance of the various techniques we divided our empirical study into four areas: firstly, we attempted to reproduce a reduced problem set similar to that used in the random category of the SAT competition (as this is the domain where local search techniques have dominated). To do this we selected the 50 satisfiable $k3$ problems from the SAT2004 competition benchmark [48]. Secondly, we obtained the 10 SATLIB quasigroup existence problems used in [1]. These problems are relevant because they exhibit a balance between randomness and structure, while also producing clause sets to which resolution can be applied effectively. Thirdly, we obtained the structured problem set used to originally evaluate SAPS [41]. These problems have been widely used to evaluate clause weighting algorithms (e.g. in [79]) and contain a representative cross-section taken from the DIMACS and SATLIB libraries. In this set we also included 4 of the well-known DIMACS 16-bit parity learning problems. Finally, we used the 16 ferry planning problems from the SAT2005 competition that our local search techniques were able to solve. This was to give an indication of relative performance on the SAT2005 industrial problems [72].

Overall, the problem set is designed to show how R+DDFW$^+$ compares in absolute terms to the other algorithms and to examine the relative effect of the adaptive mechanism on differing problem classes. For this reason we also include the results for R+DDFW (i.e. *without* the adaptive mechanism). All experiments were performed on a Dell machine with a 3.1GHz CPU and 1GB memory, except for the quasigroup problems which were run on a Sun supercomputer with 8 $\times$ Sun Fire V880 servers, each with 8 $\times$ UltraSPARC-III 900MHz CPU and 8GB memory per node. Cut-offs for the various algorithms were set as follows: first R+DDFW was given 10 trials on each problem with a flip cut-off of 1,000,000. If it was unable to solve any trial then the cut-off was raised to 10,000,000, and then in steps of 10,000,000 until at least one solution was found. R+DDFW was then allowed 100 trials at the given flip cut-off for all instances except the ferry problems, where it was limited to 10 trials. The total time allowed for R+DDFW on each set of 10 or 100 trials was then recorded and all other algorithms were given this as a time cut-off on each problem. The following results detail the mean time in seconds (including the resolution preprocessing step), mean flips and the success rate for these cut-offs (results in bold indicate the

best performance for a particular problem).

4.4.1 SAT Competition Problem Results



a. Random 3SAT problems (50 Instances) b. Industrial Ferry problems (16 Instances)

Figure 4.3: Results for the SAT2004 random problems and SAT2005 industrial problems

The results in Figure 4.3a graph the performance of R+DDFW$^+$, R+DDFW, R+AdaptNovelty$^+$ and R+G^{2}WSAT after applying resolution on the 50 $k3$ problems from the SAT2004 competition (as R+RSAPS had very poor performance on the random instances it has been omitted from the figure and the following discussion). A more detailed comparison of R+DDFW and R+AdaptNovelty$^+$ is also given in Table 4.1. The graph shows the cumulative percentage of problems solved against runtime, assuming that each instance is solved in parallel (for example, in Figure 4.3a after 5 seconds approximately 71% of the 50×100 trials for R+DDFW will have terminated). Here R+DDFW$^+$ and R+DDFW were the only solvers that could reach a 100% success rate over all trials. Although R+G^{2}WSAT was competitive and could solve the easier problems faster than R+DDFW, it was unable to match R+DDFW as problem difficulty increased. Overall the graph shows that R+DDFW$^+$ has the superior performance across the range of problem sizes, clearly dominating R+DDFW and thereby demonstrating that the new adaptive heuristic can positively affect runtime performance. Figure 4.3a also shows that R+G^{2}WSAT generally dominates R+AdaptNovelty$^+$, although R+AdaptNovelty$^+$ does match R+G^{2}WSAT's success rate over the whole problem set.

The results for the SAT2005 industrial ferry problems are shown in Figure 4.3b and in Table 4.2 (as R+G^{2}WSAT and R+AdaptNovelty$^+$ were only able to solve 29% and 9% of the ferry instances respectively, they have been removed from the graphical analysis). Looking at Figure 4.3b we can see that R+RSAPS, after performing poorly on the random problems, is now able to dominate R+DDFW across the range of the ferry problems, but cannot quite reach R+DDFW$^+$'s

	R+DDFW			R+AdaptNovelty$^+$		
Problems	Time	Flips	% Solved	Time	Flips	% Solved
k3-r4.25-v400-33	0.06	46,452	100	0.10	143,074	100
k3-r4.25-v400-34	0.01	8,967	100	0.02	22,283	100
k3-r4.25-v400-35	0.01	12,830	100	0.02	25,480	100
k3-r4.25-v400-37	0.07	56,635	100	0.10	257,930	100
k3-r4.25-v400-38	0.02	17,933	100	0.03	40,751	100
k3-r4.25-v400-39	0.05	39,888	100	0.15	203,149	100
k3-r4.25-v400-38	0.02	17,933	100	0.03	40,751	100
k3-r4.25-v450-41	0.28	184,418	100	0.37	489,443	100
k3-r4.25-v450-42	1.19	831,579	100	3.49	4,611,134	100
k3-r4.25-v450-43	0.08	63,070	100	0.19	254,503	100
k3-r4.25-v450-47	0.07	53,913	100	0.04	58,562	100
k3-r4.25-v450-49	0.02	20,256	100	0.03	49,609	100
k3-r4.25-v500-51	0.02	16,767	100	0.05	65,012	100
k3-r4.25-v500-52	3.05	1,721,031	100	24.10	3,157,813	100
k3-r4.25-v500-53	0.07	41,972	100	0.11	136,401	100
k3-r4.25-v500-54	0.17	100,065	100	0.41	531,958	100
k3-r4.25-v500-56	0.06	36,099	100	0.06	80,578	100
k3-r4.25-v500-58	0.11	68,085	100	0.07	95,607	100
k3-r4.25-v500-59	0.38	218,235	100	0.74	983,268	100
k3-r4.25-v550-61	0.02	12,253	100	0.01	23,579	100
k3-r4.25-v550-62	0.68	372,227	100	1.74	2,262,415	100
k3-r4.25-v500-63	0.02	12,424	100	0.02	28,447	100
k3-r4.25-v550-64	0.45	253,262	100	0.57	757,361	100
k3-r4.25-v550-65	0.64	353,394	100	10.80	14,235,051	100
k3-r4.25-v550-67	0.05	28,446	100	0.06	87,572	100
k3-r4.25-v550-68	0.05	35,069	100	0.11	150,459	100
k3-r4.25-v600-72	0.10	54,744	100	0.20	230,104	100
k3-r4.25-v600-73	0.37	206,142	100	0.62	804,070	100
k3-r4.25-v600-74	0.23	125,654	100	0.22	286,185	100
k3-r4.25-v600-76	0.19	104,744	100	0.86	1,116,273	100
k3-r4.25-v600-77	0.60	330,134	100	0.70	797,490	100
k3-r4.25-v600-78	0.14	79,034	100	0.23	305,322	100
k3-r4.25-v600-79	0.06	34,944	100	0.11	105,933	100
k3-r4.25-v650-83	3.42	1,783,286	100	15.30	19,657,241	4
k3-r4.25-v650-85	3.72	199,823	100	5.23	689,776	100
k3-r4.25-v650-86	5.52	2,755,963	100	11.60	14,928,321	41
k3-r4.25-v650-87	0.05	29,121	100	0.05	75,746	100
k3-r4.25-v650-88	0.43	230,375	100	0.55	718,221	100
k3-r4.25-v700-91	1.09	558,630	100	2.31	2,954,713	100
k3-r4.25-v700-93	0.13	70,443	100	0.15	201,414	100
k3-r4.25-v700-94	0.34	180,341	100	0.58	756,007	100
k3-r4.25-v700-98	1.53	777,937	100	5.98	7,701,814	94
k3-r4.25-v750-101	0.38	192,686	100	0.44	569,351	100
k3-r4.25-v750-102	0.23	116,900	100	0.27	358,582	100
k3-r4.25-v750-103	0.31	153,110	100	0.66	848,185	100
k3-r4.25-v750-105	17.10	8,053,143	100	20.60	26,354,451	37
k3-r4.25-v750-106	16.88	8,212,224	100	26.20	32,953,590	17
k3-r4.25-v750-109	1.17	577,885	100	4.47	5,686,590	100
k3-r4.25-v800-111	0.09	46,716	100	0.13	167,676	100
k3-r4.25-v800-113	1.67	806,667	100	2.39	3,031,962	100
k3-r4.25-v800-117	1.02	483,182	100	1.51	1,916,301	100
k3-r4.25-v800-119	0.08	41,894	100	0.13	174,240	100

Table 4.1: Results for the random medium sized problems (SAT2004)

	R+DDFW$^+$			R+DDFW			R+AdaptNovelty$^+$			R+G^{2}WSAT			R+RSAPS		
Problems	Time	Flips	%	Time	Flips	%	Time	Flips	%	Time	Flips	%	Time	Flips	%
ferry3994	3.48	2,073,195	100	1.1	786,967	100	*n/a*	*n/a*	0	*n/a*	*n/a*	0	**0.6**	**530,501**	**100**
ferry3995	1.54	933,726	100	0.6	458,302	100	*n/a*	*n/a*	0	*n/a*	*n/a*	0	**0.1**	**89,730**	**100**
ferry3996	0.0	7,903	100	0.0	13,942	100	3.9	8,204,511	20	0.1	275,547	100	**0.0**	**7,741**	**100**
ferry3997	10.3	8,238,690	60	**10.3**	**5,055,539**	**90**	*n/a*	*n/a*	0	*n/a*	*n/a*	0	9.2	6,742,006	50
ferry3998	0.0	6,526	100	0.0	8,586	100	2.1	3,344,936	100	0.1	180,334	100	**0.0**	**5,070**	**100**
ferry3999	9.81	5,312,170	100	3.2	1,908,547	100	*n/a*	*n/a*	0	*n/a*	*n/a*	0	**0.6**	**304,680**	**100**
ferry4000	0.0	31,774	100	0.0	19,280	100	*n/a*	*n/a*	0	1.8	2,442,300	80	**0.0**	**12,771**	**100**
ferry4001	**63.1**	**24,392,288**	**100**	99.4	40,117,368	90	*n/a*	*n/a*	0	*n/a*	*n/a*	0	90.0	54,061,467	80
ferry4002	0.0	9,637	100	0.0	20,336	100	4.8	7,535,284	30	2.1	1,958,552	90	**0.0**	**3,852**	**100**
ferry4003	21.2	10,395,968	100	21.2	7,773,439	50	*n/a*	*n/a*	0	*n/a*	*n/a*	0	**7.2**	**2,884,301**	**100**
ferry4004	0.0	30,348	100	0.1	40,547	100	*n/a*	*n/a*	0	2.4	2,437,826	50	**0.0**	**20,394**	**100**
ferry4006	0.0	14,640	100	0.0	17,697	100	*n/a*	*n/a*	0	4.9	2,616,491	20	**0.0**	**9,160**	**100**
ferry4008	**0.0**	**33,192**	**100**	0.1	51,796	100	*n/a*	*n/a*	0	3.2	2,655,066	20	0.1	42,938	100
ferry4009	**0.0**	**23,163**	**100**	0.1	24,015	100	*n/a*	*n/a*	0	*n/a*	*n/a*	0	0.1	17,612	100
ferry3992	**0.1**	**60,525**	**100**	0.2	102,413	100	*n/a*	*n/a*	0	*n/a*	*n/a*	0	0.2	92,346	100
ferry3993	**0.0**	**26,878**	**100**	0.1	43,595	100	*n/a*	*n/a*	0	7.2	3,399,169	10	0.2	54,742	100

Table 4.2: Results for the SAT2005 industrial ferry planning problems.

97.5% success rate. However, Table 4.2 shows that R+RSAPS is able to solve 10 of the 16 ferry problems faster than either DDFW variant, and that R+DDFW$^+$'s superior success rate is largely based on instance ferry4001. We must therefore conclude that there is little to choose between R+RSAPS and R+DDFW$^+$ on these problems. Nevertheless, R+DDFW$^+$ does more clearly outperform R+DDFW and again demonstrates that the adaptive heuristic can make noticeable improvements.

4.4.2 Quasigroup Problem Results

Table 4.3 shows the performance of the solvers on the quasigroup problems. Here we can see that R+DDFW and R+DDFW$^+$ clearly emerge as the two best solvers, sharing the best results for each instance and both achieving an overall success rate of 100%. Comparing between the two DDFW methods, for the first time it becomes unclear whether the adaptive heuristic has made any difference, as, for most instances the results are comparable. However R+DDFW$^+$ does exhibit noticeably better performance on instance qg1-08, whereas R+DDFW shows equally strong performance on qg7-13. We should therefore conclude that the adaptive mechanism does not change the overall performance of DDFW on this problem set, although it can make a difference, either positively or negatively, on individual instances.

4.4.3 Structured Problem Results

Table 4.4 shows the results for the structured problems taken from the original SAPS problem set [41] and the parity learning problems taken from the original PAWS study [79]. This set

	R+DDFW$^+$			R+DDFW			R+AdaptNovelty$^+$			R+G^{2}WSAT			R+RSAPS		
Problems	Time	Flips	%	Time	Flips	%	Time	Flips	%	Time	Flips	%	Time	Flips	%
qg1-07	**0.0**	**4,388**	**100**	0.1	11,375	100	0.2	14,840	100	0.1	9,600	100	0.1	4,901	100
qg1-08	**10.2**	**352,276**	**100**	21.8	601,271	100	33.8	1,076,689	100	28.8	2,818,904	100	64.6	2,153,008	99
qg2-07	0.0	2,361	100	**0.0**	**2,035**	**100**	0.1	9,094	100	0.1	5,073	100	0.1	2,478	100
qg2-08	**57.5**	**1,556,545**	**100**	60.0	1,346,438	100	77.1	1,906,196	20	79.8	4,569,088	50	71.5	1,879,019	70
qg3-08	**0.1**	**16,867**	**100**	0.1	21,986	100	0.6	78,849	100	0.1	24,534	100	0.2	11,049	100
qg4-09	**0.2**	**25,311**	**100**	0.2	26,123	100	1.5	169,169	100	0.7	142,619	100	1.2	54,920	100
qg5-11	0.2	7,303	100	**0.2**	**6,797**	**100**	2.3	131,924	100	0.4	29,992	100	0.6	11,014	100
qg6-09	0.0	478	100	**0.0**	**466**	**100**	0.0	3,644	100	0.0	686	100	0.6	11,753	100
qg7-09	**0.0**	**292**	**100**	0.0	299	100	0.0	698	100	0.0	412	100	0.0	295	100
qg7-13	9.3	229,258	100	**3.2**	**122,091**	**100**	16.3	5,351,459	56	*n/a*	*n/a*	0	24.9	373,456	10

Table 4.3: Results for Quasigroup SATLIB problems.

comprises of two blocks world planning (bw) problems, two logistics planning instances, two flat graph coloring problems (flat), two all-interval-series problems (ais) and four 16-bit parity learning problems (par16*). The results confirm our earlier observation from the random problem results that G^{2}WSAT does not scale as well as DDFW. In this case R+G^{2}WSAT is the best algorithm on the smaller ais, logistics and flat problems, but is outperformed by R+DDFW on each of the larger instances of these problems. In addition, R+RSAPS has stronger performance than R+DDFW on the ais and par16 problems.

However, the situation changes if we consider the performance of R+DDFW$^+$. In comparison to R+DDFW, R+DDFW$^+$ is better on the ais10, both logistics and all par16 problems, whereas R+DDFW is only better on the ais12 and flat200 problems (the two methods perform identically on the bw problems because the large number of literals mean the adaptive mechanism is not used). These results show that the R+DDFW$^+$ adaptive mechanism has again produced noticeable performance benefits, and has improved the overall behaviour of R+DDFW on this problem set. In addition, if we take a simple count of the number of problems on which R+DDFW$^+$ dominates we can see that it is also the best of the five algorithms considered.

4.5 Analysis

Overall we can conclude that the addition of an adaptive mechanism has improved the performance of DDFW over the entire range of the problem sets we have considered. The strongest dominance was observed on the random 3-SAT and parity problems (shown in Figure 4.3a and Table 4.4 respectively). On the other problems R+DDFW$^+$ improved over R+DDFW on 10 of the 16 ferry problems (in Table 4.2), 6 of the 10 quasigroup problems (in Table 4.3) and stays neutral on the remaining real-world problems (in Table 4.4).

	R+DDFW$^+$			R+DDFW			R+AdaptNovelty$^+$			R+G^{2}WSAT			R+RSAPS		
Problems	Time	Flips	%	Time	Flips	%	Time	Flips	%	Time	Flips	%	Time	Flips	%
ais10	0.0	298,650	100	0.5	498,911	100	1.4	1,214,321	100	**0.0**	**112,044**	**100**	0.0	25,459	100
ais12	5.0	4,036,866	100	2.3	1,934,170	100	10.1	7,328,426	51	2.4	1,854,652	100	**0.2**	**187,743**	**100**
logistics-c	0.0	242,540	100	0.3	414,645	100	0.0	26,696	100	**0.0**	**23,623**	**100**	0.0	5,364	100
logistics-d	**0.1**	**16,708**	**100**	0.1	25,869	100	0.1	109,650	100	0.5	350,711	100	0.1	20,918	100
flat200-m	0.3	262,905	100	0.2	161,902	100	0.2	351,563	100	**0.1**	**150,588**	**100**	0.4	362,786	100
flat200-h	3.2	2,814,221	100	**1.0**	**1,014,878**	**100**	3.6	8,166,964	36	2.4	5,535,185	100	3.5	3,517,562	94
bw_large.c	=	=	**100**	**0.6**	**145,607**	**100**	6.7	5,660,460	67	*n/a*	*n/a*	0	21.3	4,258,483	91
bw_large.d	=	=	**100**	**1.4**	**184,874**	**100**	13.4	7,974,818	38	*n/a*	*n/a*	0	*n/a*	*n/a*	0
par16-1	**4.3**	**3,828,086**	**100**	7.1	5,229,852	50	7.4	15,608,349	15	*n/a*	*n/a*	0	7.4	1,164,862	80
par16-2	23.2	21,670,517	100	27.9	20,542,514	60	36.8	54,634,563	10	*n/a*	*n/a*	0	**16.0**	**17,581,843**	**100**
par16-3	**7.7**	**7,146,517**	**100**	24.4	17,959,087	70	32.7	50,828,991	40	31.8	26,133,070	30	16.0	18,890,265	100
par16-4	**2.9**	**2,699,444**	**100**	11.4	12,800,152	100	26.8	41,099,634	50	26.5	51,205,540	60	8.1	9,445,556	100

Table 4.4: Results for structured problems from the SAPS and PAWS original studies, (the = symbol means that R+DDFW$^+$ behaves identically to R+DDFW on these problems)

We can further conclude that R+DDFW (i.e. even without the adaptive mechanism) has the better overall performance in comparison to AdaptNovelty$^+$, G^{2}WSAT and R+RSAPS. If we first look at R+G^{2}WSAT, while it performed well on the smaller random problems, it could not match R+DDFW on the larger more difficult random problems. In the other categories R+G^{2}WSAT was less competitive, again showing promise on the smaller structured problems in Table 4.4, but failing to scale up as well as R+DDFW on the more difficult problems. Interestingly, G^{2}WSAT performed strongly on the quasigroup problems when no resolution was performed, but was uncompetitive after resolution (these results are not reported in the current chapter). This confirms the findings in [1] that suggest clause weighting algorithms can gain more advantage from resolution than non-weighting algorithms. In addition, R+G^{2}WSAT was uniformly worse than R+DDFW on the ferry problems.

Turning our attention to R+RSAPS, this algorithm showed slightly better performance than R+DDFW on the structured and ferry problems, dominating on 10 of the 16 ferry problems and on all the parity problems, with R+DDFW showing the better performance on the remaining 6 ferry problems and on the other larger structured problems. However, R+RSAPS was outperformed by R+DDFW$^+$ on the parity problems, was uniformly worse on the random problems and was uncompetitive with R+DDFW on the quasigroup problems, thereby failing to show the same robust performance as R+DDFW and R+DDFW$^+$ across the whole range of problem sets. Our third comparison algorithm, R+AdaptNovelty$^+$, also had the worst overall performance, being unable to achieve outright dominance on any of the problems considered.

In a further unpublished study (not reported here) we investigated the effect of the preprocessing resolution step on the performance of each algorithm. This showed that resolution has little

effect on the random problem instances but has a positive effect on the quasigroup instances, with the effect being more pronounced for R+DDFW and less pronounced for R+G^{2}WSAT. For the real world instances, resolution was also generally helpful for the ferry, ais, logistics and parity problems but had little or no effect on the bw and flat problems.

4.6 Summary

In summary, we have introduced and integrated a new adaptive mechanism into the DDFW algorithm. This mechanism is unusual in that it oscillates between increasing and resetting clause weights, timing these changes according to a stagnation measure defined by the number of problem literals. While the increase mechanism increments the existing weight profile, the reset mechanism eliminates the profile entirely, returning the weights to their initial state. The reset mechanism also ensures that the amount of weight added to a problem is strictly controlled without requiring an additional weight decrease parameter.

In order to evaluate the new adaptive algorithm, R+DDFW$^+$, we also incorporated the latest resolution-based preprocessing technique used by the winning algorithm in the SAT2005 competition. In a broad ranging empirical study we have shown that integrating our new adaptive mechanism into DDFW can significantly enhance its overall performance. We have also shown that R+DDFW$^+$ has the best overall performance across a range of representative structured and random problem instances in comparison to three of the best SLS solvers currently available. The results suggest that R+DDFW$^+$ should be the SLS algorithm of choice in situations where the characteristics of a problem domain are not known in advance and manual parameter tuning is not practical.

Chapter 5

Handling Over-Constrained SAT Problems

5.1 Introduction

Many real-world problems are *over-constrained.* and require search techniques adapted to optimising cost functions rather than searching for consistency. This makes the MAX-SAT problem an important area of research for the satisfiability (SAT) community. In this chapter we perform an empirical analysis of several of the best performing SAT local search techniques in the domain of unweighted MAX-SAT. In particular, we test the two SAT clause weight redistribution algorithms developed in the previous chapters, DDFW and DDFW$^+$, against three more well-known techniques (RSAPS, AdaptNovelty$^+$ and PAWS). In addition, we look at using the DDFW heuristic to optimise over-constrained problems with hard and soft constraints and develop a new DDFW-O algorithm. We compare this algorithm with the *TWO-LEVEL* local search heuristic also developed specifically for hard and soft constraint problems.

5.2 Unweighted MAX-SAT Problems

Since the development of GSAT (see Chapter 2), there has been considerable interest and progress in developing stochastic local search (SLS) techniques for solving propositional satisfiability (SAT) problems. Because of the fact that many, if not most, real-world problems are over-constrained, if we express such problems using the SAT formalism, then the search task is no longer to find whether a satisfying assignment exists, but to find an assignment that maximises the number of true clauses (or equivalently minimises the number of false clauses). This problem is known as the unweighted MAX-SAT problem and has many applications in such areas as scheduling and the processing of Bayesian networks [81]. An extension of the MAX-SAT for-

malism, *weighted* MAX-SAT, allows that each clause c_i in the problem is assigned a weight w_i, with the objective of maximising the sum of the weights of all satisfied clauses. The unweighted MAX-SAT problem is therefore a special case of the weighted problem where each clause has a weight of one. Weighted MAX-SAT allows the expression of preferences over which clauses should be considered more important and thereby captures an additional feature of many real-world problem situations. However, an unweighted MAX-SAT can also represent weights by adding duplicate clauses to the problem (for example, see [10]).

5.3 Local Search for Unweighted MAX-SAT

The basic approach of a local search is to try and improve on an existing problem solution by making small changes selected from a local neighbourhood of possible moves. In the case of MAX-SAT, a problem consists of a set of clauses, each containing of a set of variables that can be instantiated as either true or false. In turn, the variable values within the logical structure of a clause determine the overall truth or falsity of the clause. To measure how good a particular solution is, we simply sum together the number of false clauses - the fewer false clauses, the better the solution.

SLS techniques are particularly suited to the MAX-SAT problem because, unlike complete search methods, they do not rely propagation techniques to prune the search space. Instead, to handle MAX-SAT, an SLS simply needs a modification so that it records the best solution encountered in the search so far. The disadvantage of SLS (as discussed in Chapter 2) is that there is no method to identify whether a solution is *optimal* (i.e. in terms of minimising the number of false clauses). Therefore, the usual strategy is to allow an SLS technique to time-out and report its best solution. If optimality is an important criteria then a complete search is required. However, for many large and complex problems, complete search techniques are ineffective, and SLS becomes the only practical alternative.

During the 1990s, a number of SLS techniques were applied to the MAX-SAT problem, including the Steepest Ascent Mildest Descend (SAMD) [30], Iterated Local Search (ILS) [94] and Guided Local Search (GLSSAT2) [55]. More recently, the Scaling and Probabilistic Smoothing (SAPS) algorithm (discussed in Chapter 2) was successfully applied to the unweighted MAX-SAT domain and shown to perform better than two of the previously best known algorithms, GLSSAT2 and ILS-YI [81].

SAPS relies on the setting of three sensitive parameters to obtain optimal performance [77].

This makes SAPS impractical in many real-world situations as manual parameter tuning can be a time-consuming process.[1] In addition, recent SAT research has concentrated on devising efficient algorithms for the SAT conference competitions where manual parameter tuning is not allowed.[2] For these situations a range of SLS algorithms have been engineered that are either less sensitive to problem dependent parameter settings or have a mechanism whereby a parameter can be automatically adapted during the search process. In particular, an adaptive version of SAPS, known as reactive or RSAPS [41], has been developed, based on earlier work that produced an adaptive mechanism for the WalkSAT family of SLS algorithms [36]. To date, the best performing SLS algorithms in the SAT competition have been based around this adaptive WalkSAT framework and the Novelty search heuristic (e.g. AdaptNovelty$^+$).

In the current chapter we extend the earlier work on SAPS to examine the performance of fixed parameter and adaptive SLS algorithms in the MAX-SAT domain. The motivation is to discover which of the current SLS techniques is the most practical for MAX-SAT when parameter tuning is ruled out. In particular, we are interested in evaluating DDFW in an over-constrained problem setting. As the previous chapters have demonstrated, DDFW has already proved competitive with the state-of-the-art in the SAT domain using both fixed parameter and adaptive mechanisms, but has yet to be applied to the MAX-SAT problem. It is therefore an open research question whether DDFW's promising performance can be maintained and whether its recently developed adaptive mechanism remains effective for over-constrained problems.

5.3.1 Divide and Distribute Fixed Weight for Unweighted MAX-SAT

DDFW and DDFW$^+$ have already been introduced in Chapters 3 and 4. In the current chapter, we additionally changed DDFW so that it keeps track of the current best cost and the current best solution (see Algorithm 10).[3]

5.4 MAX-SAT Experimental Study

Initial empirical evaluations of DDFW and DDFW$^+$ on a range of structured and unstructured SAT problems have shown both algorithms can outperform AdaptNovelty$^+$ and RSAPS, and that DDFW$^+$ has a better average performance than DDFW [42]. However, a similar study has yet

[1] More recently, machine learning techniques trained on example instances and then applied to unseen instances have proved useful for setting SAPS parameters [40]

[2] See http://www.satcompetition.org

[3] The same cost recording changes were made to all the SAT algorithms appearing in the study.

Algorithm 10 DDFW-MAX-SAT($\mathcal{F}$, $MaxTime$, W_{init}, C)

```
1:  A ← a randomly generated truth assignment of F
2:  set the weight w_i for each clause c_i ∈ F to W_init
3:  set C_best ← number of false clauses in A and A_best ← A
4:  while time < MaxTime do
5:    if C_best ≤ C then
6:      return A_best
7:    else
8:      select list L of literals causing the greatest reduction in weighted cost Δw when flipped
9:      if (Δw < 0) or (Δw = 0 and probability ≤ 15%) then
10:       update A by flipping a literal randomly selected from L
11:       if (number of false clauses in A < C_best) then
12:         C_best ← number of false clauses in A and A_best ← A
13:       end if
14:     else
15:       for each false clause c_f do
16:         select a satisfied same sign neighbouring clause c_k with maximum weight w_k
17:         if w_k < W_init then
18:           randomly select a clause c_k with weight w_k ≥ W_init
19:         end if
20:         if w_k > W_init then
21:           transfer a weight of 2 from c_k to c_f
22:         else
23:           transfer a weight of 1 from c_k to c_f
24:         end if
25:       end for
26:     end if
27:   end if
28: end while
29: return A_best
```

to be attempted in the MAX-SAT domain. To this end we decided to compare both DDFW variants with AdaptNovelty$^+$, RSAPS and PAWS10 (PAWS10 is a fixed parameter version of PAWS with the Max_{inc} parameter set at 10). Our first criteria was that manual parameter tuning is not allowed. This practical consideration means that the best clause weighting algorithms must be limited to using fixed parameter values (as in PAWS10) or to using an adaptive mechanism (as in RSAPS). We further chose AdaptNovelty$^+$ as our example WalkSAT algorithm because it has performed consistently well in the last three SAT competitions.

To evaluate the relative performance of these algorithms we followed the strategy used to compare SAPS with iterated local search and GLSSAT2 in [81] and divided our empirical study into three problem categories:

bor problem set: the bor-*k*u problem set comprises of a range of unweighted unsatisfiable MAX-

2-SAT (bor-2) or MAX-3-SAT (bor-3) instances used in [81][4] and first described by Borcher in [7]. We used all 17 bor-2 and 11 bor-3 unsatisfiable unweighted instances from this set.

jnh problem set: the jnh problems are generated with n variables. Each variable is added to a clause with probability $1/n$ and each literal is negated with probability 1/2 (unit and empty clauses are then deleted). We used 33 unsatisfiable jnh instances taken from the SATLIB library (www.satlib.org) each with 100 variables and between 800 and 900 clauses.

***uuf problem set*:** the uuf problems are unsatisfiable uniform random 3-SAT instances generated with n variables and k clauses with each clause containing 3 literals randomly chosen from the $2n$ possible literals. Tautological and duplicate literals are eliminated. We used the same 49 unsatisfiable instances reported in [81] each containing 200 variables and 1000 clauses.

Overall, the problem set is designed to show how DDFW and DDFW$^+$ compare to RSAPS, PAWS10 and AdaptNovelty$^+$ and to examine the relative effect of the adaptive mechanism on differing problem classes. All experiments were performed on a Dell machine with 3.1GHz CPU and 1GB memory. Cut-offs for the various algorithms were set as follows: first RSAPS and DDFW were given a trial on each problem with a flip cut-off of 10,000,000. The best cost reached by either algorithm was recorded and used as a cut-off for further runs. All algorithms were then allowed 100 trials for each instance and terminated either upon reaching the best cost (C in Algorithm 10) or after 3 seconds per trial. The graphs 5.1, 5.2, 5.3 and 5.4 summarise these results by detailing the proportion of problems for which the best cost had been found against CPU time in seconds (assuming each instance was run in parallel). Consequently, the higher the proportion of optimal solutions found at any given time, the better the overall performance of the algorithm.

5.4.1 *bor* Problem Results

Taken in combination, the graphs in Figures 5.1 and 5.2 and the results in Tables 5.1 and 5.2 show DDFW to have the best overall performance on both the bor-2 and bor-3 problem classes. In the bor-2 graph (Figure 5.1) DDFW and RSAPS clearly outperform the other techniques with DDFW having a slight advantage. The bor-2 results in Table 5.1 help to separate DDFW and RSAPS further by showing DDFW not only to be twice as fast as RSAPS on average, but also to be consistently better than RSAPS on a problem by problem basis (this is also illustrated by

[4] With thanks to Dave Tompkins who supplied us with these problems.

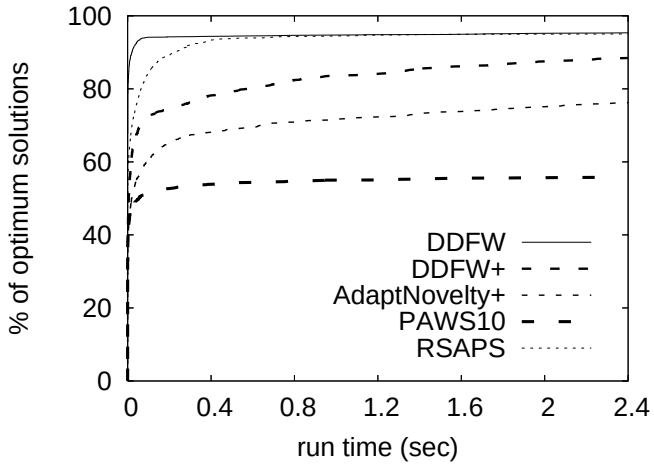


Figure 5.1: Graph of the comparative time performance for the bor-2 problems.

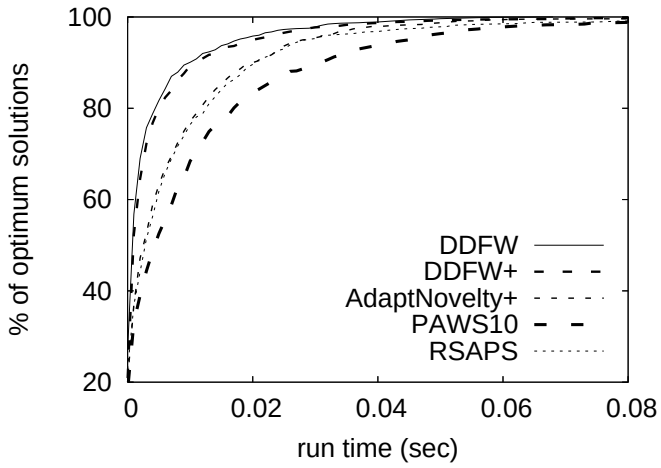


Figure 5.2: Graph of the comparative time performance for the bor-3 problems.

the 88.24% Winner value in the averaged results of Table 5.5). Interestingly, DDFW$^+$ performed quite poorly on this problem class, indicating there may be something in the two-literal clause structure of the bor-2 problems that causes the heuristic to be counter-productive.

The bor-3 graph results in Figure 5.2 are less conclusive, as all algorithms were able to achieve near 100% success, although DDFW and DDFW$^+$ both exhibit superior performance during the first 0.04 seconds. Again, the table results help to clarify that DDFW is the better algorithm, showing it to have the best performance on 68.68% of instances, with DDFW$^+$ coming a close second in terms of average time (see Table 5.5).

		DDFW			DDFW$^+$			AdaptNovelty$^+$			RSAPS			Paws10		
Problems	BF	Time	Flips	%	Time	Flips	%	Time	Flips	%	Time	Flips	%	Time	Flips	%
bor2-100.200	5	0.52	350	100	0.82	599	100	0.38	366	100	**0.37**	**355**	**100**	0.39	398	100
bor2-100.300	15	**0.21**	**153**	**100**	4.66	2,921	100	1.77	1,529	100	1.44	1,116	100	65.29	48,492	98
bor2-100.400	29	**27.30**	**12,601**	**100**	1,075	584,108	20	1,002	805,405	48	202.00	136,624	100	49.00	30,190	1
bor2-100.500	44	**4.48**	**1,707**	**100**	928.00	453,730	89	1,424	739,059	25	24.80	15,303	100	233.00	106,115	8
bor2-100.600	65	**0.98**	**407**	**100**	554.00	246,368	97	1,312	900,946	6	27.70	15,548	100	598.00	231,618	10
bor2-150.300	4	**0.09**	**140**	**100**	0.11	148	100	0.23	220	100	0.15	161	100	0.18	203	100
bor2-150.450	22	**11.40**	**5,569**	**100**	637.00	369,179	99	56.60	46,809	100	114.00	81,353	100	239.00	155,205	6
bor2-150.600	38	**1,149**	**491,330**	**21**	2,242	1,037,212	1	1,633	1,257,381	1	1,023	657,199	16	*n/a*	*n/a*	0
bor2-050.100	4	0.00	45	100	0.00	38	100	0.00	52	100	**0.00**	**35**	**100**	0.00	42	100
bor2-050.150	8	**0.02**	**48**	**100**	0.03	60	100	0.19	200	100	0.07	80	100	0.06	103	100
bor2-050.200	16	**0.38**	**234**	**100**	9.61	5,751	100	39.40	32,727	100	1.82	1388	100	124.00	89,789	96
bor2-050.250	22	**0.03**	**51**	**100**	6.29	3,452	100	8.75	6,889	100	0.49	356	100	16.50	10,914	100
bor2-050.300	32	**0.01**	**33**	**100**	1.47	738	100	8.24	6,183	99	0.15	129	100	24.72	13,627	98
bor2-050.350	41	**0.64**	**262**	**100**	21.30	9,541	100	154.00	108,901	99	113.00	68,664	100	24.62	12,459	8
bor2-050.400	45	**0.07**	**61**	**100**	3.67	1,506	100	92.90	62,021	100	6.53	3,698	100	114.00	50,658	68
bor2-050.450	63	**0.51**	**194**	**100**	62.20	23,414	100	1,055	663,772	52	31.30	16,526	100	132.00	49,804	14
bor2-050.500	66	**0.91**	**94**	**100**	19.70	7,055	100	786.00	483,250	83	42.30	21,239	100	132.00	46,935	34

Table 5.1: Results for the bor-2 problems.

		DDFW			DDFW$^+$			AdaptNovelty$^+$			RSAPS			Paws10		
Problems	BF	Time	Flips	%	Time	Flips	%	Time	Flips	%	Time	Flips	%	Time	Flips	%
bor3-100.500	4	2.93	1,450	100	**2.51**	**1,237**	**100**	3.99	2,619	100	10.80	6,862	100	11.70	7,156	100
bor3-100.550	5	**4.33**	**2,054**	**100**	4.61	2,266	100	7.05	4,458	100	7.75	4,769	100	11.10	6,054	100
bor3-100.600	8	**3.99**	**1,714**	**100**	7.79	3,604	100	16.44	10,086	99	11.20	6,524	100	16.30	9,104	100
bor3-150.675	2	14.20	7,618	100	**9.69**	**5,272**	**100**	14.50	9,824	100	15.70	10,295	100	13.10	8,310	100
bor3-150.750	5	12.60	5,996	100	17.30	8,452	100	**12.50**	**7,902**	**100**	25.00	15,386	100	33.60	19,436	100
bor3-50.250	2	0.72	422	100	0.69	406	100	0.60	434	100	**0.48**	**373**	**100**	0.67	417	100
bor3-50.300	4	**0.68**	**359**	**100**	1.06	542	100	1.66	1,055	100	0.76	516	100	0.82	504	100
bor3-50.350	8	**1.37**	**565**	**100**	1.56	704	100	4.60	2,795	100	4.74	2,557	100	5.29	2,980	100
bor3-50.400	11	**0.84**	**311**	**100**	1.58	638	100	8.01	4,611	100	4.14	2,192	100	6.06	3,157	100
bor3-50.450	15	**0.51**	**188**	**100**	1.11	413	100	8.25	4,451	100	4.78	2,403	100	12.60	6,088	100
bor3-50.500	15	**0.24**	**107**	**100**	0.40	159	99	1.31	685	99	1.83	892	99	8.58	3,968	99

Table 5.2: Results for the bor-3 problems.

5.4.2 *jnh* Problem Results

The graph in Figure 5.3 and the results in Table 5.3 show a mixed picture for the jnh problems. In the graph, AdaptNovelty$^+$ and DDFW$^+$ emerge as equal best performers in the first 0.1 seconds of execution but are joined by DDFW after 0.2 seconds, with PAWS10 starting well but trailing off and RSAPS coming last. In Table 5.5, both AdaptNovelty$^+$ and DDFW$^+$ achieve similar results, with AdaptNovelty$^+$ having a slight advantage in average time, but DDFW$^+$ doing better on flips and the % Winner value. The surprise is that PAWS10 achieves the best result on 48.48% of jnh instances, despite being beaten in terms of average time. This can be explained by PAWS10 taking a longer than average on the more difficult jnh instances. Overall, therefore, there is little to chose between DDFW$^+$ and AdaptNovelty$^+$ on these problems, although they both do better than DDFW and PAWS10, with RSAPS coming in last.

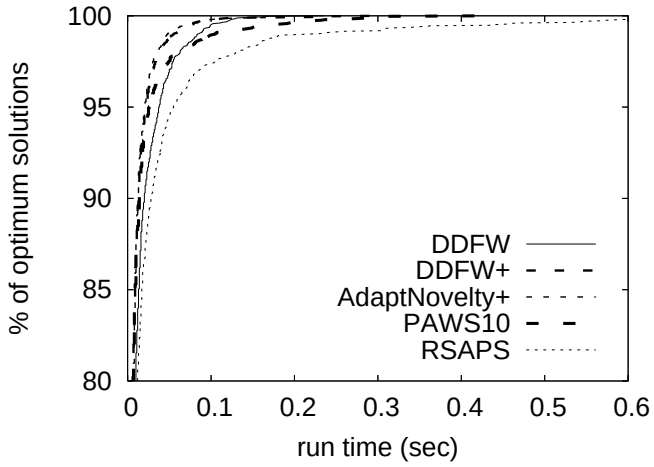


Figure 5.3: Graph of the comparative time performance for the jnh problems.

		DDFW			DDFW$^+$			AdaptNovelty$^+$			RSAPS			Paws10		
Problems	BF	Time	Flips	%	Time	Flips	%	Time	Flips	%	Time	Flips	%	Time	Flips	%
jnh-80-202	1	6.18	289	100	2.34	1,079	100	2.23	1,227	99	**1.38**	**741**	**100**	1.76	957	100
jnh-80-203	1	11.85	5,950	100	**5.18**	**2,446**	**100**	6.05	3,324	100	25.22	14,155	100	7.28	4,082	100
jnh-80-206	1	1.44	695	100	1.17	505	100	1.85	987	100	0.96	500	100	**0.84**	**433**	**100**
jnh-80-208	1	5.31	2,426	100	4.05	1,819	100	4.72	2,625	100	4.18	2,358	100	**3.51**	**1,898**	**100**
jnh-80-211	2	5.05	1,970	100	2.90	1,272	100	3.83	2,113	100	3.51	1,921	100	**2.61**	**1,411**	**100**
jnh-80-214	1	12.30	6,375	100	3.50	1,789	100	2.51	1,378	100	2.79	1,546	100	**1.74**	**936**	**100**
jnh-80-215	1	2.60	1,210	100	1.68	762	100	1.59	911	100	4.49	2,523	100	**1.58**	**881**	**100**
jnh-80-216	1	6.90	3,687	100	4.81	2,639	100	2.66	1,456	100	2.31	1,273	100	**1.87**	**1,008**	**100**
jnh-80-219	1	12.40	5,591	100	7.43	3,375	100	7.26	3,974	100	9.07	4,986	100	**7.20**	**3,955**	**100**
jnh-85-010	1	31.97	14,486	100	11.00	4,910	100	**6.02**	**3,197**	**100**	6.46	3,467	100	6.06	3,253	100
jnh-85-011	1	4.71	2,155	100	3.47	1,542	100	3.17	1,687	100	3.22	1,737	100	**2.54**	**1,370**	**100**
jnh-85-013	2	2.55	1,063	100	1.65	648	100	2.55	1,374	100	2.56	1,342	100	**1.64**	**801**	**100**
jnh-85-014	2	5.68	2,535	100	3.70	1,527	100	**3.20**	**1,691**	**100**	10.59	5,685	100	5.00	2,650	100
jnh-85-015	2	**2.95**	**1,227**	**100**	3.25	1,378	100	3.96	2,076	100	5.83	3,129	100	4.09	2,155	100
jnh-85-016	1	2.12	991	100	1.12	502	100	1.41	735	100	0.98	523	100	**0.80**	**377**	**100**
jnh-85-018	1	25.22	12,708	100	14.46	6,809	100	18.17	9,794	100	19.28	9,975	100	**11.71**	**6,323**	**100**
jnh-85-019	2	3.36	1,456	100	**2.22**	**937**	**100**	4.06	2,172	100	4.08	2,174	100	2.94	1,538	100
jnh-85-002	1	5.12	2,586	100	1.74	762	100	2.99	1,601	100	2.19	1,196	100	**1.33**	**684**	**100**
jnh-85-003	2	1.46	565	100	1.12	454	100	1.58	819	99	**0.91**	**476**	**100**	1.05	547	100
jnh-85-004	1	8.95	3,888	100	5.31	2,386	100	12.46	6,381	100	10.28	5,615	100	**4.81**	**2,600**	**100**
jnh-85-005	1	2.84	1,184	100	2.06	848	100	4.36	2,286	100	1.99	1,011	100	**1.64**	**820**	**100**
jnh-85-006	1	4.02	1,785	100	2.75	1,193	100	3.97	2,119	100	2.51	1,363	100	**2.50**	**1,307**	**100**
jnh-85-008	2	20.72	8,827	100	19.56	8,359	100	**7.89**	**4,203**	**100**	44.05	23,282	100	28.80	14,414	100
jnh-85-009	2	10.51	4,563	100	11.23	4,889	100	**10.28**	**5,542**	**100**	26.88	14,439	100	14.22	7,422	100
jnh-90-302	4	**1.51**	**575**	**100**	1.66	660	100	2.01	1,027	100	2.52	1,272	100	2.06	1,019	100
jnh-90-303	3	2.94	1,120	100	**2.71**	**1,074**	**100**	5.71	2,946	100	5.36	2,708	100	4.25	2,144	100
jnh-90-304	3	1.11	434	100	**0.91**	**349**	**100**	1.25	645	100	1.60	793	100	0.99	474	100
jnh-90-305	3	48.23	18,844	100	**43.04**	**17,001**	**100**	28.38	14,217	99	221.00	111,727	100	91.45	45,597	100
jnh-90-306	1	1.98	870	100	1.33	529	100	1.89	982	100	1.11	590	100	**1.00**	**502**	**100**
jnh-90-307	3	**4.95**	**1,818**	**100**	6.34	2,529	100	7.08	3,675	100	37.45	19,259	100	10.02	5,127	100
jnh-90-308	2	2.99	1,224	100	**1.98**	**778**	**100**	2.75	1,393	100	4.22	2,139	100	2.95	1,505	100
jnh-90-309	2	1.06	393	100	0.86	331	100	1.27	643	100	0.85	430	100	**0.72**	**344**	**100**
jnh-90-310	3	10.43	4,260	100	**5.85**	**2,329**	**100**	10.55	5,426	100	21.09	10,549	100	10.35	5,095	100

Table 5.3: Results for the jnh problems.

5.4.3 *uuf* Problem Results

The uuf results are somewhat similar to the bor-2 results, only this time DDFW has a very similar curve in Figure 5.4 to AdaptNovelty$^+$ (rather than RSAPS). In this case DDFW starts well but is caught by AdaptNovelty$^+$ after 0.5 seconds, after which AdaptNovelty$^+$ has a slight advantage. This is confirmed by the slightly better uuf success rate for AdaptNovelty$^+$ in Table 5.5. However, the table also shows DDFW to have the better average time and to have achieved the best result on 67.35% of the instances. Taking this into consideration, we can conclude that DDFW has the better uuf performance, with AdaptNovelty$^+$ coming a close second, followed by DDFW$^+$ (Table 5.4 shows the relative performance of the five algorithms before averaging).

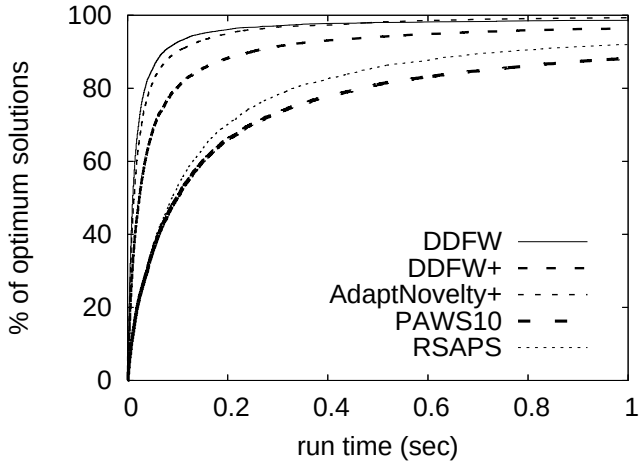


Figure 5.4: Graphs of the comparative time performance for the uuf problems.

5.5 MAX-SAT Analysis

Overall, the results show that DDFW has the best overall performance on our chosen problem set. More specifically, DDFW has the better performance on three out of four of the problem classes, emerging as a clear winner on the bor-2 and bor-3 problem sets, slightly better than AdaptNovelty$^+$ on the uuf problems but defeated by AdaptNovelty$^+$ and DDFW$^+$ on the jnh problems. DDFW's closest rivals are AdaptNovelty$^+$ and DDFW$^+$, with AdaptNovelty$^+$ defeating DDFW$^+$ on the uuf problems, equalling it on the jnh problems and being defeated by DDFW$^+$ on the bor-2 and bor-3 problems.

Both RSAPS and PAWS10 can be ruled out of consideration as they have performed consistently badly on all problem classes with the exception of PAW10's promising early start on the

		DDFW			DDFW$^+$			AdaptNovelty$^+$			RSAPS			Paws10		
Problems	BF	Time	Flips	%	Time	Flips	%	Time	Flips	%	Time	Flips	%	Time	Flips	%
uuf-01	3	14.12	4,863	100	15.30	7,672	100	**12.98**	**8,667**	**100**	87.01	50,977	100	69.49	40,104	100
uuf-02	7	**6.87**	**3,144**	**100**	12.64	6,044	100	9.50	6,153	100	39.65	22,905	100	46.76	25,472	100
uuf-03	6	**19.95**	**9,263**	**100**	55.28	26,892	100	35.23	22,764	100	216.00	125,564	99	266.00	141,636	100
uuf-04	4	**12.02**	**5,910**	**100**	26.43	13,007	100	17.29	11,240	99	178.00	106,689	100	121.00	69,582	100
uuf-05	7	**13.22**	**6,188**	**100**	43.60	21,164	100	20.47	13,362	100	118.00	68,433	100	172.00	96,996	100
uuf-06	4	**87.88**	**43,075**	**100**	268.00	132,830	98	424.00	279,363	100	469.00	276,730	89	827.00	474,631	99
uuf-07	7	55.13	25,908	100	290.00	138,615	99	**21.96**	**14,367**	**100**	422.00	244,504	90	1,010	564,849	95
uuf-08	7	58.48	27,089	100	369.00	176,132	95	**65.29**	**42,628**	**100**	595.00	341,005	69	1,832	1,009,036	82
uuf-09	5	**109.00**	**52,106**	**100**	302.00	144,751	99	112.00	74,760	100	643.00	378,478	61	1,501	853,024	80
uuf-10	6	**10.23**	**4,765**	**100**	30.78	14,925	100	10.21	6,728	99	171.00	98,808	100	204.00	113,539	100
uuf-11	7	**31.28**	**14,296**	**100**	81.22	38,506	100	46.12	30,195	100	106.00	61,243	100	166.00	91,998	100
uuf-12	6	11.66	5,507	100	39.70	19,319	100	**9.30**	**6,000**	**100**	134.00	78,701	100	119.00	67,044	100
uuf-13	6	**19.81**	**9,300**	**100**	57.79	27,978	100	26.79	17,468	100	172.00	100,270	100	240.00	134,001	100
uuf-14	6	**4.24**	**1,935**	**100**	7.63	3,665	100	6.53	5,301	100	30.11	17,263	100	27.66	14,571	100
uuf-15	6	**6.11**	**2,823**	**100**	7.89	3,777	100	6.25	4,049	100	20.10	11,604	100	21.52	11,871	100
uuf-16	5	**6.58**	**3,122**	**100**	14.97	7,301	100	14.57	8,597	100	32.08	18,601	100	37.78	20,614	100
uuf-17	5	**11.73**	**5,627**	**100**	25.93	12,633	100	16.19	10,662	100	48.51	28,599	100	48.06	26,517	100
uuf-18	7	**31.06**	**14,055**	**100**	73.68	34,447	100	61.54	39,864	100	381.00	216,741	93	1,064	586,281	98
uuf-19	6	**18.85**	**8,961**	**100**	39.86	19,579	100	38.03	25,025	100	235.00	137,553	100	258.00	145,137	100
uuf-20	4	**12.44**	**6,053**	**100**	14.23	6,976	100	22.03	14,309	100	64.02	38,160	100	49.16	28,116	100
uuf-30	5	54.55	25,912	100	97.43	47,075	100	**32.89**	**21,615**	**100**	290.00	169,373	100	268.00	148,950	100
uuf-40	4	**21.56**	**10,512**	**100**	54.84	26,917	100	32.75	21,638	100	255.00	151,787	100	220.00	124,434	100
uuf-41	5	**17.39**	**8,422**	**100**	24.54	12,068	100	20.74	13,700	100	47.98	28,412	100	73.42	38,762	100
uuf-42	6	**41.67**	**19,483**	**100**	111.00	54,207	100	64.48	41,241	99	382.00	221,807	92	698.00	388,350	100
uuf-44	3	**20.49**	**10,459**	**100**	33.09	16,617	100	21.42	14,443	100	136.00	80,181	100	97.18	56,131	100
uuf-46	7	**32.77**	**15,066**	**100**	66.20	31,315	100	112.00	71,881	100	159.00	90,313	100	406.00	222,130	100
uuf-48	5	**6.53**	**3,102**	**100**	12.06	5,869	100	11.33	7,326	100	70.38	41,271	100	80.73	45,377	100
uuf-50	6	**8.41**	**3,946**	**100**	27.08	13,094	100	15.82	10,375	100	82.16	46,545	100	89.37	49,321	100
uuf-52	5	20.49	9,838	100	55.58	24,661	100	**19.16**	**11,748**	**100**	224.00	131,579	99	308.00	175,080	100
uuf-53	5	321.00	153,398	99	561.00	271,417	51	**398.00**	**2,608,861**	**100**	723.00	423,986	18	1,754	985,827	35
uuf-54	8	45.99	20,527	100	243.00	112,640	100	**35.39**	**22,953**	**100**	423.00	240,058	81	975.00	529,865	95
uuf-56	7	49.30	22,444	100	180.00	86,045	100	**31.60**	**20,654**	**100**	247.00	142,078	100	549.00	303,046	100
uuf-58	4	**4.33**	**2,043**	**100**	4.72	2,239	100	4.48	2,866	100	92.95	54,337	100	42.42	24,046	100
uuf-60	7	511.00	232,060	44	759.00	356,483	11	**338.00**	**222,298**	**100**	555.00	317,849	8	2,536	1,391,041	9
uuf-64	4	14.53	7,036	100	28.65	14,239	100	**11.96**	**7,883**	**100**	126.00	75,237	100	108.00	62,254	100
uuf-68	6	**12.54**	**5,922**	**100**	24.12	11,668	100	16.10	10,498	100	102.00	59,648	100	102.00	56,767	100
uuf-70	5	**3.40**	**1,550**	**100**	3.76	1,779	100	5.59	3,069	100	8.64	4,962	100	8.62	4,782	100
uuf-75	7	**10.66**	**4,808**	**100**	19.57	9,162	100	14.73	9,650	100	114.00	65,070	100	217.00	118,780	100
uuf-80	6	5.64	2,628	100	12.80	6,216	100	**5.53**	**3,506**	**100**	114.00	66,308	100	102.00	56,837	100
uuf-85	6	**13.22**	**6,367**	**100**	32.05	15,850	100	21.92	14,140	100	112.00	65,909	100	122.00	69,377	100
uuf-89	3	5.81	2,910	100	**5.52**	**2,779**	**100**	6.01	3,950	99	13.95	8,432	100	6.71	3,886	100
uuf-90	6	7.77	3,616	100	19.70	9,565	100	**7.58**	**4,935**	**100**	119.00	69,021	100	137.00	76,515	100
uuf-91	3	8.97	4,440	100	**5.34**	**2,596**	**100**	13.14	8,684	100	7.46	4,358	100	5.93	3,280	100
uuf-92	6	**20.19**	**9,445**	**100**	57.49	27,366	99	33.18	21,236	100	152.00	88,298	100	235.00	128,917	100
uuf-93	6	**6.53**	**3,056**	**100**	15.15	7,330	100	16.91	11,146	99	52.54	30,620	100	69.67	38,705	100
uuf-94	6	**45.11**	**21,526**	**100**	149.00	71,816	100	125.00	82,335	100	370.00	216,120	98	681.00	382,812	100
uuf-95	5	**10.84**	**5,132**	**100**	33.41	10,993	100	11.64	7,650	100	132.00	76,495	100	93.69	52,402	100
uuf-96	4	**14.55**	**7,130**	**100**	40.99	20,441	100	16.26	10,736	100	141.00	84,046	100	157.00	89,398	100
uuf-98	5	**9.75**	**4,321**	**100**	14.18	6,845	100	8.48	5,493	99	58.78	34,178	99	47.43	26,398	99

Table 5.4: Results for the uuf problems.

jnh problems.

In relation to the poor performance of DDFW on the jnh problems, we reason that this was related to the value of the fixed W_{init} parameter. DDFW$^+$ came equal first on this problem set and yet the small average flip rate (2,376) would have meant the $^+$ heuristic had little effect.

Problem	Method	Mean Flips	Mean Seconds	% Optimal	% Winner
bor-2	DDFW	**7,719.62**	**0.01777**	**95.41**	**88.24**
	DDFW$^+$	79,308.65	0.15745	88.59	0.00
	AdaptNovelty$^+$	125,590.40	0.19005	77.24	0.00
	RSAPS	28,943.91	0.04536	95.06	11.76
	PAWS10	28,688.04	0.05131	55.82	0.00
bor-3	DDFW	**1,890.02**	**0.00385**	**100.00**	**63.63**
	DDFW$^+$	2,156.29	0.00440	99.91	18.18
	AdaptNovelty$^+$	4,449.40	0.00717	99.73	9.09
	RSAPS	4,798.05	0.00793	99.91	9.09
	PAWS10	6,109.10	0.01093	99.91	0.00
jnh	DDFW	3,721.65	0.00833	100.00	6.06
	DDFW$^+$	**2,376.13**	0.00553	100.00	21.21
	AdaptNovelty$^+$	2,870.73	**0.00546**	100.00	18.18
	RSAPS	7,724.26	0.01489	100.00	6.06
	PAWS10	3,747.89	0.00732	100.00	**48.48**
uuf	DDFW	**15,453.06**	**0.03294**	98.80	**67.35**
	DDFW$^+$	35,161.40	0.07335	96.98	4.08
	AdaptNovelty$^+$	32,441.44	0.04965	**99.92**	28.57
	RSAPS	97,412.31	0.16765	93.80	0.00
	PAWS10	167,275.80	0.30023	95.76	0.00

Table 5.5: Average results for each problem set; the time and flip statistics were measured over runs that reached an optimal solution; % Optimal measures the proportion of runs that reached an optimal solution; In both cases an optimal solution refers to the best cost solution obtained using the procedure described in Section 5.4. % Winner measures the proportion of problems for which an algorithm obtained the best result, e.g. 88.24% for DDFW on bor-2 means DDFW had the best result for 15 of the 17 bor-2 instances.

Rather, DDFW$^+$'s superior performance is likely to be due to the lower amount of initial weight added to the clauses at the start of the search (2 as opposed to 8 for DDFW). Consequently, we would assume that a lower W_{init} would similarly improve the performance of DDFW on the jnh problems.

The results of our MAX-SAT study partly confirm the initial evaluation of DDFW and DDFW$^+$ in the SAT domain [42]. There, both algorithms were shown to have better performance in comparison to several other state-of-the-art SAT algorithms (including AdaptNovelty$^+$ and RSAPS) on a range of SAT competition benchmark problems. However, it was concluded for SAT that the DDFW$^+$ heuristic was a generally helpful addition to DDFW, causing superior performance in several problem categories while remaining neutral in the others. In contrast, our MAX-SAT study has found the DDFW$^+$ heuristic to have a negative effect on performance on the bor-2, bor-3 and uuf problems, only helping for the jnh problems. This brings the general applicability of the DDFW$^+$ heuristic under question. There is certainly no sign that it is useful for MAX-SAT problems and it could be that the promising results in the SAT domain were produced simply by a fortuitous selection of problems. We leave answering this question as a matter for further research.

5.6 Handling Overconstrained Problems with Hard and Soft Clauses

The main shortcoming of the MAX-SAT formalism is that it assumes that all constraints or clauses can be treated as having equal importance. One way to extend MAX-SAT is to add duplicate clauses to a problem. For example, if we add a copy of clause c_i to a problem then whenever c_i is false then c_i's copy is also necessarily false. This doubles the cost of falsifying c_i and the search will consequently favour satisfying other non-duplicated clauses. However, such a scheme requires the unnecessary overhead of maintaining data structures for the duplicate clauses. Instead, the weighted MAX-SAT formalism has been developed, where each clause is associated with a fixed initial penalty or weight that defines how important it is to satisfy that clause. Now, when calculating the cost of a solution, we simply take the sum of the false clause weights rather than counting the number of false clauses. Several local search methods have already been successfully applied to the weighted MAX-SAT problem [83].

However, many real world problems involve constraints that are not just important, but that *must* be satisfied in order for a valid solution to exist. For example, staff scheduling problems involve constraints about minimum safe staffing levels that are required by law. If such a problem contains additional constraints that may remain unsatisfied while still producing a workable solution (such as staff preferences about particular shifts or days off) then we have a problem with both *hard* (mandatory) and *soft* (preference) constraints.

Hard and soft constraints can be modelled in a weighted MAX-SAT problem by making sure that the weight on any one hard constraint is greater than the sum of weight on all soft clauses. However, this solution is not an ideal, as excessively high weights on hard constraints can cause local search performance to degrade [76]. Given the practical importance of hard and soft constraint problems there have been various attempts to model and solve problems in this area, as we describe below.

5.6.1 Solving Over-Constrained Problems

The work of Freuder and Wallace [20] on partial constraint satisfaction was important in opening up the area of representing and solving over-constrained problems within the constraint satisfaction community. As with MAX-SAT, the objective was to maximise the total number of satisfied constraints (instead of satisfying all the hard constraints and maximising the total number of satisfied soft constraints). In other work [8], the hard and soft distinction was extended to multiple levels in a constraint hierarchy. Here constraints are classified into levels according to their impor-

tance, where each higher level constraint is strictly more important than any group of lower level constraints. In [5], a more general semiring approach for representing over-constrained problems was proposed that is capable of modeling traditional CSPs, constraint hierarchies, fuzzy CSPs and probabilistic CSPs.

The various approaches used to represent over-constrained problems are independent of the algorithms used to solve them, and most of the early work concentrated on the use of systematic techniques [20]. However, more recent work in [78, 32] shows that local search techniques can also be successfully applied to this area. More specifically, the work of Thornton *et al.* in [78] shows that clause weighting can be successfully and efficiently applied to handling over-constrained problems with hard and soft clauses.

Clause Weighting for Problems with Hard and Soft Clauses

Applying a clause weighting algorithm to problems where constraints have different levels of importance produces additional difficulties, as such techniques need to distinguish between weight that is added during the search to escape local minima and the original clause weights that represent the relative importance of the constraints. The solution proposed in [78] involved treating the weight added during the search as a *multiplier* of the weight assigned at the beginning of the search. For instance, if a clause is given an initial (importance) weight of w and is later penalised during the search as a result of being false in a local minimum, then the multiplier for the clause is increased from 1 to 2 and the overall weight on the clause will increase from w to $2w$. As a result of this work, a new algorithm was developed known as *TWO-LEVEL* which combined a PAWS like additive weight heuristic for satisfying hard constraints (or clauses) with a tabu search phase for satisfying soft clauses. The performance of *TWO-LEVEL* was compared with Novelty$^+$, MAX-AGE and a Tabu search algorithm on a range of randomly generated over-constrained SAT problems with hard and soft clauses. Overall *TWO-LEVEL* was found to convincingly outperform the other algorithms on this test set. *TWO-LEVEL*'s superior performance was explained by the use of the following three main ideas:

- **The use of a Tabu list**: As the main aim in *TWO-LEVEL* is to first satisfy all the hard clauses and only then to search for an optimum soft clause cost solution, weights are only added to the hard clauses. This means the search has no mechanism to escape from local minima once all hard clauses are satisfied. To address this, the search uses a short Tabu list to keep track of moves when optimising the soft constraints which it then ignores once the

search moves to a hard constraint violating area of the search space.

- **The use of a dynamic *hardMultiplier* weight update scheme**: As previously discussed, distinguishing between hard and soft clauses in *TWO-LEVEL* is done by adding a weight penalty during the search to the hard clauses. However there is still the question of the amount of initial weight that should be given to the hard clauses (to distinguish them as more important than the soft clauses). *TWO-LEVEL* solves this problem by starting with a minimum *hardMultiplier* weight of 2 on each hard constraint and a weight of 1 on each soft constraint. After this, if the search reaches a local minimum where hard constraints are still unsatisfied then the *hardMultiplier* weight for all hard clauses is incremented by 1 (otherwise it is decremented by 1). In this way the relative importance of the hard clauses is raised until all hard clauses are satisfied. Then the search tries to optimise the soft clauses, during which time the *hardMultiplier* is reduced until the search again prefers to violate a hard constraint, and the cycle continues.

- **The use of an additive weighting scheme**: The ongoing penalty weights of the hard clauses is maintained during the search via a DLM/PAWS type additive weighting scheme. This scheme simply adds weights to the current false hard constraints at each of local minima and reduces the weight of all clauses by 1 after every 10 increases. The total weight on each clause is then calculated as the penalty weight $\times$ *hardMultiplier*.

Extending DDFW for Problems with Hard and Soft Clauses

Given the strength of *TWO-LEVEL* in comparison to Novelty$^+$ on hard and soft constraint problems in [78] and the promising performance of DDFW on the MAX-SAT domain, a natural next step was to look at applying DDFW to handle over-constrained problems with hard and soft clauses. However, the choice of how to implement DDFW given two classes of clause is not straightforward. We therefore tried a number of different heuristics, as follows:

1. Use the DDFW heuristic to transfer weight from hard satisfied clauses to soft unsatisfied clauses.

2. Use the DDFW heuristic to transfer weight between satisfied and unsatisfied hard clauses while hard clauses are unsatisfied, otherwise use the DDFW heuristic to transfer weight between satisfied and unsatisfied soft clauses.

3. Use the DDFW heuristic to transfer weight between satisfied and unsatisfied hard clauses while hard clauses are unsatisfied, otherwise use a greedy search to optimise the number of unsatisfied soft clauses.

4. Use the DDFW heuristic to transfer weight from satisfied to unsatisfied clauses regardless of whether the clauses are hard or soft.

5. Use the DDFW heuristic to transfer weight from hard satisfied clauses to soft unsatisfied clauses.

In addition, we experimented with the following features of the *TWO-LEVEL* algorithm:

- Use the *TWO-LEVEL hardMultiplier* to adjust the weights of the hard clauses.
- Use the *TWO-LEVEL* tabu heuristic to avoid cycles.

We evaluated these five DDFW heuristics in combination with the two *TWO-LEVEL* heuristics in a preliminary study on a reduced problem set. This was because the number of permutations of possible algorithms was too large to engage in a full study and because the great majority of the heuristic combinations performed so badly as to not warrant further investigation. From this study it emerged that the best performing algorithm used the third DDFW heuristic (i.e. use the DDFW heuristic to transfer weight between satisfied and unsatisfied hard clauses while hard clauses are unsatisfied, otherwise use a greedy search to optimise the number of unsatisfied soft clauses) in conjunction with the use of the *TWO-LEVEL hardMultiplier* to adjust the weights of the hard clauses.

This new heuristic, DDFW-O, is detailed in Algorithm 11. It starts by initialising all clauses to a W_{init} weight of 8 and setting the *hardMultiplier* to 1. It then proceeds to use the standard *TWO-LEVEL* cost function to calculate the current solution cost as the sum of the weights on the soft clauses plus the sum of the weights on the hard clauses multiplied by the *hardMultiplier*. This cost value is inserted into DDFW's normal flip selection heuristic (borrowed from PAWS) that randomly selects a move from a list of best cost improving flips, or, if no such flip exists, with a probability of 15% selects a flip that leaves the cost unchanged . In the event that no such flip is selected and given that hard clauses reamin unsatisfied, DDFW-O invokes the DDFW weight redistribution heuristic, but only redistributes between hard clauses. It then increments the *hardMultiplier*. This means the soft clause weights remain static throughout the search (just

Algorithm 11 DDFW-O($\mathcal{F}$, $MaxTime$, W_{init}, A_{best})

```
1:  A ← a randomly generated truth assignment of F
2:  set the weight w_i for each clause c_i ∈ F to W_init
3:  set C_best ← number of false clauses in A, A_best ← A and hardMultiplier ← 1
4:  while time < MaxTime do
5:    Δw_best ← ∞
6:    for each false clause c_f do
7:      for each literal l_j ∈ c_f do
8:        hChange ← change in weighted hard constraint cost from flipping l_j
9:        sChange ← change in weighted soft constraint cost from flipping l_j
10:       Δw ← (hardMultiplier × hChange) + sChange
11:       if Δw ≤ Δw_best then
12:         if Δw < Δw_best then
13:           L ← ∅ and Δw_best ← Δw
14:         end if
15:         L ← L ∪ l_j
16:       end if
17:     end for
18:   end for
19:   if (Δw < 0) or (Δw = 0 and probability ≤ 15%) then
20:     update A by flipping a literal randomly selected from L
21:     if (number of false clauses in A < C_best) then
22:       C_best ← number of false clauses in A and A_best ← A
23:     end if
24:   else
25:     for each hard false clause hc_f do
26:       select a satisfied same sign neighbouring clause c_k with maximum weight w_k
27:       if w_k < W_init then
28:         randomly select a clause c_k with weight w_k ≥ W_init
29:       end if
30:       if w_k > W_init then
31:         transfer a weight of 2 from c_k to c_f
32:       else
33:         transfer a weight of 1 from c_k to c_f
34:       end if
35:     end for
36:     if false hard clauses exist then
37:       ++hardMultiplier
38:     else if hardMultiplier > 1 then
39:       - -hardMultiplier
40:     end if
41:   end if
42: end while
```

as with *TWO-LEVEL*). In the event that no flip is selected *and* all the hard clauses are satisfied, then DDFW-O simply decrements the *hardMultiplier*. This changes the relative importance of

the hard clauses and ensures that eventually a move that falsifies a hard clause will be preferred (i.e. will produce a cost reducing flip).

5.6.2 Hard and Soft Clause Experimental Study

In order to evaluate DDFW-O we decided to compare it with *TWO-LEVEL* on the original problem set reported in the *TWO-LEVEL* study [78]. Our reasoning was that if DDFW-O was unable to reach an equivalent or superior performance to *TWO-LEVEL* then further evaluation would be unnecessary. To the best of our knowledge, *TWO-LEVEL* remains the best performing SLS algorithm on the problem class chosen in the original study.

Hard and Soft Clause Problem Generation

The *TWO-LEVEL* study problems were based on randomly generated 3-SAT problems that were originally designed to evaluate four problem dimensions: problem size, overall problem difficulty, the hard constraint problem difficulty and the ratio of hard to soft constraints. The use of randomly generated problems means that we cannot evaluate the effect of structure on algorithm performance. However, the advantage is that we can control average problem difficulty by varying the clause to variable ratio. In addition, using 3-SAT problems allows us to divide clauses into hard and soft constraints and still have a measure of the relative difficulty of the hard constraint sub-problem.

The *TWO-LEVEL* problem set consisted of three sets of randomly generated 3-SAT problems all sampled from the phase transition region (i.e. with a clause to variable ratio of 4.3). Set 1 consisted of problems with 400 variables, set 2 of problems with 800 variables and set 3 of problems with 1600 variables. For each problem, an extra clause was randomly generated for each clause in the original problem (e.g. for a 400 variable 1720 clause problem another 1720 clauses were added), resulting in a second problem set with a clause to variable ratio of 8.6. This problem set was further multiplied by dividing up the hard and soft constraints into two ratios of 50% and 75%, where an n% ratio means the first n% of the clauses in the original problem are defined as hard constraints (e.g. in a 400 variable 3440 clause problem, if n = 50, the first 50% of the 1720 clauses which made up original problem are defined as hard). Using these problem generation procedures we selected 2 problem classes at each of the 3 problem sizes, making 6 data sets containing a total of 30 individual problems. The data sets are identified using the format hxhns, where x specifies the number of variables and n specifies the ratio of hard constraints.

	DDFW-O				*TWO-LEVEL*			
Problems	Time (secs)	Flips	Best cost	% Found	Time (secs)	Flips	Best cost	% Found
h400h50-1	46.2	1,272,608	122	10	0.2	5,356	113	10
h400h50-2	118.0	5,582,921	122	10	10.8	396,468	105	20
h400h50-3	111.0	2,969,757	125	10	5.1	164,359	109	30
h400h50-4	140.4	4,178,019	120	20	1.2	44,020	104	10
h400h50-5	132.0	7,693,809	119	10	2.4	82,839	112	30
h400h75-1	102.0	1,198,201	159	10	0.6	27,671	118	100
h400h75-2	25.3	267,543	162	10	16.0	764,533	123	100
h400h75-3	45.8	593,077	151	10	15.8	760,454	121	90
h400h75-4	52.7	654,279	154	10	5.5	258,554	118	100
h400h75-5	40.2	497,050	154	10	6.7	344,686	118	100
h800h50-1	84.5	1,995,405	243	10	73.6	1,701,174	203	20
h800h50-2	74.2	2,098,698	261	10	32.3	643,890	207	20
h800h50-3	40.8	355,697	244	10	61.8	350,544	200	20
h800h50-4	115.0	427,510	252	10	70.2	1,409,439	196	10
h800h75-1	41.1	354,305	282	10	41.3	1,366,984	223	60
h800h75-2	27.0	338,025	321	10	70.0	1,852,300	236	30
h800h75-3	123.0	2,926,387	298	10	86.2	2,377,644	233	20
h800h75-4	28.6	204,885	307	10	36.4	1,077,410	230	60
h1600h50-1	88.5	1,150,793	582	20	0.4	1,544	662	10
h1600h50-2	176.0	2,514,248	587	10	0.3	1,032	640	10
h1600h50-3	171.0	2,627,160	571	10	5.2	55,762	451	10
h1600h50-4	0.2	950	634	10	0.6	1,994	634	10
h1600h50-5	101.0	1,410,102	527	10	11.0	101,852	448	10
h1600h50-6	93.5	913,350	586	10	0.9	2,833	575	10
h1600h75-1	2.1	3,645	1479	10	4.1	7,607	1299	10
h1600h75-2	6.2	133,081	1933	10	0.4	1,307	1486	10
h1600h75-3	5.3	134,554	1587	10	4.3	7,739	1302	10
h1600h75-4	17.2	133,060	1545	10	0.3	1,255	1514	10
h1600h75-5	10.3	19,904	1487	10	1.4	2,581	1390	10
h1600h75-6	0.6	2,775	1435	10	2.1	3,269	1408	10

Table 5.6: Average performance of DDFW-O and *TWO-LEVEL* on the random hard and soft clause problem set.

Hard and Soft Clause Results and Analysis

The results in Table 5.6 show the averages for ten runs on each problem by each of the two algorithms. All experiments were performed on a Dell machine with a 3.1GHz CPU and 1GB memory running Linux and all runs had a flip cut-off of 10 million. As both DDFW-O and *TWO-LEVEL* were able to find hard clause satisfying solutions for all runs, we do not report the success rates of the algorithms. Instead we report statistics for those runs that found the best cost for each set of ten runs. For example, in row 1 of the results table, the best cost value for DDFW is 122. This means that of all ten runs of DDFW on the h400h50-1 problem, the best solution found that was able to satisfy all hard clauses still left 122 soft clauses unsatisfied. The % found result of 10 in the same row means that only one of the ten DDFW-O runs was able to find a best cost solution of 122. The time and flip values indicate the average elapsed execution time and

number of flips that had occurred when the best cost solution was found. If we consider the row 6 results of *TWO-LEVEL* for the h400h75-1 problem, the % found value of 100 means that all ten runs of *TWO-LEVEL* found a minimum cost of 118 and the average time of these runs to find a 118 solution was 0.6 seconds (27,671 flips). The consistency and speed of these runs strongly indicates that 118 is an optimal cost solution for this problem and that the problem itself is not that challenging (at least for *TWO-LEVEL*).

Overall, the results demonstrate that DDFW-O was unable to match the performance of *TWO-LEVEL*. This is shown by *TWO-LEVEL* having a superior best cost result for 27 of the 30 problems (DDFW-O was better on h1600h50-1 and h1600h50-2 and equal best on h1600h50-4). The overall best cost for the table makes this even clearer with *TWO-LEVEL* reaching an average best cost of 476 and DDFW-O reaching an average best cost of 547.

We should also note that the best cost scores in the Table 5.6 are the determining statistic in comparing the two algorithms. Unlike tables that compare performance on under-constrained problems, better average time and flip values do not imply better performance. For example, the results for h800h75-2 show that DDFW-O has the better time performance of 27.0 seconds (and 338,025 flips) compared to *TWO-LEVEL*'s 70.0 seconds and 1,852,300 flips. But this simply means that DDFW-O happened to find its best cost solution of 321 in one run of 27 seconds, despite the fact that it had 10 million flips per run to find a better solution. Conversely, *TWO-LEVEL* was able to find a considerably better cost solution of 236 during 3 different runs, making it a clear winner on this problem.

The overall results should also be interpreted in the light of the original *TWO-LEVEL* study [78]. There *TWO-LEVEL* was compared to a tabu search, Novelty$^+$ and MAX-AGE algorithms.[5] These algorithms were augmented to use the *TWO-LEVEL hardMultiplier*, but otherwise left unchanged. The original results for the problem classes used in the current study show that the tabu search was unable to reach 100% success in finding hard clause satisfying solutions. Of the remaining two techniques, Novelty$^+$ was able to achieve a best cost ratio of 170% compared to *TWO-LEVEL* and MAX-AGE achieved 139% (here we define the best cost ratio as the average best cost achieved by algorithm x divided by the average best cost achieved by *TWO-LEVEL* multiplied by 100). The best cost ratio for DDFW-O on the current results is $(547 \div 476) \times 100 = 115\%$. This tells us that although DDFW-O does not reach *TWO-LEVEL*'s best cost levels it has done better than Novelty$^+$, MAX-AGE or a simple tabu search on these problems. This at least

[5] MAX-AGE [80] was an additive clause weighting forerunner of PAWS and had near identical performance.

indicates there is promise for a weight redistribution approach to handling problems with hard and soft constraints.

However, given that we also tried a large number of other DDFW hybrid heuristics, we must conclude that *TWO-LEVEL*'s approach to optimising hard and soft constraint problems appears the stronger. We conjecture that this is because the greater complexity of the DDFW heuristic works against it when dealing with two classes of clauses and also because the *hardMultiplier* approach to updating hard clause weights appears particularly suited for embedding in a two phase additive weighting scheme. Clearly more work is needed to come to a firmer conclusion, such as looking at a broader cross-section of problems and trying out still further DDFW variants. However, our preliminary study suggests the more promising avenue of future research would be to try and improve the *TWO-LEVEL* heuristic itself.

5.7 Summary

The main contribution of this chapter is to have evaluated and extended the weight redistribution strategy of the DDFW heuristic to the domain of unweighted MAX-SAT problems. In the process we have shown that DDFW outperforms several class-leading local search algorithms on a range of previously studied MAX-SAT benchmark problems. These results strengthen the case for the use of weight redistribution in clause weighting, as opposed to using the two-stage increase/reduce strategies of PAWS and SAPS. As with earlier studies, DDFW's performance has been shown to be more robust than RSAPS or a fixed parameter PAWS, making it the most promising approach for situations where manual parameter tuning is impractical.

In the second section of the chapter, we extended DDFW to handle over-constrained problems with hard and soft constraints, producing a new DDFW-O algorithm which we compared with another class leading clause weighting algorithm, *TWO-LEVEL*. The resulting experimental study showed that the DDFW-O heuristic does not perform as well as *TWO-LEVEL* on the hard and soft clause random 3-SAT problems we considered (although DDFW-O did perform better than several other algorithms considered in the original *TWO-LEVEL* study). From this we concluded that a *TWO-LEVEL* approach using a standard additive update scheme is the most promising avenue for further research in the area of optimising hard and soft constraints with local search.

Chapter 6

Conclusions

In this chapter, we conclude the thesis by recaping the main research contributions. We also briefly discuss two future research directions.

6.1 Summary of Contributions

In this thesis, we aimed to advance and further improve the performance of the state-of-the-art in SAT solving. We focused on clause weighting techniques used in stochastic local search methods and developed a deeper understanding of parameter tuning related problems in these methods. We further developed a new method of distributing fixed weights among satisfied and unsatisfied clauses for the design of parameter-free clause weighting schemes. The proposed algorithms were then empirically evaluated on a large suite of standard benchmarks used by the SAT community.

Here is a summary of the thesis:

- Chapter 1 introduced the basic definitions and motivations for the work carried out in this thesis.

- Chapter 2 surveyed the background material on local search techniques. The relevant techniques were classified as memory-based strategies, restart-based strategies, stochastic escape-based strategies and weighting-based strategies. We then discussed the role of parameters in SLS SAT solvers, and the need for developing parameter-free local search methods.

- Chapter 3 described a new SAT-solving approach that maintains clause weights via a weight redistribution mechanism called divide and distribute fixed weights (DDFW). This approach also uses a new neighbourhood structure for weight redistribution. The resulting method was compared with several of the best SLS solvers: PAWS, SAPS, RSAPS, and

AdaptNovelty$^+$. These algorithms were tested on a range of benchmark problems taken from the well-known SATLIB and DIMACS libraries. The results show that weight redistribution based on neighbourhood structure is an effective and competitive heuristic for handling a range of SAT solving problems particularly when parameter tuning is disallowed.

- In Chapter 4 we further studied the performance of DDFW and developed a new variant (DDFW$^+$) that uses an adaptive mechanism to alter clause weights dynamically based on the current search state. Using DDFW's approach to initialise clause weights, the adaptive mechanism then dynamically adjusts the amount of weight that is to be distributed according to the current state of the search space. As a result, further performance improvements were achieved. In addition, we extended our experimental study to investigate the performance of our new approach by using a resolution-based preprocessor on the SAT input. Preprocessing techniques have been shown to simplify and relax a problem and in turn to improve the performance of the solver. In the experimental study, we compared resolution-based versions of DDFW and DDFW$^+$ with resolution-based versions of three of the best performing algorithms from the recent SAT competition: R+RSAPS, R+AdaptNovelty$^+$ and R+G^{2}WSAT. After testing on a range of SATLIB, DIMACS and SAT competition benchmark problems, we concluded that preprocessing can further improve the performance of DDFW by an order of magnitude on some domains (while having a marginal effect on some others). Overall, we also concluded that DDFW$^+$ had the best average performance over the range of problems considered.

- In Chapter 5, we studied how the neighbourhood weight redistribution mechanism can be extended to handle overconstrained problems. Here we focused on unweighted MAX-SAT problems. To handle these problems we extended DDFW in a simple manner to keep track of the current best cost solution. We then applied the algorithm to three challenging classes of unsatisfiable problems obtained from the SATLIB library: the jnh, uuf and bor problem sets. Using these problems, we compared the performance of our extended DDFW with that of RSAPS, AdaptNovelty$^+$, PAWS10 and DDFW$^+$. The results indicated that DDFW was generally superior to the other algorithms used in our study. We also considered extending DDFW to handle over-constrained problems with hard and soft constraints, producing a DDFW-O algorithm which we compared with the class leading *TWO-LEVEL* algorithm.

This study showed *TWO-LEVEL* to have the stronger performance but also showed DDFW-O to be competitive with several other algorithms that have been applied to the hard and soft constraint area.

If we consider these contributions in the light of the original thesis, i.e. *that a clause weighting approach to SAT solving based on redistributing a fixed amount of weight can outperform existing parameter-based and parameter adapting clause weighting algorithms when the option of hand-tuning parameters is ruled out*, then we can conclude that the thesis has been largely supported by our results. Firstly, in Chapter 3, a fixed parameter DDFW was shown to be competitive with two of the best known adaptive SLS solvers (RSAPS and AdaptNovelty$^+$). This shows that, although the tuning of the DDFW parameter does improve performance, DDFW is more robust than other techniques when operating in default mode. Then, in Chapter 4, we showed that DDFW's performance can be further improved by adapting the amount of weight distributed during the search according to the search state. This again moved us one step further to a class-leading parameter-free clause weighting algorithm. The one hurdle that DDFW did not cross was being able to outperform existing clause weighting solvers (such as PAWS) when manual parameter tuning *is* allowed. While this would be an important achievement it also falls out of the scope of the original thesis. Finally, in Chapter 5 we showed that DDFW's competitive performance extends to the unweighted MAX-SAT domain, although the results raised a question mark over the broad applicability of the DDFW$^+$ adaptive mechanism, as it proved less effective overall than the original fixed parameter DDFW.

6.2 Future Directions

The work presented in this thesis opens up several avenues for further research, of which we mention two immediate directions:

- The significance of parameter-free local search methods is well known, and any further progress in this direction would be a worthwhile contribution to the field. We have shown DDFW to be effective with a default parameter setting and DDFW$^+$ to be even better on the under-constrained benchmarks we have chosen. However, a parameter-tuned version of PAWS still has superior performance on under-constrained problems and DDFW$^+$ proved less effective on the over-constrained MAX-SAT problems. Consequently there is more work to be done in developing parameter-free SLS techniques that can capture the perfor-

mance of the best parameter dependent algorithms.

- The last chapter of the thesis extended DDFW to handle unweighted MAX-SAT problems and problems involving hard and soft constraints. DDFW's promising results in the unweighted MAX-SAT domain suggest it would be worthwhile to investigate DDFW on *weighted* MAX-SAT problems. Conversely, the relatively poor results of DDFW on the hard and soft constraint problems suggest that further work is needed to adapt clause weight redistribution to handle multiple classes of constraint. In addition, the strong results of the *TWO-LEVEL* heuristic suggest that future work should look at extending the *TWO-LEVEL* approach to look at more realistic problems and to handle more complex constraint hierarchies.

Bibliography

[1] Anbulagan, Duc Nghia Pham, John Slaney, and Abdul Sattar. Old resolution meets modern SLS. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, pages 354–359, 2005.

[2] Roberto Battiti and Giampietro Tecchiolli. The reactive tabu search. *ORSA Journal on Computing*, 6(2):126–140, 1994.

[3] Ramoón Béjar and Felip Manyà. Solving the round robin problem using propositional logic. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00)*, pages 262–266, 2000.

[4] Alain Billionnet and Alain Sutter. An efficient algorithm for 3-satisfiability problem. *Operations Research Letters*, 12:29–36, 1992.

[5] U. Bistarelli, S. Montanari, and F. Rossi. Semiring-based constraint solving and optimization. *ACM*, 44:201–236, 1997.

[6] James R. Bitner and Edward M. Reingold. Backtrack programming techniques. *Communication of the ACM*, 18(11):651–656, 1975.

[7] B. Borchers and J. Furman. A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. *Journal of Combinatorial Optimization*, 2:299–306, 1999.

[8] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5:223–270, 1992.

[9] Byungki Cha and Kazuo Iwama. Adding new clauses for faster local search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 332–337, 1996.

[10] Byungki Cha, Kazuo Iwama, Y. Kambayashi, and S. Mikazaki. Local search algorithms for partial MAX-SAT. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 263–268, 1997.

[11] P. Z. Chinn, J. Chvatalova, A. K. Dewdney, and N. E. Gibbs. The bandwidth problem for graphs and matrices - a survey. *Journal of Graph Theory*, 6(3):223–254, 1982.

[12] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[13] Stephen A. Cook and David G. Mitchell. Finding hard instances of the satisfiability problem: A survey. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 35:1–17, 1997.

[14] M. Davis and H. Putnam. A computing procedure for quantification theory. In *Automation of Reasoning 1: Classical Papers on Computational Logic 1957-1966*, pages 125–139. 1983.

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

[16] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[17] Rina Dechter and Daniel Frost. Backtracking algorithms for constraint satisfaction problems; a survey, 1998.

[18] Rina Dechter and Daniel Frost. Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence*, 136(2):147–188, 2002.

[19] Jeremy Frank. Learning short-term clause weights for GSAT. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 384–389, 1997.

[20] Eugene Freuder and Richard Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

[21] Eugene C. Freuder. Synthesizing constraint expressions. *Communication of ACM*, 21(11):958–966, 1978.

[22] Eugene C. Freuder. A sufficient condition for backtrack-free search. *Journal of ACM*, 29(1):24–32, 1982.

[23] J. Gasching. A general backtracking algorithm that eliminate the most redundant tests. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 457–462, 1997.

[24] Ian P. Gent. Arc consistency in SAT. In *Proceedings of the Fifteenth European Conference on Artificial Intelligence (ECAI-02)*, pages 121–125, 2002.

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

[26] Ian P. Gent and Toby Walsh. Unsatisfied variables in local search. In *Proceedings of AISB-95: Hybrid Problems, Hybrid Solutions*, pages 73–85, 1995.

[27] Norman E. Gibbs, William G. Poole Jr., and Paul K. Stockmeyer. An algorithm for reducing the bandwidth and profile of a sparse matrix. *SIAM Journal of Numerical Analysis*, 13(2):236–250, 1976.

[28] Fred Glover. Tabu serach - part 1. volume 1(3), pages 190–206, 1989.

[29] O. Martin H. Lourenco and T. Stützle. Iterated local search. In *Technical Report AIDA-00-06, Technishe Universität Darmastadt*, 2002.

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

[31] Robert M. Haralick and Gordon L. Elliott:. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980.

[32] Martin Henz, Lim Yun Fong, Lua Seet Chong, and Roland H.C. Yap. Solving hierarchical constraints over finite domains. In *Proceeedings of the Sixth International Symposium on Artificial Intelligence and Mathematics*, 2000.

[33] Holger Hoos and Thomas Stulze. *Stochastic Local Search.* Morgan Kaufmann, Cambridge, Massachusetts, 2005.

[34] Holger H. Hoos. On the run-time behaviour of stochastic local search algorithms for SAT. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 661–666, 1999.

[35] Holger H. Hoos. SAT-encodings, search space structure, and local search performance. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 296–302, 1999.

[36] Holger H. Hoos. An adaptive noise mechanism for WalkSAT. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-02)*, pages 635–660, 2002.

[37] Holger H. Hoos and Thomas Stutzle. Systematic vs. local search for SAT. In *KI - Kunstliche Intelligenz*, pages 289–293, 1999.

[38] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann, San Francisco, CA, 2005.

[39] F. Hutter and Y. Hamadi. Parameter adjustment based on performance prediction: Towards an instance aware problem solver. In *Technical Report: MSR-TR-2005-125, Microsoft Research, WA*, 2005.

[40] Frank Hutter, Youssef Hamadi, Holger H. Hoos, and Kevin Leyton-Brown. Performance prediction and automated tuning of randomized and parametric algorithms. In *Proceedings of the Twelth International Conference on Principles and Practice of Constraint Programming (CP-06)*, pages 213–228, 2006.

[41] Frank Hutter, Dave A.D. Tompkins, and Holger H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP-02)*, pages 233–248, 2002.

[42] Abdelraouf Ishtaiwi, John Thornton, Anbulagan, Abdul Sattar, and Duc Nghia Pham. Adaptive clause weight redistribution. In *Proceedings of the Twentieth International Conference on Principles and Practice of Constraint Programming (CP-06)*, pages 229–243, 2006.

[43] Abdelraouf Ishtaiwi, John Thornton, Abdul Sattar, and Duc Nghia Pham. Neighbourhood clause weight redistribution in local search for sat. In *Proceedings of the Ninteenth International Conference on Principles and Practice of Constraint Programming (CP-05)*, pages 772–776, 2005.

[44] Henry Kautz, Yongshao Ruan, Dimitris Achlioptas, Carla Gomes, Bart Selman, and Mark Stickel. Balance and filtering in structured satisfiable problems. In *Proceedings of the*

Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01), pages 351–358, 2001.

[45] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 1194–1201, 1996.

[46] S. Kirkpatrick, Jr C. D. Gellat, and Mp. P. Vicchi. Optimization by simulated annealing. In *Science 220*, pages 671–680.

[47] Grzegorz Kondrak and Peter van Beek. A theoretical evaluation of selected backtracking algorithms. In Chris Mellish, editor, *IJCAI'95: Proceedings International Joint Conference on Artificial Intelligence*, Montreal, 1995.

[48] Daniel Le Berre and Laurent Simon. 55 solvers in Vancouver: The SAT 2004 competition. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing, SAT 2004*, pages 321–344, 2004.

[49] Chu Min Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP-97)*, pages 341–355, 1997.

[50] Chu Min Li and Wen Qi Huang. Diversification and determinism in local search for satisfiability. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT-05)*, pages 158–172, 2005.

[51] Chu Min Li and Wenqi Huang. Diversification and determinism in local search for satisfiability. In *Proceedings of 8th SAT*, pages 158–172, 2005.

[52] Yogesh Mahajan, Zhaohui Fu, and Sharad Malik. *Zchaff2004: An Efficient SAT Solver*, volume 3542, pages 360–375. 2004.

[53] David A. McAllester, Bart Selman, and Henry A. Kautz. Evidence for invariants in local search. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 321–326, 1997.

[54] Patrick Mills and Edward Tsang. Guided local search applied to the SAT problem. In *Proceedings of the Fifteenth National Conference of the Australian Society for Operations Research (ASOR-99)*, pages 872–883, 1999.

[55] Patrick Mills and Edward Tsang. Guided local search for solving SAT and weighted MAX-SAT problems. *Journal of Automated Reasoning*, 24(1-2):205–223, 2000.

[56] Paul Morris. The Breakout method for escaping from local minima. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pages 40–45, 1993.

[57] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[58] Duc Nghia Pham, John Thornton, Abdul Sattar, and Adelraouf Ishtaiwi. SAT-based versus CSP-based constraint weighting for satisfiability. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, pages 455–460, 2005.

[59] David Poole, Alan Mackworth, and Randy Goebel. *SComputational Intelligence - A Logical Approach*. Oxford University Press, 1998.

[60] Patrick Prosser. Domain filtering can degrade intelligent backtrack search. In *IJCAI'93: Proceedings International Joint Conference on Artificial Intelligence*, pages 262–267, 1993.

[61] W. Pullan and L. Zhao. Resolvent clause weighting local search. In *Proceedings of 17th Canadian AI*, pages 233–247, 2004.

[62] W. V. Quine. A way to simplify truth functions. *American Mathematical Monthly*, 62:627–631, 1955.

[63] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[64] Daniel Sabin and Eugene C. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming, PPCP'94, Rosario, Orcas Island, Washington, USA*, volume 874, pages 10–20, 1994.

[65] Dale Schuurmans and Finnegan Southey. Local search characteristics of incomplete SAT procedures. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00)*, pages 297–302, 2000.

[66] Dale Schuurmans and Finnegan Southey. Local search characteristics of incomplete SAT procedures. *Artificial Intelligence*, 132:121–150, 2001.

[67] Dale Schuurmans, Finnegan Southey, and Robert C. Holte. The exponentiated subgradient algorithm for heuristic boolean programming. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 334–341, 2001.

[68] Bart Selman and Henry A. Kautz. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 290–295, 1993.

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

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

[71] Yi Shang and Benjamin Wah. A discrete Lagrangian-based global-search method for solving satisfiability problems. *Journal of Global Optimization*, 12(1):61–99, 1998.

[72] Laurent Simon and Daniel Le Berre. Special volume on the SAT 2005 competitions and evaluations. *Journal on Satisfiability, Boolean Modeling and Computation*, 2, 2006.

[73] Laurent Simon, Daniel Le Berre, and Edward Hirsch. The SAT2002 competition report. *Annals of Mathematics and Artificial Intelligence*, 43:307–342, 2005.

[74] Barbara M. Smith. A Tutorial on Constraint Programming. Technical Report 95.14, 1995.

[75] K. Smith, H. Hoos, and T. Stützle. Iterated robust tabu search for MAX-SAT. In *Advances in Artificial Intelligence, 16th Conference of the Canadian Society for Computational Studies of Intelligence*, volume 2671, pages 129–144, 2003.

[76] John Thornton. *Constraint weighting local search for constraint satisfaction.* PhD thesis, Griffith University, Queensland, Australia, 2000.

[77] John Thornton. Clause weighting local search for SAT. *Journal of Automated Reasoning*, 35(1-3):97–142, 2005.

[78] John Thornton, Stuart Bain, Abdul Sattar, and Duc Nghia Pham. A two level local search for MAX-SAT problems with hard and soft constraints. In *Proceedings of the 15th Australian Joint Conference on Artificial Intelligence, AI-2002, Canberra. Lecture Notes in Artificial Intelligence 2557, Springer-Verlag*, pages 603–614, 2002.

[79] John Thornton, D. N. Pham, Stuart Bain, and Valnir Ferreira Jr. Additive versus multiplicative clause weighting for SAT.

[80] John Thornton, Wayne Pullan, and Justin Terry. Towards fewer parameters for clause weighting SAT algorithms. In *Proceedings of the 15th Australian Joint Conference on Artificial Intelligence, AI-2002, Canberra. Lecture Notes in Artificial Intelligence 2557, Springer-Verlag*, pages 569–578, 2002.

[81] Dave A.D. Tompkins and Holger H. Hoos. Scaling and probabilistic smoothing: Dynamic local search for unweighted MAX-SAT. In *Proceedings of the Sixteenth Conference of the Canadian Society for Computational Studies of Intelligence (AI 2003)*, pages 145–159, 2003.

[82] Dave A.D. Tompkins and Holger H. Hoos. Warped landscapes and random acts of SAT solving. In *Proceedings of the Eighth International Symposium on Artificial Intelligence and Mathematics (AIMA-04)*, 2004.

[83] Dave A.D. Tompkins and Holger H. Hoos. UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. In *Theory and Applications of Satisfiability Testing: Revised Selected Papers of the Seventh International Conference (SAT 2004)*, pages 306–320, 2005.

[84] Edward Tsang. *Foundations of constraint satisfaction*. Academic Press Limited, UK, 1996.

[85] R. Vaessens, E. Aarts, and J. Lenstras. Job shop scheduling by local search. In *Technical Report OSOR memorandum 94-05, revised version*, 1999.

[86] Miroslav N. Velev and Randal E. Bryant. Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. *Journal of Symbolic Computation*, 35(2):73–106, 2003.

[87] Chris Voudouris. *Guided Local Search for Combinatorial Optimisation Problems*. PhD thesis, University of Essex, United Kingdom, 1997.

[88] Chris Voudouris and Edward Tsang. Function optimization using local search. In *Technical Report CSM-249, Department of Computer Science, University of Essex, Colchester, CO4 3SQ, UK*, 1995.

[89] Chris Voudouris and Edward Tsang. Guided local search and its application to the travelling salesman problem. *European Journal of Operational Research*, 113(2):469–499, 1999.

[90] Toby Walsh. SAT v CSP. In *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming (CP-00)*, pages 441–456, 2000.

[91] Huang Wenqi, Zhang Defu, and Wang Houxiang. An algorithm based on tabu search for satisfiability problem. *J. Comput. Sci. Technol.*, 17(3):340–346, 2002.

[92] Zhe Wu and Benjamin W. Wah. Trap escaping strategies in discrete Lagrangian methods for solving hard satisfiability problems and maximum satisfiability problems. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 673–678, 1999.

[93] Zhe Wu and Benjamin W. Wah. An efficient global-search strategy in discrete Lagrangian methods for solving hard satisfiability problems. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00)*, pages 310–315, 2000.

[94] M. Yagiura and T. Ibaraki. Analyses on the 2 and 3-flip neighborhoods for the MAX SAT. *Journal of Combinatorial Optimisation*, 3:95–114, 1999.