UNIVERSITY OF AGDER

Multilevel Techniques and Learning Automata for the Maximum Satisfiability (MAXSAT) Problem

By

Øystein Brådland **Mats G. L. Oseland**

Supervisor
Associate Professor Noureddine Bouhmala, UiA/HiVe

This Master's Thesis is carried out as a part of the education at the University of Agder and is therefore approved as a part of this education. However, this does not imply that the University answers for the methods that are used or the conclusions that are drawn.

University of Agder, 2012
Faculty of Engineering and Science
Department of Information and Communication Technology

Abstract

The Maximum Satisfiability (MAXSAT) Problem is a propositional logic and an optimization based problem that has great importance in the theoretical and practical domain. In the recent years MAXSAT has risen great interest in the industry. Example problems from the industry that can be encoded as MAXSAT problems are circuit design and debugging, hardware verification, bioinformatics and scheduling. These kind of problems often tend to be large and increase exponentially with the problem size, and therefore algorithms for solving such problems incorporate different techniques and methods to solve the problems in a smart and efficient manner.

In this thesis we introduce a range of algorithms that extend the well-known Stochastic Local Search (SLS) algorithm called `WalkSAT`. `WalkSAT` is extended with the multilevel paradigm and Learning Automata. The multilevel paradigm is a technique that splits large and difficult problems into smaller problems. These problems are expectedly less complex and therefore easier to solve. Learning Automata are a branch of machine learning that can be seen as a decision-making entity that is employed in an unknown environment. Through feedback from the environment the Learning Automata try to learn the optimal actions.

The core of this thesis is the observations and findings of how these dissimilar techniques affect the performance and behaviour of `WalkSAT` when solving industrial MAXSAT problem instances. Through extensive experiments our results confirm that combining multilevel techniques and Learning Automata with `WalkSAT`, separately and together, give promising results. We compare these composite algorithms with `WalkSAT` on selected industrial MAXSAT problems throughout the thesis, and show that all these composite algorithms perform better than `WalkSAT`.

Acknowledgements

This master's thesis is the final project in the master's program in Information and Communication Technology (ICT) at the University of Agder. The thesis has been under the supervision of Associate Professor Noureddine Bouhmala.

We would like to thank Bouhmala for the valuable guidance and advices throughout the work. This thesis would not have been possible without the help of Bouhmala through his knowledge and insight in his field of research.

A paper entitled *Combining WalkSAT with Learning Automata for MAXSAT* is under writing and will be submitted to a journal.

Grimstad, June 1, 2012

Øystein Brådland　　　　Mats Grunde Løvdahl Oseland

Contents

1 Introduction **1**

1.1 Importance of topic and motivation . . . 1
1.2 Thesis definition . . . 2
1.3 Research questions . . . 2
1.4 Research approach . . . 3
1.5 Contribution to knowledge . . . 3
1.6 Limitations and key assumptions . . . 3
1.7 Thesis outline . . . 4

2 The Maximum Satisfiability Problem **5**

2.1 Review of propositional logic . . . 5
2.2 Conjunctive normal form . . . 6
2.2.1 Transformation of propositional logic formulas to CNF . . . 7
2.3 The Propositional Satisfiability Problem . . . 8
2.4 The Maximum Satisfiability Problem . . . 8

3 Solvers for MAXSAT **9**

3.1 Algorithms for SAT and MAXSAT . . . 9
3.1.1 Stochastic local search algorithms . . . 10
3.1.2 Systematic search algorithms . . . 11
3.2 Solvers combined with Learning Automata . . . 11
3.3 Solvers combined with multilevel techniques . . . 11
3.4 Other prominent MAXSAT solvers . . . 12

4 Experimental Design **13**

4.1 Problem suite . . . 13
4.2 Experiment setup . . . 13
4.3 Implementation and machine specifications . . . 13

5 WalkSAT **14**

5.1 The WalkSAT family 14

5.2 WalkSAT/SKC 15

5.3 WalkSAT implementation 16

5.4 Experiment - WalkSAT noise evaluation 17

5.4.1 Results 17

6 Combining WalkSAT with Multilevel Techniques **20**

6.1 Introduction to multilevel techniques 20

6.2 Multilevel techniques in a MAXSAT context 21

6.3 Multilevel WalkSAT 23

6.3.1 Dynamic noise 24

6.3.2 Variables and cluster flipping 25

6.4 Experiment - benchmarking of multilevel WalkSAT 26

6.4.1 Results 27

7 Combining WalkSAT with Learning Automata **32**

7.1 Learning Automata and reinforcement learning 32

7.2 Learning Automata in a MAXSAT context 33

7.3 Learning Automata WalkSAT 35

7.4 Experiment - Learning Automata state evaluation 37

7.4.1 Results 38

7.5 Experiment - benchmarking of Learning Automata WalkSAT 41

7.5.1 Results 41

8 Combining WalkSAT with Multilevel Techniques and Learning Automata **44**

8.1 Multilevel Learning Automata WalkSAT 44

8.2 Experiment - benchmarking of multilevel Learning Automata WalkSAT 45

8.2.1 Results 45

9 Discussion **49**

9.1 Multilevel WalkSAT 49

9.1.1 Wasting computational resources 49

9.1.2 Number of levels 50

9.1.3 Single versus cluster . . . 51
9.1.4 Static versus dynamic noise . . . 51
9.2 Learning Automata WalkSAT . . . 51
9.3 Multilevel Learning Automata WalkSAT . . . 51
9.4 Summary . . . 52

10 Conclusion of the Research 53
10.1 Conclusion . . . 53
10.2 Further work . . . 54

Bibliography 55

A Benchmarking Results 58
A.1 WalkSAT combined with multilevel techniques . . . 59
A.2 WalkSAT combined with Learning Automata . . . 62
A.3 WalkSAT combined with multilevel techniques and Learning Automata . . . 64

List of Figures

1.1 The parts of the research approach. 3

5.1 Problem instance fpu8-problem.dimacs_24.filtered.cnf, |variables| =160 232, |clauses| = 548 848. Vertical axis gives the number of unsatisfied clauses, horizontal axis represents the noise p. 17

5.2 Problem instance wb1.dimacs.filtered.cnf, |variables| = 49 525, |clauses| = 140 091. Vertical axis gives the number of unsatisfied clauses, horizontal axis represents the noise p. 18

5.3 Problem instance 38584-bug-onevec-gate-0.dimacs.seq.filtered.cnf, |variables| = 314 272, |clauses| = 819 830. Vertical axis gives the number of unsatisfied clauses, horizontal axis represents the noise p. 18

6.1 Clustering single objects. Single objects can be seen as one object when clustered. 20

6.2 Clustering and reduction of variables. |Variables| = 20, |variables per cluster| = 2, |levels| = 3 (not counting level 0). X denotes a variable, and c denotes a cluster. 21

6.3 Dynamic noise procedure: 11 different noise values. Starts at 0%, ends at 100%, increments by 10%. The arrows specifies the possible transitions. . . 24

6.4 Log plot: Problem instance divider-problem.dimacs_3.filtered.cnf, |variables| = 216 900, |clauses| = 711 249. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds. 27

6.5 Log plot: Problem instance dividers_multivec1.dimacs.filtered.cnf, |variables| = 106 128, |clauses| = 397 650. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds. 27

6.6 Log plot: Problem instance i2c_master1.dimacs.filtered.cnf, |variables| = 82 429, |clauses| = 285 987. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds. 28

6.7 Log plot: Problem instance rsdecoder1_blackbox_CSEEblock-problem.dimacs_32. filtered.cnf, |variables| =277 950, |clauses| = 806 460. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds. . . 28

6.8 Log plot: Problem dividers_multivec1.dimacs.filtered.cnf, |variables| = 106 128, |clauses| = 397 650. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds. The vertical bars indicate level transitions. 29

6.9 Log plot: Problem instance spi2.dimacs.filtered.cnf, |variables| = 124 260, |clauses| = 515 813. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds. 29

7.1 Interaction between an agent and its environment. 32

7.2 Learning automaton states, actions, rewards and penalties, and their effects. 33

7.3 Left: Learning automaton state mirroring mechanism. Right: Learning Automata WalkSAT with one state per action. 36

7.4 Problem instance divider-problem.dimacs_3.filtered.cnf. |variables| = 216 900, |clauses| = 711 249. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the number of states per Learning automaton. 38

7.5 Problem instance fpu8-problem.dimacs_24.filtered.cnf. |variables| = 160 232, |clauses| = 548 848. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the number of states per Learning automaton. 38

7.6 Problem instance SM_RX_TOP.dimacs.filtered.cnf. |variables| = 235 456, |clauses| =934 091. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the number of states per Learning automaton. 39

7.7 Log plot: Problem instance divider-problem.dimacs_3.filtered.cnf |variables| = 216 900, |clauses| = 711 249. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds. 41

7.8 Log plot: Problem instance dividers6_hack.dimacs.filtered.cnf |variables| = 35 376, |clauses| = 132 699. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds. 42

7.9 Log plot: Problem instance spi2.dimacs.filtered.cnf |variables| = 124 260, |clauses| = 515 813. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds. 42

8.1 Log plot: Problem instance divider-problem.dimacs_3.filtered.cnf |variables| = 216 900, |clauses| = 711 249. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds. 45

8.2 Log plot: Problem instance fpu8-problem.dimacs_24.filtered.cnf |variables| = 160 232, |clauses| = 548 848. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds. 46

8.3 Log plot: Problem instance dividers_multivec1.dimacs.filtered.cnf |variables| =106 128, |clauses| = 397 650. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds. 46

8.4 Log plot: Problem instance i2c_master1.dimacs.filtered.cnf |variables| = 82 429, |clauses| = 285 987. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds. 47

9.1 Log plot: Problem dividers_multivec1.dimacs.filtered.cnf, |variables| = 106 128, |clauses| = 397 650. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds. The vertical bars indicate level transitions. 50

List of Tables

2.1 Truth tables for logical operations. 6

2.2 Truth table for formula g. 6

5.1 Problem instances used in the experiments, 9 in total. Listed with number of variables and clauses. 17

5.2 Industrial problem instances with the optimal noise for WalkSAT. 19

6.1 Multilevel WalkSAT variants and their differences. 23

6.2 Problem instances used in the experiment, 20 in total. Listed with number of variables and clauses. 26

6.3 Results from the experiment. Multilevel WalkSAT variants and WalkSAT. . 31

7.1 Learning Automata WalkSAT variants and their differences. 36

7.2 Problem instances used in the experiments, 8 in total. Listed with number of variables and clauses. 37

7.3 Results from the experiment. Problem instances given with optimal number of states for both Learning Automata WalkSAT variants. 39

7.4 Results from experiment. Learning Automata WalkSAT variants and WalkSAT. 43

8.1 Multilevel Learning Automata WalkSAT building blocks. 44

8.2 Results from experiment. Multilevel WalkSAT, Learning Automata WalkSAT, multilevel Learning Automata WalkSAT and WalkSAT. 48

9.1 Multilevel consecutive reduction example. Contains levels, %PVA (percentage of possible variable assignments) and total amount of variables in a hypothetical MAXSAT problem with 100 variables at level 0. Note that we have 2 variables per cluster. 50

A.1 Results from benchmarking. Multilevel WalkSAT variants and WalkSAT. . 59

A.2 Results from benchmarking. Multilevel WalkSAT variants and WalkSAT. . 60

A.3 Results from benchmarking. Multilevel WalkSAT variants and WalkSAT. . 61

A.4 Results from benchmarking. Learning Automata WalkSAT variants and WalkSAT. 62

A.5 Results from benchmarking. Learning Automata WalkSAT variants and WalkSAT. 63

A.6 Results from benchmarking. Multilevel WalkSAT, Learning Automata WalkSAT, multilevel Learning Automata WalkSAT and WalkSAT. 64

A.7 Results from benchmarking. Multilevel WalkSAT, Learning Automata WalkSAT, multilevel Learning Automata WalkSAT and WalkSAT. 65

A.8 Results from benchmarking. Multilevel WalkSAT, Learning Automata WalkSAT, multilevel Learning Automata WalkSAT and WalkSAT. 66

List of Algorithms

1 SAT-SLS 10

2 WalkSAT architecture 14

3 WalkSAT 15

4 The multilevel paradigm 22

5 Multilevel WalkSAT 23

6 Learning Automata WalkSAT 35

List of Definitions

1 Thesis definition 2
2 Satisfiability 7
3 The SAT problem 8
4 The unweighted MAXSAT problem 8
5 Break-count 15
6 Cluster flip 24
7 Multilevel-multiplier 24
8 Multilevel-break-count 25

Chapter 1

Introduction

In this chapter we briefly put forward the Maximum Satisfiability Problem (MAXSAT). We introduce the problem area, its importance, and our motivation behind this research in section 1.1. The thesis definition is given in section 1.2, research questions based on the definition are formed in section 1.3. The steps of conducting our research are given in section 1.4. We highlight our contribution to knowledge in section 1.5 followed by limitations and key assumptions in section 1.6. The chapter ends with an outline of the thesis in section 1.7.

1.1 Importance of topic and motivation

Research on solving MAXSAT has great importance. It is mainly important because there are many theoretical and practical problems that can be encoded as MAXSAT problems. Circuit design and debugging, scheduling of how an observation satellite captures photos of Earth, and protein structure alignment (bioinformatics) are all applications and problems from the industry that can be translated to MAXSAT [1–3].

MAXSAT is the optimization variant of the propositional satisfiability problem known as SAT. Due to the MAXSAT problem's structure, the complexity grows exponentially depending on the problem size, hence the time it would take to test all the different solutions could easily be too long. In order to solve a MAXSAT problem efficiently, it is not sufficient to have a fast implementation, but also to be able to move around in the search space in an intelligent manner using various methods and techniques. By doing so, a large number of solutions will be visited and evaluated, thereby increasing the probability for reaching the best possible solution. Hence, the solver will perform more efficiently.

Another important factor for researching on MAXSAT solvers is that if there is a MAXSAT problem, it can be solved by any MAXSAT solver[1]. However, some solvers might work better on specific problem types, and worse on others. It is therefore important to have this in mind when developing a MAXSAT solver.

Stochastic Local Search (SLS) algorithms are known to perform well on SAT problems [4–6]. Therefore, it is interesting to see if it is possible to enhance the performance of SLS algorithms by applying various techniques for solving MAXSAT. Extending an existing SLS algorithm is the integral part of our research, with the goal being that it will be able to solve MAXSAT problems in a more efficient manner. Our research will focus on an

[1] Any MAXSAT problem can essentially be solved by any MAXSAT solver, assuming the problem format is supported.

algorithm called `WalkSAT` that will be combined with Learning Automata and multilevel techniques. We then explore the effects of extending `WalkSAT` with the aforementioned methods. Especially, we want to compare the performance of `WalkSAT` to our proposed extensions of `WalkSAT` on industrial MAXSAT problem instances.

1.2 Thesis definition

Our thesis definition is given below.

Definition 1 *Thesis definition: The MAXSAT problem is known to be NP-hard and is an optimization variant of the SAT problem. The performance of many optimization techniques deteriorates very rapidly mostly due to two reasons. First, the complexity of the problem usually increases with its size, and second, the solution space of the problem increases exponentially with the problem size. Therefore it is beneficial for MAXSAT solvers to move around in the solution space in a smart and efficient manner to reach the optimal or the best possible solution. In the paper A Multilevel Memetic Algorithm for Large SAT-Encoded Problems, Bouhmala proposes a multilevel approach for solving bounded model checking problems. In Using Learning Automata to Enhance Local-Search Based SAT Solvers with Learning Capability by Granmo and Bouhmala, they introduce a novel approach of using machine learning in a SAT context. Results from the multilevel and machine learning methods proved to be promising. The purpose of this master thesis is to see whether combining state of the art stochastic local search methods with multilevel-techniques and Learning Automata for solving MAXSAT results in better performance.*

1.3 Research questions

Based on the thesis definition we establish four research questions. These questions will function as our main objectives. They are important subjects of the study that we will answer by the end of this work.

1. **Learning Automata and multilevel techniques combined with other SAT solvers have recently been used to solve SAT. What will be the outcome when `WalkSAT` is extended with the aforementioned techniques for solving MAXSAT?** Previous successful research shows that Learning Automata paired with GSATRW and RW [7], and multilevel techniques paired with a memetic algorithm [8] for solving SAT problems gave great improvement. However, it is not hereby given that the results will be the same for MAXSAT problems. Further, SLS solvers that perform well in SAT does not necessarily perform equally well in MAXSAT [1]. It might be the case that the structure and semantics of `WalkSAT` will increase or decrease the significance of Learning Automata and multilevel techniques.

2. **Assuming we experiment with different combinations of `WalkSAT` with multilevel techniques, and `WalkSAT` with Learning Automata. What is the optimal combination in both aspects?** `WalkSAT's` structure opens for a degree of freedom for extension. How the combination is done may play an important role. We will therefore experiment with different combinations.

3. **If `WalkSAT` extended with Learning Automata and multilevel techniques separately yields good results, what will be the result of `WalkSAT` combined**

with both techniques? The two techniques differs from each other, and it might be the case that one of the two will boost the other to achieve an overall higher performance when combined.

4. **Is there an optimal configuration of Learning Automata when coupled with `WalkSAT`?** A Learning automaton is a kind of a finite state machine that can be configured in several ways. The number of internal states can affect the performance.

1.4 Research approach

In order to achieve the goals of this thesis the research is divided into four parts. Figure 1.1 illustrates the research approach and the four parts. First, we implement the original version of `WalkSAT`, second, we combine multilevel techniques with `WalkSAT`. In the third part we combine Learning Automata and `WalkSAT`. Finally, we combine both multilevel techniques and Learning Automata with `WalkSAT` at the same time.

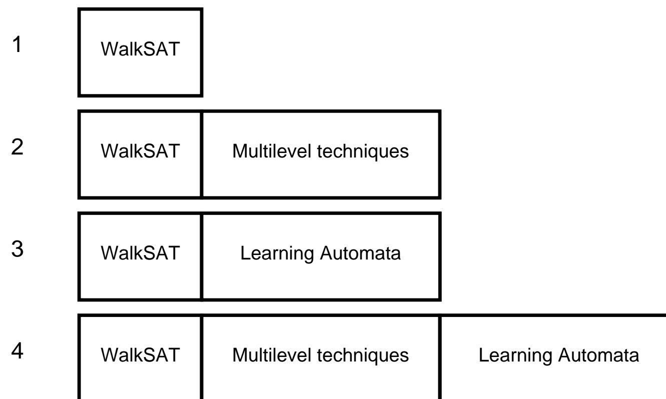

Figure 1.1: The parts of the research approach.

In all parts we perform experiments and compare our proposed algorithms with each other and with `WalkSAT`. Findings from part 2 and 3 will be used as a basis for part 4.

1.5 Contribution to knowledge

The main contribution of our research will be the combination of the well-known stochastic local search algorithm `WalkSAT` with multilevel techniques and Learning Automata. It will provide valuable information on how multilevel techniques and Learning Automata affect the performance of `WalkSAT`. Additionally, this research will function as a foundation and reference point for possible future work.

1.6 Limitations and key assumptions

All the implemented algorithms presented in this thesis are not to be seen as complete algorithms that can serve as final products. They are rather unoptimized prototypes of what can be, and at this stage they only satisfy our needs required to complete and pursue the goals of this thesis. Hence the results displayed in tables and graphs might not correspond to optimized versions. Due to this, the results are not to be compared

with competing solvers from MAXSAT competitions and evaluations where computational speed plays an important role.

1.7 Thesis outline

The remainder of the paper is organized as follows: Chapter 2 holds the problem area of this research, namely the MAXSAT problem. In chapter 3 we present methods for solving MAXSAT, and we also introduce prior research with emphasis on Learning Automata and multilevel techniques for SAT. Notes on experimental design are put forward in chapter 4. In chapter 5 the `WalkSAT` algorithm is set forth and examined in-depth. Chapter 6 contains `WalkSAT` combined with multilevel techniques, and in chapter 7 `WalkSAT` is combined with Learning Automata. We present `WalkSAT` combined with both multilevel techniques and Learning Automata in chapter 8. Then, in chapter 9 we discuss results and observations found from all experiments we have performed in chapter 6, 7 and 8. Finally, we conclude our research and introduce possible further work in chapter 10.

Chapter 2

The Maximum Satisfiability Problem

In this chapter we present the Maximum Satisfiability (MAXSAT) Problem. The chapter comprises preliminary background information that helps the reader get acquainted with MAXSAT.

We start by introducing propositional logic in section 2.1, which is the core structure of a MAXSAT problem. In section 2.2 we show how propositional logic formulas can be transformed into a form called CNF, which is used as input for many SAT and MAXSAT solvers. We then turn the attention to MAXSAT. Due to the close relationship between SAT and MAXSAT, we first introduce SAT in section 2.3, and then we introduce MAXSAT in section 2.4. These sections also point out applications and definitions.

2.1 Review of propositional logic

Propositional logic is a system where the interest lies within propositions and their interrelationships. A proposition is a statement that can be assigned two values, either $TRUE$ or $FALSE$, also equivalent to 1 and 0, hereafter known as truth values. The logical operations, also known as connectives, include $\vee$ (OR and logical disjunction), $\wedge$ (AND and logical conjunction), $\neg$ (NOT and logical negation), $\rightarrow$ (implication), and $\leftrightarrow$ (equivalence). These can be applied on two propositions, but $\neg$ can only be applied on one proposition. When applied, the outcome is another proposition.

A formula with propositions can be evaluated as $TRUE$ or $FALSE$. Examples of propositions from real life are x_1 = *It is* -5 *degrees Celsius outside today in Grimstad*, and x_2 = *I will wear my scarf and mittens today.* These propositions are primitive propositions which can be combined to a compound proposition by using logical conjunction: *It is* -5 *degrees Celsius outside today in Grimstad*, AND *I will wear my scarf and mittens today.* With symbols this can be expressed as $(x_1 \wedge x_2)$.

A truth table is used to show the semantics for a logical operation. Table 2.1 gives the truth tables for the mentioned logical operations.

X	Y	$X \vee Y$	$X \wedge Y$	$\neg X$	$X \leftrightarrow Y$	$X \rightarrow Y$
0	0	0	0	1	1	1
1	0	1	0	0	0	0
0	1	1	0	1	0	1
1	1	1	1	0	1	1

Table 2.1: Truth tables for logical operations.

An example of a propositional logic formula g with propositions, or variables, x_1, x_2, x_3, and x_4 is given below.

$$g = (\neg x_1 \vee (\neg x_2 \rightarrow x_3)) \leftrightarrow x_4 \tag{2.1}$$

The truth table for this formula is given in table 2.2. Each entry in the table shows the truth values x_1, x_2, x_3, and x_4 take to evaluate g as $TRUE/1$ or $FALSE/0$. This can be observed in the rightmost column.

$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{x_3}$	$\mathbf{x_4}$	$\mathbf{g}$
1	1	1	1	1
1	1	1	0	0
1	1	0	1	1
1	1	0	0	0
1	0	1	1	1
1	0	1	0	0
1	0	0	1	0
1	0	0	0	1
0	1	1	1	1
0	1	1	0	0
0	1	0	1	1
0	1	0	0	0
0	0	1	1	1
0	0	1	0	0
0	0	0	1	1
0	0	0	0	0

Table 2.2: Truth table for formula g.

2.2 Conjunctive normal form

A possible encoding form for propositional logic formulas is the Conjunctive Normal Form (CNF). This form is widely used for describing SAT and MAXSAT problems. A CNF formula consists of a conjunction of clauses, $\bigwedge_i C_i$. Each clause C_i is a disjunction of literals, $\bigvee_j x_{i,j}$, where a literal is a propositional variable x ($TRUE$ or $FALSE$) or its negation $\neg x$ [9]. A CNF formula can be expressed as in equation 2.2 and 2.3, where m is the number of clauses and x is a literal in a given clause. Equation 2.4 shows an instance of a CNF formula with four variables x_1, x_2, x_3, x_4 and four clauses.

$$f = \bigwedge_{i=1}^{m} C_i \tag{2.2}$$

$$C_i = \bigvee_{j=1}^{k_i} x_{i,j} \tag{2.3}$$

$$f = (x_1 \vee x_4) \wedge (\neg x_2 \vee x_4) \wedge (\neg x_3 \vee x_4) \wedge (\neg x_4 \vee \neg x_1 \vee x_2 \vee x_3) \tag{2.4}$$

A formula expressed in CNF is satisfied if all its clauses are satisfied, i.e. all clauses evaluate to $TRUE$. For a clause to be satisfied at least one of the literals must evaluate to $TRUE$, because of the disjunctions of literals. In formula 2.4 there are four variables x_1, x_2, x_3 and x_4, which means there are $2^4 = 16$ possible variable assignments of truth values. On the other hand, if the formula has 500 variables then there are $3.27339061 \times 10^{150}$ assignments. In a SAT/MAXSAT context these possible variable assignments are referred to as the search space. One of the assignments of equation 2.4, where $x_1 = TRUE$, $x_2 = FALSE$, $x_3 = TRUE$, $x_4 = TRUE$, makes f evaluate to $TRUE$. Hence, this assignment is called a solution. This can also be seen from the truth table for the formula g (table above), table 2.2 and equation 2.1. Since g can be transformed into CNF yielding f, the table is also valid for f. The table reports that there exists eight solutions, eight rows where the rightmost column holds the value 1 (the rows that are colored gray). Based on this we define *satisfiability* in respect to propositional logic as following:

> **Definition 2** *Satisfiability: Given a propositional formula f. If an assignment of truth values to the variables in f evaluates f to $TRUE$, the assignment* **satisfies** *f. Further, f is claimed to be* **satisfiable** *if and only if there exists at least one assignment that satisfies f [6].*

According to definition 2 we can claim that formula f is satisfiable since there is at least one assignment of truth values that satisfies the formula, thus makes the formula satisfiable.

2.2.1 Transformation of propositional logic formulas to CNF

The main idea behind transforming a propositional logic formula to CNF is by using logic transformation rules, where the goal is to create clauses with literals by only using the logical connectives $\wedge$, $\vee$ and $\neg$. The listing below shows how the propositional logic formula g given in equation 2.1 can be transformed into CNF by using De Morgan's law, double negative law and the distributive law. We also need to decompose the connective double implication $\leftrightarrow$: $(x_1 \leftrightarrow x_2) \Leftrightarrow (x_1 \rightarrow x_2) \wedge (x_2 \rightarrow x_1)$, and the logical connective implication $\rightarrow$: $(x_1 \rightarrow x_2) \Leftrightarrow (\neg x_1 \vee x_2)$ [10].

$$(\neg x_1 \vee (\neg x_2 \rightarrow x_3)) \leftrightarrow x_4$$

$$\Leftrightarrow ((\neg x_1 \vee (\neg x_2 \rightarrow x_3)) \rightarrow x_4) \wedge (x_4 \rightarrow (\neg x_1 \vee (\neg x_2 \rightarrow x_3)))$$

$$\Leftrightarrow (\neg(\neg x_1 \vee (\neg x_2 \rightarrow x_3)) \vee x_4) \wedge (\neg x_4 \vee (\neg x_1 \vee (\neg x_2 \rightarrow x_3)))$$

$$\Leftrightarrow (\neg(\neg x_1 \vee (x_2 \vee x_3)) \vee x_4) \wedge (\neg x_4 \vee (\neg x_1 \vee (x_2 \vee x_3)))$$

$$\Leftrightarrow ((x_1 \wedge \neg(x_2 \vee x_3)) \vee x_4) \wedge (\neg x_4 \vee (\neg x_1 \vee (x_2 \vee x_3)))$$

$$\Leftrightarrow ((x_1 \wedge (\neg x_2 \wedge \neg x_3)) \vee x_4) \wedge (\neg x_4 \vee (\neg x_1 \vee (x_2 \vee x_3)))$$

$$\Leftrightarrow ((x_1 \wedge \neg x_2 \wedge \neg x_3) \vee x_4) \wedge (\neg x_4 \vee \neg x_1 \vee x_2 \vee x_3)$$

$$\Leftrightarrow (x_1 \vee x_4) \wedge (\neg x_2 \vee x_4) \wedge (\neg x_3 \vee x_4) \wedge (\neg x_4 \vee \neg x_1 \vee x_2 \vee x_3)$$

The expression in the last line in the listing above is the same as given in equation 2.1, hence the propositional logic formula has been transformed into CNF. As can be seen from the final result, negation ($\neg$) now only appears on single literals inside clauses. The only logic connectives left are $\wedge$ and $\vee$, and all the occurrences of $\leftrightarrow$ and $\rightarrow$ have been decomposed.

2.3 The Propositional Satisfiability Problem

The Propositional Satisfiability Problem, also known as the SAT problem, is essential in mathematical logic and computing theory, but it also has interests in applications belonging to the practical domain. As can be understood from the name, the structure of the SAT problem builds on the propositional logic presented in the previous sections.

Real-world problems and applications include Blocks World (planning problem in the AI domain), graph coloring, circuit design and hardware verification can all be encoded as SAT problems [1]. The SAT problem is under the NP-complete class of problems [9], and the goal is to determine whether there exists an assignment of truth values to the variables in a given CNF formula such that it becomes satisfiable [6]. We define the SAT problem as following:

Definition 3 *The SAT problem: Given a propositional formula f, does there exist an assignment of truth values to the variables such that f becomes satisfiable?*

2.4 The Maximum Satisfiability Problem

The Maximum Satisfiability (MAXSAT) Problem is an optimization variant of the SAT problem. The complexity of such problems is known to be NP-hard [9]. Applications and problems from the industry involve bioinformatics [3], scheduling [2] and integrated circuits such as field-programmable gate array (FPGA) routing [11].

MAXSAT has several variants: the weighted MAXSAT problem, the unweighted MAXSAT problem, the partial MAXSAT problem and the weighted partial MAXSAT problem [12]. In the weighted variants each clause in a problem has an associated weight, and where the goal is to maximize the total weight of the satisfied clauses. For unweighted MAXSAT the goal is to minimize the amount of unsatisfied clauses[1] in a given CNF formula [6]. We define the unweighted MAXSAT problem as following:

Definition 4 *The unweighted MAXSAT problem: Given a propositional formula f, does there exist an assignment of truth values to the variables that maximizes the number of satisfied clauses in f?*

The difference between SAT and MAXSAT is now evident according to definition 3 and 4: SAT is a decision based problem, while MAXSAT is an optimization based problem. Our focus lies on unweighted MAXSAT, and therefore for the rest of this thesis the term MAXSAT refers to unweighted MAXSAT.

[1] Minimizing the amount of unsatisfied clauses is the same as maximizing the satisfied clauses.

Chapter 3

Solvers for MAXSAT

This chapter comprises methods for solving MAXSAT and introduces prior significant work for solving SAT through the employment of multilevel techniques, and algorithms with learning capabilities. We also present a selection of MAXSAT solvers.

An introduction to SAT and MAXSAT solvers is given in section 3.1, where the two main branches of solvers, stochastic local search algorithms and systematic search algorithms, are discussed in more detail in section 3.1.2 and 3.1.1. Since stochastic local search algorithms is our main focus we provide more information on this subject.

Section 3.2 introduces prior work where various stochastic local search algorithms have been paired with Learning Automata. A memetic algorithm and an algorithm called `GSAT` combined with multilevel techniques are set forth in section 3.3. The chapter ends with section 3.4 that contains a brief overview of prominent MAXSAT solvers.

3.1 Algorithms for SAT and MAXSAT

SAT and MAXSAT solvers, or algorithms, are used to solve propositional problem instances. Due to the definitions of SAT and MAXSAT the goals for these solvers are different, see definition 3 and 4. The goal for a SAT solver is to achieve satisfiability for a given problem. However, a MAXSAT solver seeks to maximize the amount of satisfied clauses. Hence, it might not solve all clauses, but the problem might be satisfied according to given constraints. The goals are realized by *flipping* variables in the problem, where flipping means to negate the truth value of a variable.

As we pointed out in section 2.2, a propositional formula that consists of 500 variables has $3.27339061 \times 10^{150}$ different variable assignments. This illustrates the magnitude of dealing with propositional logic formulas with a high number of variables. It is therefore clear that solvers need to be designed efficiently to obtain good coverage of the search space and find optimal solutions within reasonable time.

Solvers are generally divided into two branches: Stochastic local search algorithms, and systematic search algorithms. The two types of solvers are discussed in more detail in the next sections.

3.1.1 Stochastic local search algorithms

Solvers that are based on Stochastic Local Search (SLS) are often said to be incomplete. An incomplete solver does not guarantee that it will provide a satisfying assignment, or claim that the problem instance is unsatisfiable. Most of the solvers are concerned with reporting satisfying assignments, if there exists one. Incomplete algorithms often run until a constraint has been met, e.g. a time limit, max amount of flips or other such properties.

Systematic search algorithms used to dominate the field of solving large SAT problem instances. However, in the 1990's stochastic local search algorithms proved its strength for solving large and hard SAT problems [13]. Two of the most important solvers among incomplete solvers that paved the way for SLS algorithms for SAT were `GSAT` [5] and `WalkSAT` [4]. `GSAT` was one of the first SLS SAT solvers presented in 1992 and `WalkSAT` was introduced in 1994 which originated from `GSAT` with slight modifications.

The strength and approach for SLS algorithms are the usage of local information. With this approach SLS algorithms are able to find good solutions without exploring the entire search space. They start by examining the search space at some position and then move to a position nearby. From the latter position the decision about the next move is decided by the information from the local neighborhood, where the neighborhood can be seen as a variable assignment. The search is stochastic in the sense that the decisions and moves can be based on stochastic properties. On the other hand, one of the weaknesses of SLS algorithm is that they can get stuck in a state known as local minima[1]. There are several suggested ways to deal with this issue. Restart[2] and non-improving steps are common mechanisms used in many algorithms [14].

In most cases SLS algorithms have an associated objective function. The objective function is often denoted as the number of unsatisfied clauses under a given assignment. Consequently, SLS algorithms try to minimize this function [14]. In general, each variable x in a given problem has a score associated with it, in respect to an assignment. There are different scoring functions of measuring this score, depending on the algorithm (e.g. `GSAT`, `WalkSAT` or other SLS algorithms).

Algorithm 1 SAT-SLS

```
1:  {Input: CNF formula G}
2:  {Output: Satisfied assignment for G, or 'no solution exists'}
3:  for i ← 0 to MAX_TRIES do
4:    T ← random_assignment()
5:    for j ← 0 to MAX_FLIPS do
6:      if T is satisfied then
7:        return T
8:      else
9:        /* heuristics */
10:       variable ← select_variable()
11:     end if
12:     flip(variable)
13:   end for
14: end for
```

[1] A local minimum is a point in the search space where the local neighborhood does not have a point that is better (i.e. lower number of unsatisfied clauses).

[2] A restart may mean a restart of the entire algorithm, or a completely new random assignment.

In algorithm 1 above (reproduced from [15]) a generic outline of SLS algorithms for SAT is given. In the initialization phase all the variables in a given formula are randomly assigned a truth value. Then the algorithm selects a variable and flips its. This is done in an iterative process and can also be seen as a trial and error approach. The heuristics of variable selection is what changes from one SLS algorithm to another. The selection is often done according to the scoring function, and has significant impact on the overall performance [6]. Further, according to algorithm 1 the SLS algorithm will repeat selecting variables and doing flips until MAX_FLIPS is reached. If no solution is found, the algorithm restarts with a new random assignment. The algorithm stops executing when it has reached a predefined number of allowed restarts denoted as MAX_TRIES.

The algorithm presented in the outline is for solving SAT problems, but it is also valid for MAXSAT with minor modifications. Generally, all SLS algorithms can be modified and adapted to solve unweighted MAXSAT problems. However, this does not imply that SLS solvers performing well in SAT yield the same performance in MAXSAT [6].

3.1.2 Systematic search algorithms

Solvers that are classified as systematic search algorithms are known as complete solvers. In comparison to incomplete solvers complete solvers are guaranteed to find a satisfying assignment if one exists, or prove unsatisfiability. Hence, systematic search algorithms search the entire solution space (all possible variable assignments) of the given problem. Because of this complete solvers may be more computationally expensive, and have often difficulties solving large problems.

Many of the complete solvers are based on `DPLL` [16] that was introduced in 1962, which is also based on the `DP` [17] algorithm from 1960. `DPLL` builds a binary tree by assigning truth values to variables, and uses backtracking and branching as its main components. Successful algorithms based on `DPLL/DP` are `GRASP` [18], `SATO` [19], `TABLEAU` [20] and `SATZ` [21].

3.2 Solvers combined with Learning Automata

Previous work in the field of SAT solvers and machine learning have shown that mixing (stochastic) local search SAT algorithms with Learning Automata has given promising results. The goal is to let a SAT solver learn and decide what steps are the best ones to achive better performance than without learning. The machine learning technique that has been used by Granmo and Bouhmala was a Tsetlin automaton. Granmo and Bouhmala's work on fusing Learning Automata (LA) with both `Random Walk` [22] and `GSATRW` [22], giving `LARW` and `LA-GSATRW` respectively, had a higher success rate and better performance than their non-LA counterparts on selected problems from the SATLIB benchmarks in [23].

3.3 Solvers combined with multilevel techniques

Bouhmala proposed the multilevel memetic algorithm `MLVLMA` in [8] for solving large SAT Bounded Model Checking problem instances. His research shows how a memetic algorithm[3] can utilize the multilevel paradigm. The multilevel paradigm is a method for splitting large

[3]Memetic algorithm: An extension of a genetic algorithm with a local search strategy [24].

and difficult problems into smaller problems. These problems are expectedly less complex and therefore easier to solve. The results show that combining a memetic algorithm with the multilevel paradigm can improve the quality of the solution up to 77% in some cases.

In [25] the multilevel paradigm is combined with the stochastic local search algorithm `GSAT`. Conclusions drawn from the results indicate that the multilevel paradigm speeds up `GSAT` and improves its convergence.

3.4 Other prominent MAXSAT solvers

Solvers that distinguished themselves in the Max-SAT Evaluation 2006[4] are `MAXSATZ` [27] and `MAX-DPLL` [28]. The former was the best unweighted MAXSAT solver, and the latter was pointed out as the best solver for weighted MAXSAT. `MAX-DPLL` was also announced as the second best unweighted solver [29, 30].

Other algorithms related to our work that are based on SLS for solving MAXSAT are `SAPS` [31] and `DLS-MC` [32]. Both of these algorithms use a technique called Dynamic Local Search (DLS). DLS is used to avoid getting stuck in local minima. This is done by dynamically altering the evaluation function of solutions at run time. Further, DLS allows the search space to be dynamically changed and modified, thus the solver is guided to avoid local minima.

[4]The MAXSAT Evaluation [26] is an affiliated event of *International Conferences on Theory and Applications of Satisfiability Testing* where the performance of MAXSAT solvers is evaluated.

Chapter 4

Experimental Design

In this chapter we give general information on how the experiments and benchmarking of algorithms have been performed. We present the problem suite in section 4.1. Information about the setup of the experiments is given in section 4.2, and in section 4.3 we report notes on the algorithms' implementation, and system specifications for our test environment.

4.1 Problem suite

In our experiments and benchmarks we use a selection of the same problems as used in the MAXSAT Evaluations. All the problems follow the DIMACS CNF file format [33]. We are focusing on industrial problems, and all of these are available at MAXSAT Evaluation websites found at http://www.maxsat.udl.cat/. Our problem suite is a mixture of the problems used in MAXSAT Evaluation 2008, 2009, 2010 and 2011.

4.2 Experiment setup

All the algorithms run 10 times on the same problem instance due the random nature in stochastic local search algorithms. We calculate the mean results from these 10 times to get an overview of how the algorithms perform in general. These mean results are used as a basis for all the graphs and numbers in tables occurring in this thesis. In addition, all the algorithms have an execution duration of x seconds, or y flips. When presenting figures, graphs and tables we use the notation $|.|$ for size (e.g. $|\text{clauses}| = 4$), and # for "number of" (e.g. #Unsatisfied clauses equals "number of Unsatisfied clauses"). We also use graphs in logarithmic and arithmetic scale to visualize data and results. For some results we also give sample variance and sample standard deviation.

4.3 Implementation and machine specifications

All algorithms have been implemented by using the *C++* programming language, and run on a machine with the following specifications:

- Central processing unit (CPU): 2x Intel Xeon Six Core E5645 2.4 GHz with 12 MB Cache
- Memory (RAM): 24 GB DDR
- Operating system (OS): Ubuntu 11.10 - 64 Bit

Chapter 5

WalkSAT

In chapter 3 we introduced methods for solving the MAXSAT problem. We especially focused on the branch called stochastic local search algorithms. In this chapter we take a closer look at the algorithm called `WalkSAT`.

Introduction to the `WalkSAT` family is given in section 5.1, while in section 5.2 we introduce `WalkSAT/SKC` which is our focus in this thesis. In section 5.3 we point out the implementation of this algorithm, and in section 5.4 we present results from noise evaluation.

5.1 The WalkSAT family

The `WalkSAT` family, often referred to as an architecture, is a collective term for successful stochastic local search algorithms for solving propositional logic problems. There exists a great amount of `WalkSAT` variants such as `WalkSAT/SKC`, `Novelty`, `Novelty+`, `Adaptive Novelty+`, `R-novelty` and `Walk-SAT/Tabu` [6]. Nonetheless, all the variants have the same basis for variable selection and is done in two stages: In stage one a currently unsatisfied clause is picked at random.

Algorithm 2 WalkSAT architecture

1: {**Input:** CNF formula G}
2: {**Output:** Satisfied assignment for G, or 'no solution exists'}
3: **for** $i \leftarrow 0$ to MAX_TRIES **do**
4:    $T \leftarrow$ random_assignment()
5:    **for** $j \leftarrow 0$ to MAX_FLIPS **do**
6:       **if** T is satisfied **then**
7:          **return** T
8:       **end if**
9:       /* stage 1*/
10:       $C_k \leftarrow$ random_unsatisfied_clause()
11:       /* stage 2*/
12:       $variable \leftarrow random_variable(C_k)$
13:       flip(variable)
14:    **end for**
15: **end for**

In stage two a variable that occurs in the unsatisfied clause is chosen at random. A generic outline of the `WalkSAT` architecture is given in algorithm 2. The main difference between all variants is how the variable is selected, as we will see in the next section. Due to this two-stage process, we observe that `WalkSAT` has 2-way randomness built in. In addition, these algorithms incorporate

randomized behaviour, such that during the execution they will perform a random move according to a given probability on how to select a variable. This probability is the so-called noise parameter. As with any other stochastic local search algorithms, the noise parameter is also one of the key elements that influences the performance for all `WalkSAT` variants.

5.2 WalkSAT/SKC

The `WalkSAT/SKC` [4] algorithm was introduced by Selman, Kautz, and Cohen in 1994. It was the first algorithm belonging to the `WalkSAT` architecture, and is the algorithm of interest in our thesis. For simplicity, we will for the rest of this paper refer to `WalkSAT/SKC` as `WalkSAT`.

As stated earlier, SLS algorithms and variants of `WalkSAT` mainly differ in the variable selection. `WalkSAT` contains three moves for variable selection: (1) a side move/best move, (2) a greedy move, and (3) a random move. The pseudo code for `WalkSAT` is given in algorithm 3.

Algorithm 3 WalkSAT

```
 1: {Input: CNF formula G}
 2: {Output: Satisfied assignment for G, or 'no solution exists'}
 3: Initialization: p_noise ∈ [0, 1]
 4: for i ← 0 to MAX_TRIES do
 5:   T ← random_assignment
 6:   for j ← 0 to MAX_FLIPS do
 7:     if T is satisfied then
 8:       return T
 9:     end if
10:     C_k ← random_unsatisfied_clause()
11:     /* best move/side move */
12:     if ∃ variable v ∈ C_k with break-count = 0 then
13:       variable ← v
14:     else
15:       if random(0, 1) < p_noise then
16:         /* random move*/
17:         variable ← random_variable(C_k)
18:       else
19:         /* greedy move*/
20:         variable ← random_lowest_breakcount_variable(C_k)
21:       end if
22:     end if
23:     flip(variable)
24:   end for
25: end for
```

The scoring function for `WalkSAT` is the break-count, and is defined as the number of clauses that currently are satisfied but will become unsatisfied after a variable is flipped, see definition 5. Consequently, each variable has its own break-count.

> **Definition 5** *Break-count: The number of clauses that are currently satisfied but will become broken (unsatisfied) after a variable is flipped [6].*

`WalkSAT` starts with an initialization phase that generates a random truth assignment for all variables in a given problem. Then an unsatisfied clause is picked at random. At line 12 in algorithm 3 the side move is chosen if there exists a variable or multiple variables with break-count = 0.

In the case of multiple variables, one variable is chosen at random at line 13. The side move is said to be a side move because it does not improve or worsen the quality of the solution, but the variable assignment is different. Therefore, it can also be seen as a best move, because no clauses will break if the selected variable is flipped. This move is crucial for the performance, and it also helps `WalkSAT` to get out of local minima. At line 14, if the break-count $\neq 0$, then the variable will be chosen either by the random move, or by the greedy move.

The random move at line 16 randomly picks a variable in the chosen unsatisfied clause. This move is also referred to as the walk move. It might not be a beneficial move, but it helps to get a better coverage of the search space. The probability for making the random move is denoted as p. This is the so-called noise parameter, where $p \in [0.0, 1.0]$, and 1.0 is equal to 100%. In other words, the noise parameter also controls the greediness (greedy move) of `WalkSAT`. There is no standard value for the p parameter, as it varies and depends on the problem type and size. Therefore, the optimal noise value is not easily obtained without extensive empirical tuning[1] [35]. However, for Random 3-SAT problems, a type of problem where each clause has exactly 3 literals, p is often set to ≈ 0.55 [36].

With the probability $1 - p$ the greedy move is taken. The greedy move selects a variable by computing the score (break-count) for each variable within the unsatisfied clause by using the scoring function. The variable with the lowest score is chosen, or if there are multiple variables with the same lowest score then one is chosen at random.

The remaining part of the algorithm follows the `WalkSAT` architecture; the solver will continue flipping variables at line 23 chosen by the side, random or greedy move until the problem is satisfied, or until MAX_FLIPS has been reached. If no solution has been found during MAX_FLIPS then the algorithm restarts with a new random truth assignment. When it has iterated through MAX_TRIES the algorithm stops executing.

It is implied that `WalkSAT` can be used to solve MAXSAT problems. The only modification needed is that the algorithm keeps record of the best number of satisfied clauses, or equally, the lowest number of unsatisfied clauses. This does not change the behavior and workings of `WalkSAT`, but the output of the algorithm will be the lowest number of unsatisfied clauses instead of a satisfying assignment.

5.3 WalkSAT implementation

Our implementation of `WalkSAT` is based on algorithm 3. This is the version that was introduced by Selman, Cohen and Kautz in [4]. A version of `WalkSAT` with source code by Henry Kautz is available at http://www.cs.rochester.edu/u/kautz/walksat/. We have verified that our implementation of `WalkSAT` behaves correctly and produces the same results. This has been done by running the same problems with the same amount of flips, and then compared the results. The results show that our implementation is valid, however, it is not optimized and cannot compete in terms of speed.

[1] In [34] an algorithm called `Auto-WalkSAT` is proposed that automatically tunes the noise parameter.

5.4 Experiment - WalkSAT noise evaluation

In this section we perform noise evaluation of `WalkSAT`. We examine the importance of the noise parameter in respect to industrial problem instances. The goal is to find the optimal value for the noise parameter. By optimal noise we mean the noise that gives the least number of unsatisfied clauses when the algorithm has finished executing. The noise value discovered will be used to run `WalkSAT` on other industrial problem instances throughout the thesis.

In the experiment we run `WalkSAT` with the noise-set $p = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]$, and MAX_FLIPS is set to 300 000 000. We use problem instances listed in table 5.1.

Problem instance	**#Variables**	**#Clauses**
rsdecoder2.dimacs.filtered.cnf	415 480	1 632 526
mem_ctrl-debug.dimacs.cn.cnf	381 721	505 547
s38584-bug-onevec-gate-0.dimacs.seq.filtered.cnf	314 272	819 830
SM_RX_TOP.dimacs.filtered.cnf	235 456	934 091
divider-problem.dimacs_11.filtered.cnf	215 964	709 377
fpu8-problem.dimacs_24.filtered.cnf	160 232	548 848
i2c_master1.dimacs.filtered.cnf	82 429	285 987
wb1.dimacs.filtered.cnf	49 525	140 091
dividers6_hack.dimacs.filtered.cnf	35 376	132 699

Table 5.1: Problem instances used in the experiments, 9 in total. Listed with number of variables and clauses.

5.4.1 Results

Figure 5.1 shows a noise graph for the problem fpu8-problem.dimacs_24.filtered.cnf. We observe that a bigger noise value increases the number of unsatisfied clauses.

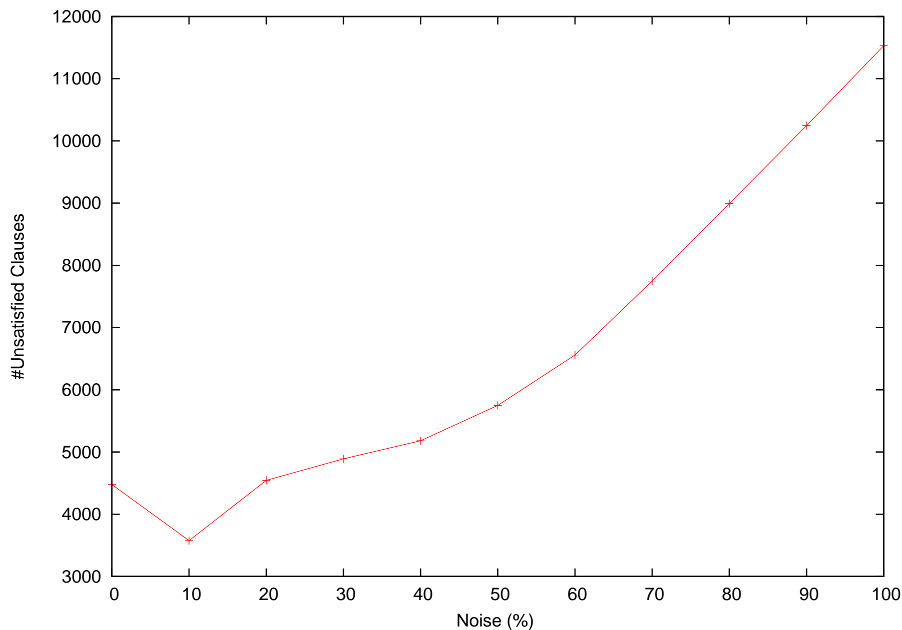

Figure 5.1: Problem instance fpu8-problem.dimacs_24.filtered.cnf, |variables| =160 232, |clauses| = 548 848. Vertical axis gives the number of unsatisfied clauses, horizontal axis represents the noise p.

The figure below shows `WalkSAT` solving wb1.dimacs.filtered.cnf. We observe that $p = 10\%$ is a good value for the noise parameter.

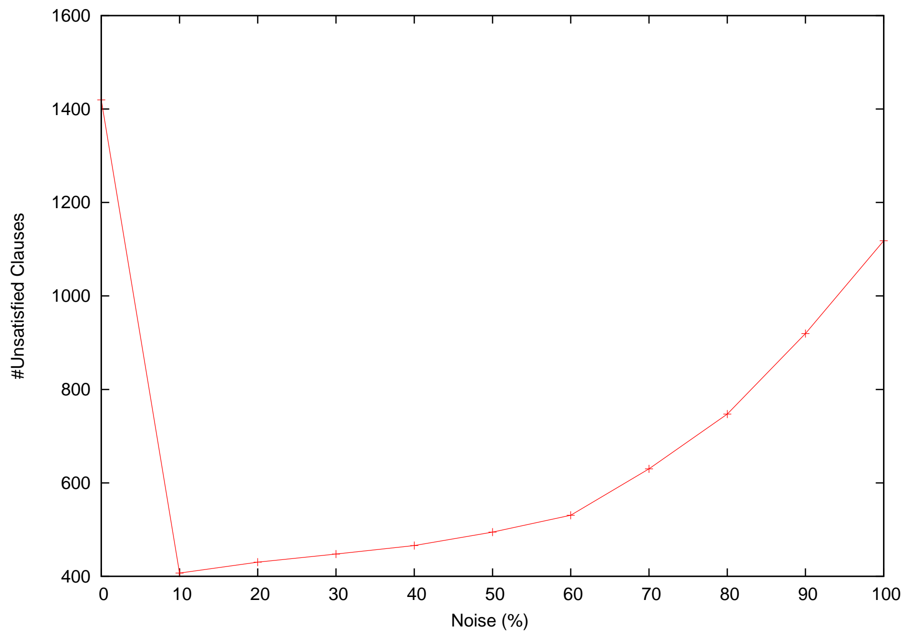

Figure 5.2: Problem instance wb1.dimacs.filtered.cnf, |variables| = 49 525, |clauses| = 140 091. Vertical axis gives the number of unsatisfied clauses, horizontal axis represents the noise p.

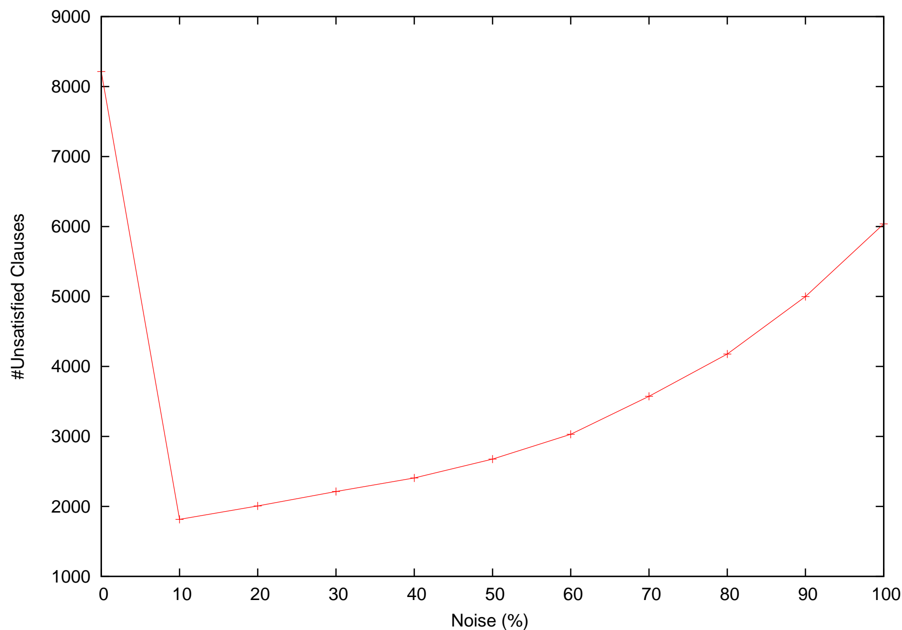

Figure 5.3: Problem instance 38584-bug-onevec-gate-0.dimacs.seq.filtered.cnf, |variables| = 314 272, |clauses| = 819 830. Vertical axis gives the number of unsatisfied clauses, horizontal axis represents the noise p.

We see that the number of unsatisfied clauses in figures 5.2 and 5.3 is very high at 0% noise, and the same for 100%. The optimal noise seems to be 10% for these two as well. Remaining results are given in table 5.2 below, with optimal noise and number of unsatisfied clauses for each problem.

Problem instance	**%Noise**	**#Mean unsatisfied clauses**
rsdecoder2.dimacs.filtered.cnf	10	675
i2c_master1.dimacs.filtered.cnf	10	316
SM_RX_TOP.dimacs.filtered.cnf	30	2 630
s38584-bug-onevec-gate-0.dimacs.seq.filtered.cnf	10	1 816
divider-problem.dimacs_11.filtered.cnf	10	5 952
wb1.dimacs.filtered.cnf	10	407
fpu8-problem.dimacs_24.filtered.cnf	10	3 576
dividers6_hack.dimacs.filtered.cnf	10	419
mem_ctrl-debug.dimacs.cnf.cnf	10	158

Table 5.2: Industrial problem instances with the optimal noise for WalkSAT.

The results suggest that `WalkSAT` prefers 10% noise on most of the industrial problems shown in table 5.2. The results also indicate that `WalkSAT` needs a rather fair amount of greedy moves compared to random moves when solving industrial MAXSAT problems. Based on this we conclude that $p = 10\%$ is the optimal noise for these problems.

Chapter 6

Combining WalkSAT with Multilevel Techniques

In this chapter we extend `WalkSAT` with multilevel techniques. We start with an introduction of the multilevel paradigm in section 6.1. How this paradigm can be applied in a MAXSAT context is given in section 6.2. In section 6.3 we present our solution where we extend `WalkSAT` with multilevel techniques. Section 6.4 ends this chapter with benchmarking and corresponding results.

6.1 Introduction to multilevel techniques

We give an informal introduction to the multilevel paradigm by the following everyday example: Imagine you have bought a new house and you are in the process of moving your belongings from your old apartment to your new house. There are several ways to do this. It is possible to think of this as an optimization problem, where the goal is to avoid unnecessary back-and-forth trips between your old and new home. Let us say we have a large amount of objects in different shapes and sizes that we want to move. Obviously, moving these objects one-by-one would not be very efficient. If we had chosen this method, we would have spent a lot of time and trips. However, if we packed several objects into one box, we would be able to move more objects at the same time. Basically what we do is to gather, or cluster, several objects into one object, see figure 6.1. We have gone from one big problem, many single objects, to a smaller problem where the single objects can be seen as one object. Our problem now consists of one box that encapsulates several objects. Hopefully, it now becomes more efficient and easier to move your belongings to your new house.

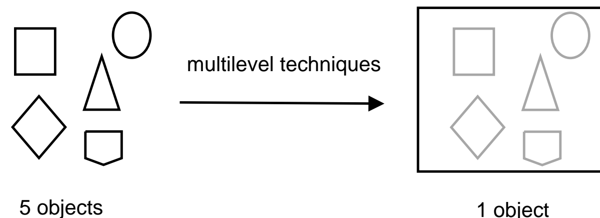

Figure 6.1: Clustering single objects. Single objects can be seen as one object when clustered.

With the example above in mind, there exists several problems where multilevel techniques can and have been applied. Multilevel techniques have been utilized in Graph Coloring [37] and on the well-known combinatorial optimization problem called The Travelling Salesman Problem (TSP) [38,39]. In the next section we explain how the multilevel paradigm can be applied to solve MAXSAT problems.

6.2 Multilevel techniques in a MAXSAT context

In chapter 3 we briefly reviewed Bouhmala's work on solving the SAT problem with multilevel techniques. We will now go into more details, and explain multilevel techniques in a SAT context. Due to the similarities between SAT and MAXSAT, we assume the reader acknowledges that this technique is applicable to MAXSAT in an identical way.

The multilevel algorithm, or paradigm, consists of 4 steps or phases: (1) reduction, (2) initial solution, (3) projection, and (4) refinement. The idea is that the SAT problem can be split up into smaller problems, i.e. multiple levels starting from the start state (level 0) to a given maximum reduced state (level N). Each level is simpler than the previous level. An illustration of the reduction phase is given in figure 6.2.

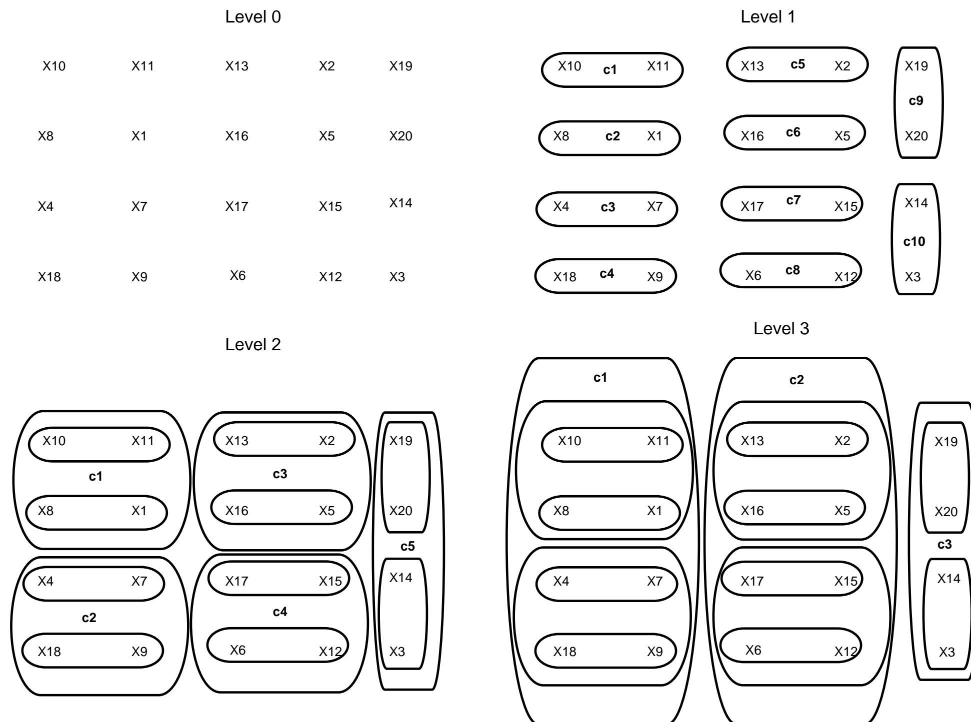

Figure 6.2: Clustering and reduction of variables. |Variables| = 20, |variables per cluster| = 2, |levels| = 3 (not counting level 0). X denotes a variable, and c denotes a cluster.

The first step, or the reduction process, combines two variables at random[1] from level 0 (also called the *top level* in multilevel terminology) into a cluster, and continues doing the same operation until there are no more variables to combine. All the clusters that have been made, are said to be contained in level 1. This means one such reduction step, takes us from level X to level X+1. Be aware that below level 0, instead of combining variables, the process would combine clusters since there are no more variables. Note that if there is only one variable or cluster left after all the other variables or clusters have been combined, then this variable or cluster is copied to the next level. An example of this can be seen in figure 6.2; cluster c_5 in level 2 is copied to level 3 as cluster c_3.

In algorithm 4 below (an algorithm from [8] and slightly modified) we can see that reduction continues until the desired amount of levels has been created. In multilevel terminology we call the final level either the *lowest level, bottom level* or *level N*. As is evident from the illustration in figure 6.2 we see that level N corresponds to level 3. We also see that there are 20 variables in level 0, and in level 1 two and two of these variables are *clustered* together. Then at level 2, we have clusters of two and two clusters from level 1. Finally in level 3 we have clusters of two and two clusters from level 2.

[1]Note that due to illustrational purposes, figure 6.2 contains clusters with variables combined in an orderly fashion.

When we have finished reducing the problem according to some level limit (e.g. maximum of 3 levels) then we are left with level N. We now choose to assign an initial solution to each of the clusters in level N. In a SAT or MAXSAT context, this initial solution is a cluster assignment of truth values. Looking at the illustration in figure 6.2 this corresponds to setting each cluster in level 3 to either $TRUE$ or $FALSE$. An example of such a cluster assignment would be $c_1 = TRUE, c_2 = TRUE, c_3 = FALSE$. Note that we can either think of a cluster as a single variable, or as a collection of variables. If we think of the clusters as collections of variables, we say that if a cluster is assigned a truth value then all subclusters and hence all subvariables are assigned that same truth value.

Algorithm 4 The multilevel paradigm

1: {**Input:** SAT problem P_0}
2: {**Output:** Solution $S_{final(P_0)}$}
3: **Initialization:** $level \leftarrow 0$
4: **while** MAX_LEVELS not reached **do**
5: $\quad P_{level+1} \leftarrow Reduce(P_{level})$
6: $\quad level \leftarrow level + 1$
7: **end while**
8: {Initial solution computed at the most reduced level}
9: $S(P_{level}) \leftarrow Initialsolution(P_{level})$
10: **while** $level > 0$ **do**
11: $\quad S_{start}(P_{level-1}) \leftarrow Project(S_{final}(P_{level}))$
12: $\quad S_{final}(P_{level-1}) \leftarrow Refine(S_{start}(P_{level-1}))$
13: $\quad level \leftarrow level - 1$
14: **end while**

After we have assigned an initial truth value to all the clusters at level N, we see that in algorithm 4 we continue with the projection phase. Projection is the opposite of reduction, meaning it extracts the clusters or variables that are compressed into clusters. The projection process hence takes us from level X to level X-1. A side effect of this is that at each projection phase we keep the variable or cluster assignment from the previous level. This is evident if we think of each cluster as a collection of subclusters and subvariables. Continuing with our example in figure 6.2 from earlier, projection would mean that if $level = 3$ then we would go from level 3 to level 2. Keep in mind that the solution, or cluster assignment, from earlier would be passed on to level 2. Thus, at level 2 we would have the same cluster assignment as earlier. In level 3 we had the solution: $c_1 = TRUE, c_2 = TRUE, c_3 = FALSE$, and since each cluster would be unpacked from level 3 to level 2 the solution for level 2 would now become $c_1 = TRUE, c_2 = TRUE$, $c_3 = TRUE, c_4 = TRUE, c_5 = FALSE$.

Finally, we refine the solution at each level. This means going through a process where we change the cluster assignment so as to try to achieve a better solution. A better solution in a SAT context would correspond to minimizing the amount of unsatisfied clauses in a given SAT problem instance. Later in this chapter we will use `WalkSAT` as this refinement step.

Algorithm 4 repeats until it has projected and refined all the levels, and is back at level 0. The benefits of multilevel lies within the fact that the problem is packed together into multiple hierarchies. For SAT problems this means that a larger area of the entire variable assignment is evaluated. Thus, we believe that when a good solution is found at level N, this solution is a good starting point for the next level, and so on.

6.3 Multilevel WalkSAT

In the preceding section we set forth the four phases of the multilevel paradigm: (1) reduction, (2) initial solution, (3) projection and (4) refinement. As hinted near the end of the previous section, the combination of `WalkSAT` and multilevel techniques is accomplished by letting `WalkSAT` be the refinement algorithm in step 4. The general operation for `multilevel WalkSAT` is given in algorithm 5, and at line 12 we see that `WalkSAT` acts as the refinement algorithm.

Algorithm 5 Multilevel WalkSAT

1: {**Input:** SAT problem P_0}
2: {**Output:** Solution $S_{final(P_0)}$}
3: **Initialization:** $level \leftarrow 0$
4: **while** MAX_LEVELS not reached **do**
5: $P_{level+1} \leftarrow Reduce(P_{level})$
6: $level \leftarrow level + 1$
7: **end while**
8: {Initial solution computed at the most reduced level}
9: $S(P_{level}) \leftarrow Initialsolution(P_{level})$
10: **while** $level > 0$ **do**
11: $S_{start}(P_{level-1}) \leftarrow Project(S_{final}(P_{level}))$
12: $S_{final}(P_{level-1}) \leftarrow WalkSAT(S_{start}(P_{level-1}))$
13: $level \leftarrow level - 1$
14: **end while**

In order to let `WalkSAT` work in a multilevel environment, some modifications are needed. We are primarily dealing with another way of flipping variables, namely through clusters. Therefore, we have experimented with two different ways of combining the multilevel paradigm and `WalkSAT` resulting in `mlv-wsat1.0` and `mlv-wsat2.0`. These two algorithms will be introduced in section 6.3.2. We have also experimented with having a dynamic noise parameter (described in more detail in section 6.3.1), as opposed to a static noise parameter like in `WalkSAT`, giving `mlv-wsat1.1` and `mlv-wsat2.1`.

Algorithm	Greedy move flip method	Random move flip method	Best move flip method	Noise method
mlv-wsat1.0	Cluster	Cluster	Cluster	Static
mlv-wsat1.1	Cluster	Cluster	Cluster	Dynamic
mlv-wsat2.0	Single	Cluster	Single	Static
mlv-wsat2.1	Single	Cluster	Single	Dynamic

Table 6.1: Multilevel WalkSAT variants and their differences.

After the reduction phase has been completed we give all the clusters at level N a random initial value of either $FALSE$ or $TRUE$. Then we calculate an initial solution. In our implementation the calculation of the initial solution is the same as the refinement step, meaning we also run multilevel `WalkSAT` to find our initial solution at level N.

To ensure that each level has a chance of finding a better solution than the previous level, because of the stochastic nature of `WalkSAT`, we let each level run at least X cluster flips in each refinement phase, see definition 6. One single cluster flip of cluster c_1 on level 3 in figure 6.2 would correspond to 8 variable flips. Consequently, c_3 on level 3 would correspond to 4 variable flips. The value of X consists of the amount of clusters in the current level multiplied by a given value, the so-called

multilevel-multiplier, see definition 7. Looking at figure 6.2 we see that for level 3, the value of X would be 3 clusters × 10 = 30, where 10 is the multilevel-multiplier. If `WalkSAT` finds a better solution before X cluster flips, the current level is allowed to run X cluster flips more, and so on. Therefore, the projection process only starts if the current level has completed at least X cluster flips without improving the solution. This is common for level N, level N-1, ..., and level 1. However, for level 0 there are no longer any clusters, only variables (see figure 6.2). Since there are only variables left, `multilevel WalkSAT` turns into regular `WalkSAT`. When the algorithm has reached level 0, there are no more levels to project to so the X cluster flips are no longer considered. Due to this, `WalkSAT` can execute until it is aborted, either due to hitting a target solution (e.g. 30 unsatisfied clauses left), a time, or a flip limit.

Definition 6 *Cluster flip: A cluster flip is flipping all the variables in a given cluster. One cluster flip counts as N flips, where N is the amount of variables (not clusters) in that given cluster.*

Definition 7 *Multilevel-multiplier: A constant that is multiplied by the amount of clusters and/or variables in given cluster.*

6.3.1 Dynamic noise

In addition to having a static noise parameter for two of our `multilevel WalkSAT` algorithms, we also experimented with having a dynamic noise, i.e. a noise parameter that changes at runtime. This has been done because the performance of `WalkSAT` heavily depends on the noise parameter, as seen in the `WalkSAT` noise evaluation, section 5.4. The levels might also prefer an individual noise value.

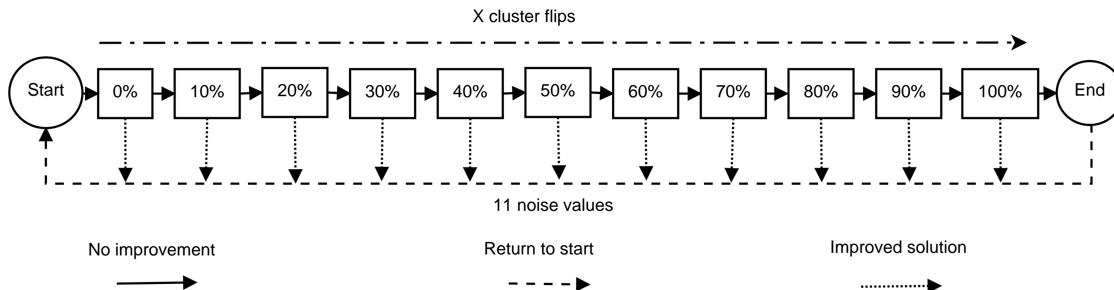

Figure 6.3: Dynamic noise procedure: 11 different noise values. Starts at 0%, ends at 100%, increments by 10%. The arrows specifies the possible transitions.

Figure 6.3 above illustrates the dynamic noise procedure. In this example we have 11 different noise values, due to increments of 10% from 0% up to and including 100%. These noise values are applied directly to `multilevel WalkSAT` just as if they were static noises. The duration each noise value is given is controlled by the value of X cluster flips as can be seen in the figure. Each noise value is given an equal share of the cluster flips. In this example and in our implementation the share each noise value is given is X cluster flips divided by 11.

Assuming we are at a level below 0, the dynamic noise procedure works as follows: First, we start by allocating a share S_X of X cluster flips to each noise value as described. `Multilevel WalkSAT` then runs with noise 0% for S_X flips. If the current solution does not improve during this time, then we advance to the next noise 10% (by following the solid black arrows). However, if the solution improves we go back to the start (by following the dotted arrows). Naturally, the procedure repeats all over again.

When `multilevel WalkSAT` reaches X cluster flips, due to no solution improvement, we go the next level. At the start of the next level, X is given a new value, and hence the dynamic noise procedure restarts as seen in figure 6.3. However, at level 0, there are no clusters, so the share S_X each noise is given is the number of variables divided by 11. Consequently, the dynamic noise procedure works like before.

6.3.2 Variables and cluster flipping

In table 6.1 there are two different versions of multilevel `WalkSAT`, not counting the dynamic noise versions since they do not affect the multilevel paradigm. Whereas `WalkSAT` picks a variable from the randomly selected unsatisfied clause, both `Multilevel WalkSAT` variants pick a variable's corresponding cluster from the randomly selected unsatisfied clause. For the sake of simplicity, we say that we pick a cluster from the randomly selected unsatisfied clause.

The first version, `mlv-wsat1.0`, has cluster flipping on the random move, the greedy move and on the best move. All moves are essentially equivalent to the original `WalkSAT's` moves as mentioned in chapter 5, except for picking clusters instead of variables. To be able to pick the best cluster in an unsatisfied clause, `WalkSAT's` break-count needs to be altered to accomodate for choosing the best cluster. Instead of looking at how many clauses a single variable breaks when flipped, we now need to look at how many clauses are broken by a cluster when flipped, see definition 8.

> **Definition 8** *Multilevel-break-count: The number of clauses that are currently satisfied but will become broken (unsatisfied) after a cluster is flipped.*

The second version, `mlv-wsat2.0`, has single variable flipping on the greedy move and best move, and cluster flipping on the random move. When a single variable is selected, it means that only this variable's truth value is flipped (like in regular `WalkSAT`), even though it belongs to a cluster. This version uses the regular break-count calculation from `WalkSAT` since the greedy move operates on variables.

In summary, `mlv-wsat1.0` is the true version of `multilevel WalkSAT`. By true version we refer to the way the multilevel paradigm has been implemented in [8]. This version has cluster flipping on the random, greedy and best move. `mlv-wsat2.0` is a mix of `WalkSAT` and `mlv-wsat1.0`. The idea of having just cluster flips on the random move of `mlv-wsat2.0` was that the `WalkSAT` algorithm already performs very well, and that it would help to improve it with parts of the multilevel paradigm. It is also important to note that `mlv-wsat1.0's` clusters always have a common truth value among their variables, compared to `mlv-wsat2.0` which might have differing truth values in their clusters, due to the fact that single variables are chosen on the greedy and best move.

6.4 Experiment - benchmarking of multilevel WalkSAT

In the following experiment we evaluate the performance of the multilevel variants of `WalkSAT` against `WalkSAT`. The number of levels is set to 4. Further, variables per clusters is set to 2, and the multilevel-multiplier is set to 10. The noise parameter for both `WalkSAT` and its multilevel variants is set to 10%. The time limit is set to 1800 seconds (30 minutes) per problem. Table 6.2 lists the problems that are used in the experiment.

Problem instance	**#Variables**	**#Clauses**
ac97_ctrl-debug.dimacs.cnf	1 058 921	168 471
spi-debug.dimacs.cnf	682 609	1 928 296
wb_4m8s3.dimacs.filtered.cnf	463 080	1 759 150
rsdecoder2.dimacs.filtered.cnf	415 480	1 632 526
wb-debug.dimacs.cnf	399 591	621 323
mem_ctrl-debug.dimacs.cnf	381 721	505 547
wb-problem.dimacs_45.filtered.cnf	309 491	806 440
rsdecoder1_blackbox_CSEEblock-problem. dimacs_32.filtered.cnf	277 950	806 460
rsdecoder_fsm1.dimacs.filtered.cnf	238 290	238290
SM_RX_TOP.dimacs.filtered.cnf	235 456	934 091
divider-problem.dimacs_3.filtered.cnf	216 900	711 249
divider-problem.dimacs_11.filtered.cnf	215 964	709 377
c5_DD_s3_f1_e1_v1-bug-gate-0.dimacs.seq. filtered.cnf	200 944	540 984
fpu8-problem.dimacs_24.filtered.cnf	160 232	548 848
fpu_fsm1-problem.dimacs_15.filtered.cnf	160 200	548 843
spi2.dimacs.filtered.cnf	124 260	515 813
dividers_multivec1.dimacs.filtered.cnf	106 128	397 650
i2c_master1.dimacs.filtered.cnf	82 429	285 987
dividers6_hack.dimacs.filtered.cnf	35 376	132 699

Table 6.2: Problem instances used in the experiment, 20 in total. Listed with number of variables and clauses.

6.4.1 Results

Below we present six figures for `WalkSAT` and the four `multilevel WalkSAT` variants presented in this chapter. Each figure shows the number of unsatisfied clauses over time for a specific problem instance. Note that the horizontal axis, time, is given in logarithmic scale. After the figures, we show a table with a selection of results. More results are available in appendix A.1.

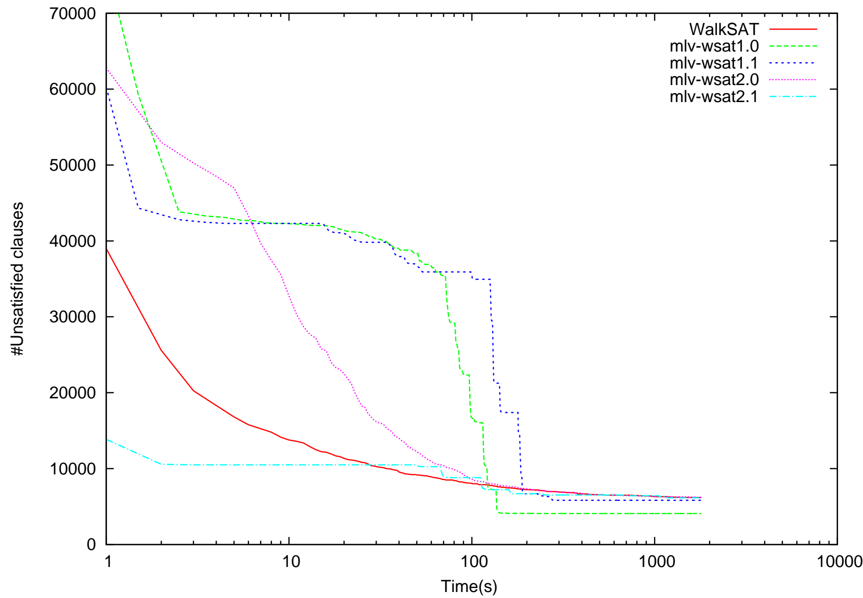

Figure 6.4: Log plot: Problem instance divider-problem.dimacs_3.filtered.cnf, |variables| = 216 900, |clauses| = 711 249. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds.

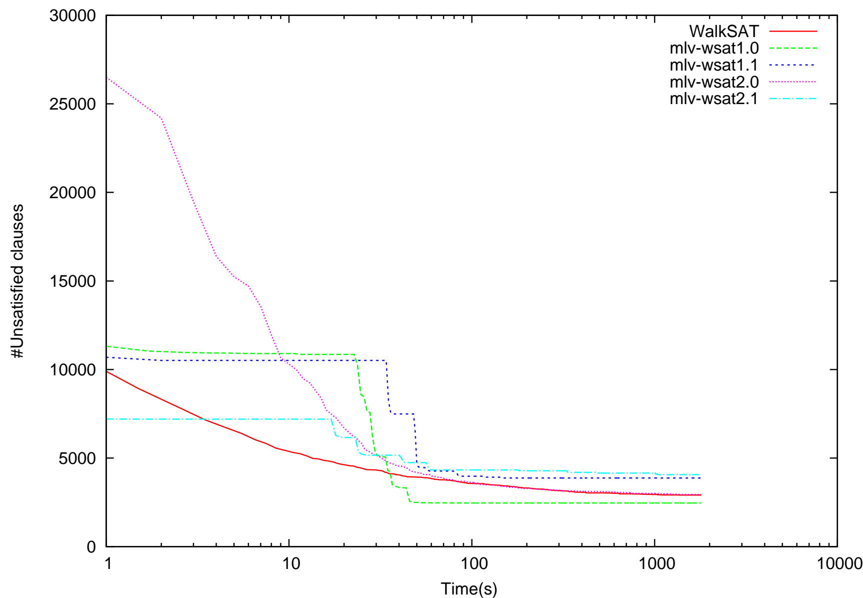

Figure 6.5: Log plot: Problem instance dividers_multivec1.dimacs.filtered.cnf, |variables| = 106 128, |clauses| = 397 650. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds.

In figures 6.4 and 6.5 all the multilevel algorithms, except `mlv-wsat2.1`, have a high number of unsatisfied clauses at the beginning, while `WalkSAT` and `mlv-wsat2.1` have a low number. Note that

the multilevel variants start at level 4, and therefore will have less possible variable assignments than `WalkSAT`. Hence these variants might have a higher number of unsatisfied clauses than `WalkSAT` at the beginning.

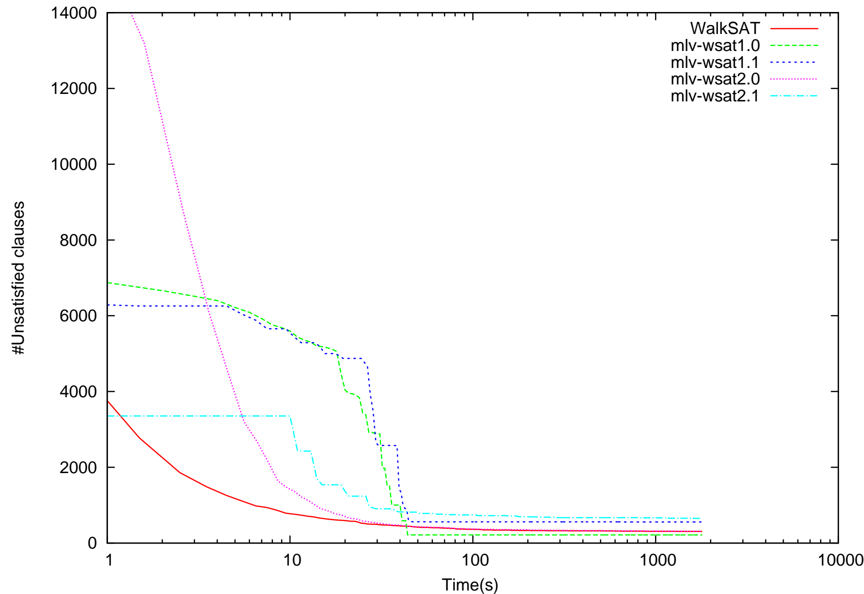

Figure 6.6: Log plot: Problem instance i2c_master1.dimacs.filtered.cnf, |variables| = 82 429, |clauses| = 285 987. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds.

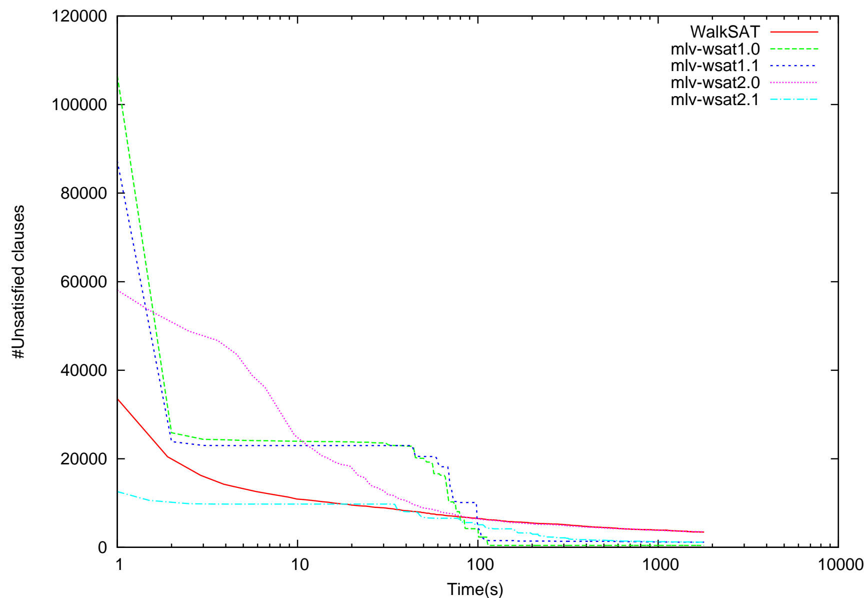

Figure 6.7: Log plot: Problem instance rsdecoder1_blackbox_CSEEblock-problem.dimacs_32. filtered.cnf, |variables| =277 950, |clauses| = 806 460. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds.

We also observe two multilevel characteristics: breaks and early convergence. First, the breaks in unsatisfied clauses. As an example, looking at figure 6.6 at time = $[1, 60]$ we see that all `multilevel WalkSAT` variants stabilize on a plateau and then drop to a new plateau and so on. These steep drops are consequences of going from a lower level to a higher level. In figure 6.8 we have plotted `mlv-wsat1.1` with level markers that denote going from one level to the other. The first blue line

that appears at time = 10 shows `mlv-wsat1.1` going from the bottom level, namely level 4, to level 3. Experiments have shown that going from one level to another does not necessarily mean a drop in unsatisfied clauses, however, at the higher levels a drop is almost certain, as can be seen from the two last vertical lines. The last vertical line represents the transition from level 1 to level 0.

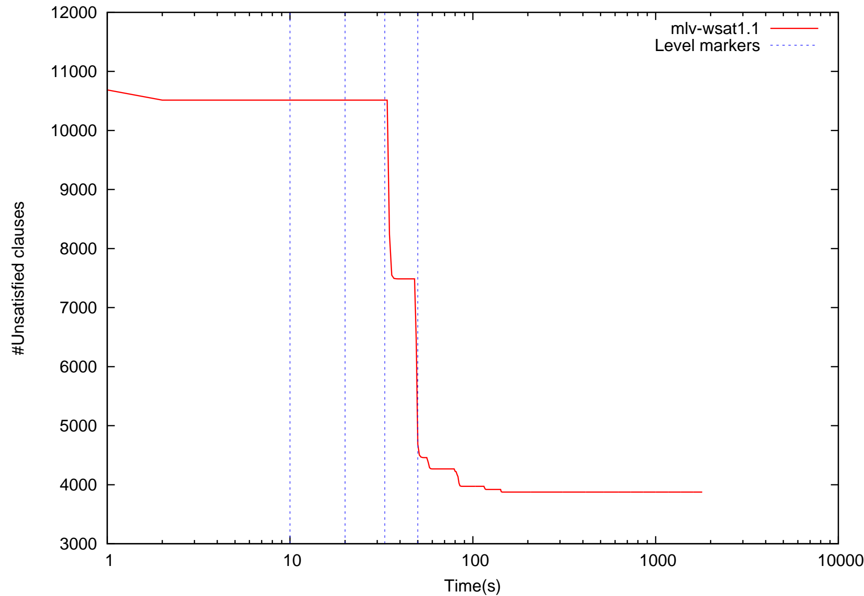

Figure 6.8: Log plot: Problem dividers_multivec1.dimacs.filtered.cnf, |variables| = 106 128, |clauses| = 397 650. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds. The vertical bars indicate level transitions.

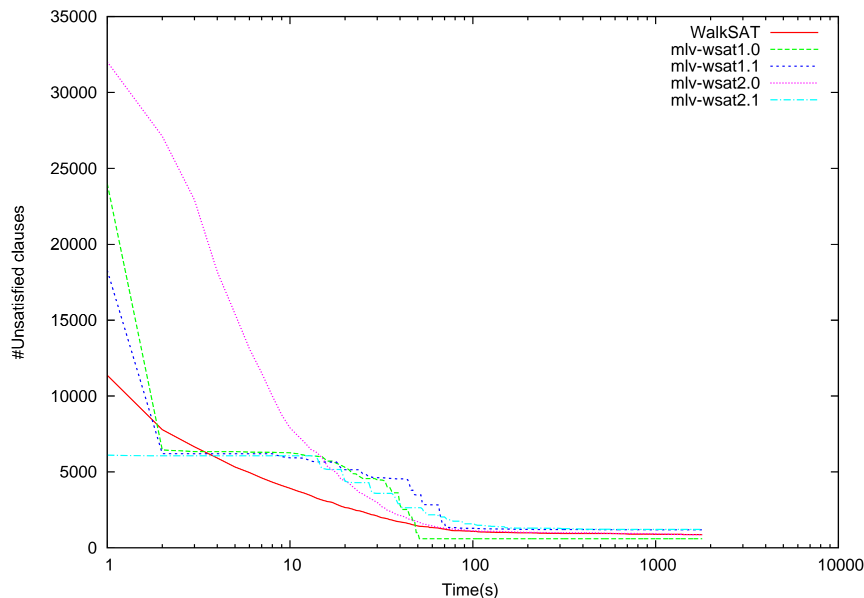

Figure 6.9: Log plot: Problem instance spi2.dimacs.filtered.cnf, |variables| = 124 260, |clauses| = 515 813. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds.

Second, we see that `WalkSAT` tends to converge later than `mlv-wsat1.0` and `mlv-wsat1.1`. This is evident in all figures, and in figure 6.7 and 6.9 we see that convergence for `mlv-wsat1.0` start at time = 110 and time = 50 respectively.

Comparing `mlv-wsat2.0` with static noise and `mlv-wsat2.1` with dynamic noise, we notice that `mlv-wsat2.1` performs better in the beginning on the selected problems above. However, in the end `mlv-wsat2.0` seems to deliver better results.

From table 6.3 we see that `mlv-wsat1.0` performs better than `WalkSAT` on all the selected problems, in terms of unsatisfied clauses. It also seems more reliable than the other multilevel variants. We also see that `mlv-wsat2.0` seems to achieve the same result as `WalkSAT`.

On the problem named wb-debug.dimacs.cnf the dynamic noise variants, `mlv-wsat1.1` and `mlv-wsat2.1`, dominate. We believe this can either mean that the other variants should have been given another static noise, or that this problem in particular benefits from a variable noise.

In summary, one out of the four `multilevel WalkSAT` variants performs better than `WalkSAT` on all problems.

Problem instance	Solver	#Mean unsatisfied clauses	Standard deviation	Variance
rsdecoder_fsm1. dimacs.filtered.cnf	WalkSAT	271.20	18.76	351.96
	mlv-wsat1.0	185.70	19.19	368.23
	mlv-wsat1.1	405.30	83.22	6926.23
	mlv-wsat2.0	267.50	7.17	51.39
	mlv-wsat2.1	1428.20	172.45	29737.73
rsdecoder2.dimacs. filtered.cnf	WalkSAT	671.80	14.22	202.18
	mlv-wsat1.0	442.50	41.23	1699.61
	mlv-wsat1.1	1002.50	211.72	44825.83
	mlv-wsat2.0	667.50	17.92	321.17
	mlv-wsat2.1	2904.00	83.19	6920.00
divider-problem. dimacs_11.filtered.cnf	WalkSAT	5948.50	154.48	23864.72
	mlv-wsat1.0	4616.30	900.43	810767.79
	mlv-wsat1.1	5491.50	68.36	4672.50
	mlv-wsat2.0	6004.00	129.66	16812.00
	mlv-wsat2.1	6163.30	273.27	74679.12
dividers6_hack. dimacs.filtered.cnf	WalkSAT	426.70	19.83	393.34
	mlv-wsat1.0	412.50	22.65	513.17
	mlv-wsat1.1	991.80	27.93	779.96
	mlv-wsat2.0	432.40	13.89	192.93
	mlv-wsat2.1	979.90	43.67	1906.77
fpu_fsm1-problem. dimacs_15.filtered.cnf	WalkSAT	3690.70	54.52	2972.01
	mlv-wsat1.0	3284.00	715.02	511256.00
	mlv-wsat1.1	2706.60	113.24	12822.27
	mlv-wsat2.0	3686.30	67.00	4489.34
	mlv-wsat2.1	2949.80	69.61	4845.29
wb-debug.dimacs.cnf	WalkSAT	314.20	24.43	597.07
	mlv-wsat1.0	264.10	13.40	179.66
	mlv-wsat1.1	121.50	12.78	163.39
	mlv-wsat2.0	322.00	23.47	550.89
	mlv-wsat2.1	107.10	15.56	242.10
fpu8-problem. dimacs_24.filtered.cnf	WalkSAT	3608.10	53.66	2879.21
	mlv-wsat1.0	2822.90	629.11	395776.32
	mlv-wsat1.1	2737.50	210.93	44493.17
	mlv-wsat2.0	3560.40	70.42	4958.71
	mlv-wsat2.1	2845.20	150.06	22518.40
i2c_master1.dim acs.filtered.cnf	WalkSAT	306.00	14.48	209.56
	mlv-wsat1.0	214.20	26.89	723.07
	mlv-wsat1.1	557.50	114.32	13068.50
	mlv-wsat2.0	309.80	15.30	234.18
	mlv-wsat2.1	650.20	29.73	883.96

Table 6.3: Results from the experiment. Multilevel WalkSAT variants and WalkSAT.

Chapter 7

Combining WalkSAT with Learning Automata

In this chapter we add learning capabilities to `WalkSAT`. More specifically, we extend `WalkSAT` with a branch of machine learning called Learning Automata.

We first give a brief introduction to Learning Automata and reinforcement learning theory in section 7.1 where essential concepts are given. Further, the employment of Learning Automata in a MAXSAT context is put forward in section 7.2. In section 7.3 we present our solution for combining `WalkSAT` with Learning Automata. We end this chapter with experiments and results in section 7.4 and 7.5.

7.1 Learning Automata and reinforcement learning

Originally known as a Tsetlin automaton [40], but in later years as a Learning automaton (singular form of Learning Automata) by Narendra and Thathachar in [41]. It is a branch of machine learning that can be interpreted as a finite state automaton[1]. A Learning automaton is a type of a reinforcement learning agent that is situated in an unknown environment [42].

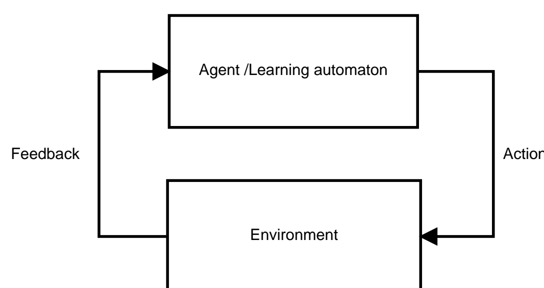

Figure 7.1: Interaction between an agent and its environment.

The agent's goal is to maximize its gain (e.g. points in a game or winning as much as possible). This goal is pursued by performing beneficial actions on the current environment. This is made possible through feedback from the environment, see figure 7.1. The feedback (which may be stochastic) consists of a reward value and the new state of the environment. To be able to adapt to the environment the agent has an internal memory. The agent uses this memory together with the feedback to decide the next actions. This can also be seen as a learning process. Reinforcement learning agents are different in the way they see the environment, and how they decide what

[1]An automaton can be thought of as an abstract self-running machine.

actions are beneficial for their maximal gain. Depending on the complexity of the environment, reinforcement learning agents have to go through several iterations to be able to maximize its gain. This also applies to Learning Automata.

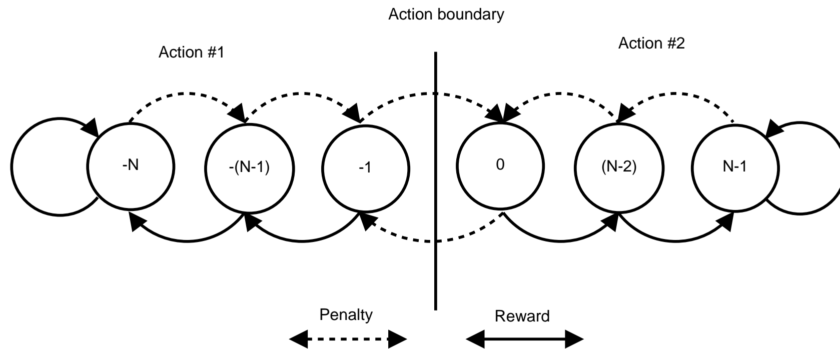

Figure 7.2: Learning automaton states, actions, rewards and penalties, and their effects.

In figure 7.2 we can see a Learning automaton that has two actions $A = \{a_1, a_2\}$ and N states per action. The states N can be seen as the agent's internal memory. This is the only place the Learning automaton learns to favor a specific action. If the state $n \in N$ of the Learning automaton is on the left of the action boundary it chooses a_1, else it chooses action a_2. The Learning automaton is initialized to state -1 or 0 at random. From the very first action the Learning automaton performs, the environment will give it the reward value specified according to the environment's current state and its next state. The reward value that is given by the environment takes the form of either rewarding or punishing the Learning automaton. Looking at figure 7.2 we see that punishing the automaton might cause a crossing of the action boundary. At this crossing, the automaton changes its action. By rewarding the automaton the confidence that the action is beneficial is strengthened. In essence, using such a two-action Learning automaton effectively dampens the possibility of choosing a different action if the confidence is high and vice versa.

Learning Automata has been applied in a variety of fields and domains such as wireless networks [43], vehicle path control [44] and routing algorithm for mobile networks [45]. We will in the next section see how Learning Automata can be an asset for solving SAT and MAXSAT.

7.2 Learning Automata in a MAXSAT context

The following section describes the framework by Granmo and Bouhmala for merging Learning Automata (LA) with SAT solvers in [7, 22]. We assume the reader understands that the following section can also be applied to MAXSAT. One important thing to notice is that LA never solve a SAT problem alone. LA are always combined with an algorithm or a method.

Learning Automata has been adopted to SAT in the following way: A SAT problem has a variable set containing truth values e.g. $V = \{v_1 = FALSE, v_2 = TRUE, ..., v_{n-1} = FALSE, v_n = FALSE\}$. Instead of the latter, we now look at V as a set of variables, where each variable is associated with a Learning automaton (La) such that $V = \{La_1, La_2, La_3, ..., La_n\}$ with actions $A = \{TRUE, FALSE\}$. In a MAXSAT context the environment can be thought of as being the MAXSAT problem, but is also influenced by the solver. The truth values are now specified by a Learning automaton's action at every variable, instead of the truth value itself. Every $La \in V$ is also set to an initial state of -1 or 0 at random as described in section 7.1. Each Learning automaton's state dictates the action the Learning automaton would choose, either $TRUE$ or $FALSE$. The maximal gain for each Learning automaton is to minimize the number of unsatisfied clauses where its variable appears. In the bigger picture the maximal gain for the total system of multiple Learning Automata is to minimize the number of unsatisfied clauses in the SAT problem. Since there might be multiple Learning Automata sharing the same clauses, due to the fact that variables occur in the same clauses, the Learning Automata will have to compete against each other.

The SAT solver works like before it was combined with Learning Automata, but instead of flipping each of the variables in V itself, it either rewards or penalizes every $La \in V$ such that it is the La that decides whether to flip or not. In turn, this ensures that each variable's value is equivalent to the La's action. Where flipping means the negation of a truth variable in a SAT context, flipping means going across the action boundary in a Learning Automata context. Thus, the reward and penalty mechanisms in respect to MAXSAT has the following properties: If the La's action, which also corresponds to the variable's truth value, contributes to the quality of the solution (i.e. less unsatisfied clauses) then the La is rewarded, and vice versa.

In summary, each variable is governed by a Learning automaton that strengthens the confidence of the variable's truth value. Since the overall variable assignment for the entire SAT problem depends on all LA, each La is indirectly coupled to each other.

7.3 Learning Automata WalkSAT

The combination of `WalkSAT` with Learning Automata follows the same scheme as normal `WalkSAT`, but with some modifications. The first step is to initialize the LA and generate a random state assignment to each Learning automaton. In turn, this state results in the variable's truth value as mentioned earlier. An outline of the algorithm is given in algorithm[2] 6.

Algorithm 6 Learning Automata WalkSAT

```
1: {Input: CNF formula G}
2: {Output: Satisfied assignment for G, or 'no solution exists'}
3: {La_variable returns the Learning automaton associated with variable}
4: Initialization: p_noise ∈ [0, 1]
5: for i ← 0 to MAX_TRIES do
6:   /* The following also indirectly initializes the variable assignment */
7:   T ← random_LA_state_assignment()
8:   for j ← 0 to MAX_FLIPS do
9:     if T is satisfied then
10:      return T
11:    end if
12:    C_k ← random_unsatisfied_clause()
13:    /* punish section */
14:    /* best move/side move */
15:    if ∃ variable v ∈ C_k with break-count = 0 then
16:      variable ← v
17:      state_mirror(La_variable)
18:    else
19:      if random(0, 1) < p_noise then
20:        /* random move */
21:        variable ← random_variable(C_k)
22:        punish(La_variable)
23:      else
24:        /* greedy move */
25:        variable ← random_lowest_breakcount_variable(C_k)
26:        if solver = la-wsat1.0 then
27:          state_mirror(La_variable)
28:        else if solver = la-wsat1.1 then
29:          punish(LA_variable)
30:        end if
31:      end if
32:    end if
33:    if La_variable crossed action boundary then
34:      flip(variable)
35:    end if
36:    /* reward section */
37:    C_s ← random_satisfied_clause()
38:    variable ← random_variable(C_s)
39:    reward(La_variable)
40:  end for
41: end for
```

[2]Like for the algorithm for WalkSAT, the algorithm for Learning Automata WalkSAT is also given in its SAT form. To solve MAXSAT problems slight modifications are needed, see section 5.2.

We have experimented with two types of implementations of `Learning Automata WalkSAT` called `la-wsat1.0` and `la-wsat1.1`. These are given in table 7.1. Each algorithm has a best move, random move and greedy move like regular `WalkSAT`. After each of the standard moves, a satisfied clause C_s is selected at random. A random variable v_s is picked from C_s and its corresponding LA_{v_s} is rewarded.

Algorithm	**Greedy move operation**	**Random move operation**	**Best move operation**
la-wsat1.0	Mirroring	Punish	Mirroring
la-wsat1.1	Punish	Punish	Mirroring

Table 7.1: Learning Automata WalkSAT variants and their differences.

Like `WalkSAT`, both versions of `Learning Automata WalkSAT` start by picking an unsatisfied clause C at random. Further, K is defined as those variables in C with the lowest common break-count[3]. If the break-count of $K = 0$, which accords to the best move of `WalkSAT`, then a variable v selected at random from K is flipped and the corresponding La_v's state is mirrored about the action boundary according to figure 7.3 on the left side. Alternatively, La_v's state could be punished or rewarded as mentioned in the previous section. However, since the immediate flipping on the best move is crucial for the performance of `WalkSAT`, we apply this state mirroring mechanism. Without this operation, there would be no guarantee that the selected variable v would be flipped due to the state transitions of the Learning automaton LA_v. As seen in table 7.1 both algorithms have mirroring on their best move. If there is no variable with break-count = 0 then either the random move or greedy move is selected according to regular `WalkSAT`. Depending on the algorithm, either punish or mirroring is selected. Punish and reward is performed as described in section 7.2.

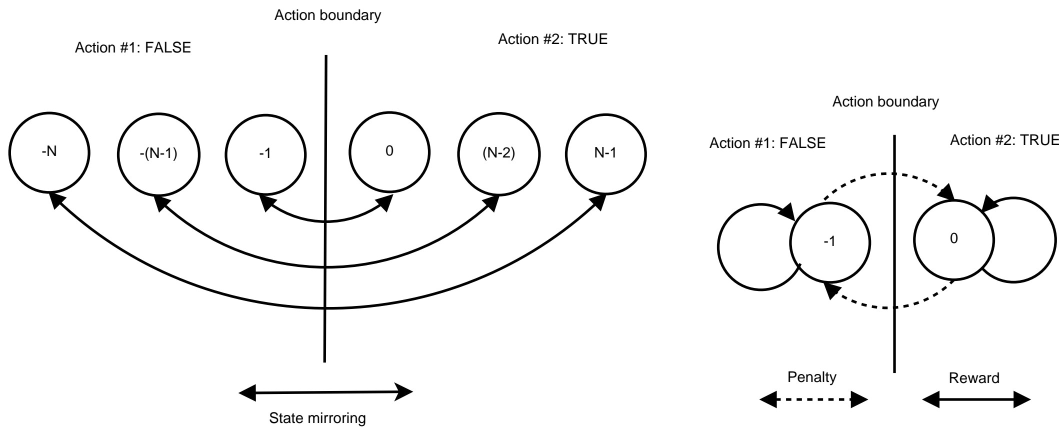

Figure 7.3: Left: Learning automaton state mirroring mechanism. Right: Learning Automata WalkSAT with one state per action.

Note that `Learning Automata WalkSAT` with 1 state per action is equivalent to `WalkSAT`, as can be seen from figure 7.3 on the right side. This is because the action boundary would always be crossed at any punish operation. The reward mechanism would not have any affect due to that the La's state transition will loop on the same state and not advance to another state when rewarded.

[3]Break-count as given in definition 5.

7.4 Experiment - Learning Automata state evaluation

In the following experiment we explore the number of states associated with the Learning automaton to see how this affects the performance. The importance of this experiment is to find the optimal common number of states for the Learning Automata. The optimal number of states is defined as the number of states that gives the least number of unsatisfied clauses in a given problem when the algorithm has finished executing. The optimal number of states for `la-wsat1.0` and `la-wsat1.1` found in this experiment will be used in other experiments throughout this paper.

The number of states for `la-wsat1.0` and `la-wsat1.1` is the set $S_{states} = \{2, 3, 4, 5, 10\}$. We run all the algorithms with 10% noise, and all algorithms have an execution time of 1800 seconds. Table 7.2 lists the problems that are used in the experiment.

Problem instance	**#Variables**	**#Clauses**
rsdecoder2.dimacs.filtered.cnf	415 480	1 632 526
rsdecoder1__blackbox__CSEEblock-problem.dimacs__32.filtered.cnf	277 950	806 460
SM__RX__TOP.dimacs.filtered.cnf	235 456	934 091
divider-problem.dimacs__3.filtered.cnf	216 900	711 249
fpu8-problem.dimacs__24.filtered.cnf	160 232	548 848
spi2.dimacs.filtered.cnf	124 260	515 813
dividers__multivec1.dimacs.filtered.cnf	106 128	397 650
i2c__master1.dimacs.filtered.cnf	82 429	285 987

Table 7.2: Problem instances used in the experiments, 8 in total. Listed with number of variables and clauses.

7.4.1 Results

Figure 7.4 shows how the number of unsatisfied clauses increases when increasing number of states for both `la-wsat1.0` and `la-wsat1.1`.

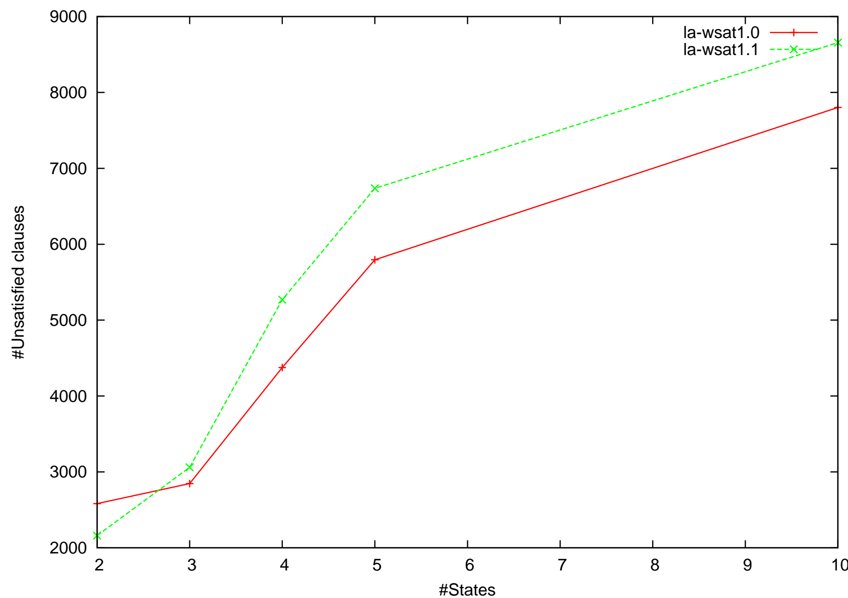

Figure 7.4: Problem instance divider-problem.dimacs_3.filtered.cnf. |variables| = 216 900, |clauses| = 711 249. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the number of states per Learning automaton.

In the figure below we have a situation where `la-wsat1.0` has the lowest number of unsatisfied clauses when the number of states is equal to 3, while `la-wsat1.1` has its optimal solution at 2 states.

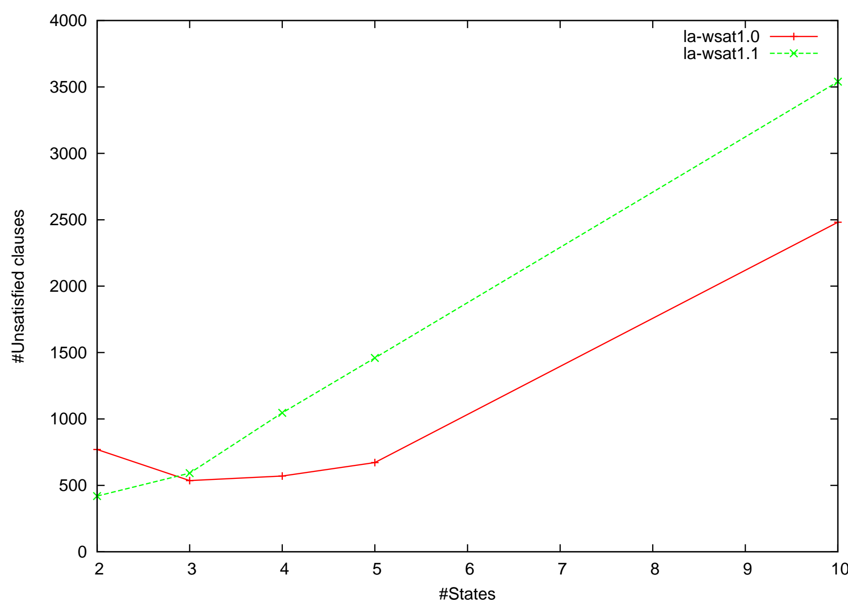

Figure 7.5: Problem instance fpu8-problem.dimacs_24.filtered.cnf. |variables| = 160 232, |clauses| = 548 848. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the number of states per Learning automaton.

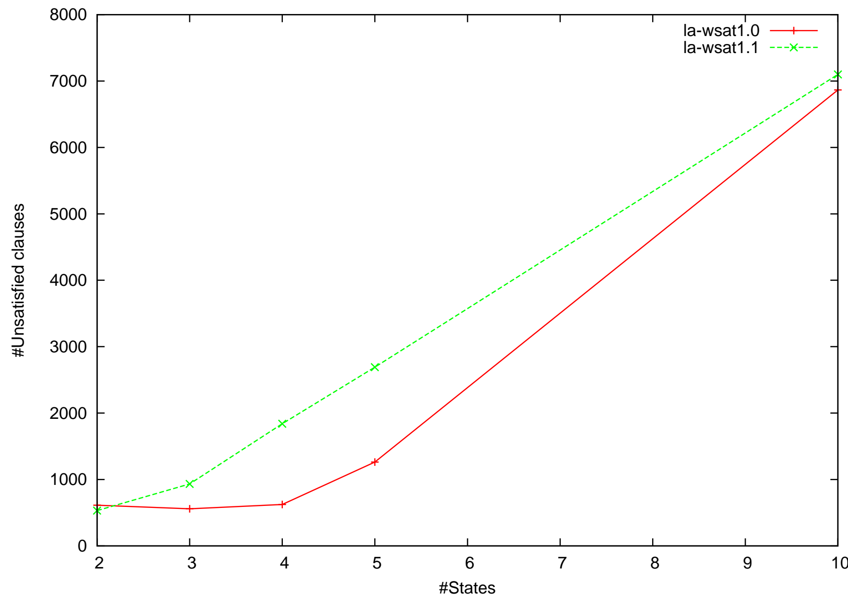

Figure 7.6: Problem instance SM_RX_TOP.dimacs.filtered.cnf. |variables| = 235 456, |clauses| =934 091. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the number of states per Learning automaton.

In figure 7.6 we also observe that the optimal number of states for `la-wsat1.0` is 3, and 2 states for `la-wsat1.1`. The remaining results are given in the table below.

Problem instance	Solver	#Optimal states	#Mean unsatisfied clauses
spi2.dimacs.filtered.cnf	la-wsat1.0	3	282
	la-wsat1.1	2	208
rsdecoder2.dimacs.filtered.cnf	la-wsat1.0	2	1 376
	la-wsat1.1	2	1 445
divider-problem.dimacs_3.filtered.cnf	la-wsat1.0	2	2 582
	la-wsat1.1	2	2 160
fpu8-problem.dimacs_24.filtered.cnf	la-wsat1.0	3	536
	la-wsat1.1	2	419
dividers_multivec1.dimacs.filtered.cnf	la-wsat1.0	3	339
	la-wsat1.1	2	265
i2c_master1.dimacs.filtered.cnf	la-wsat1.0	3	108
	la-wsat1.1	2	95
rsdecoder1_blackbox_CSEEblock-problem.dimacs_32.filtered.cnf	la-wsat1.0	2	2252
	la-wsat1.1	2	1879
SM_RX_TOP.dimacs.filtered.cnf	la-wsat1.0	3	559
	la-wsat1.1	2	531

Table 7.3: Results from the experiment. Problem instances given with optimal number of states for both Learning Automata WalkSAT variants.

The results presented in table 7.3 imply that the optimal number of states for `la-wsat1.1` is 2. For `la-wsat1.0` the optimal number of states varies between 2 and 3. However, 5 out of the 8

problems, $\approx 62\%$, favor 3 states as optimal. Therefore we conclude that 3 is the optimal number of states in general for `la-wsat1.0`, and that 2 is the optimal number of states for `la-wsat1.1`.

7.5 Experiment - benchmarking of Learning Automata WalkSAT

In this experiment we benchmark our proposed variants of `WalkSAT` combined with Learning Automata against `WalkSAT`. The noise parameter for all the algorithms is set to 10% and the time limit is set to 1800 seconds. Number of states per Learning automaton is set to 3 for `la-wsat1.0` and 2 for `la-wsat1.1` (see experiment 7.4). We use the same problem suite, table 6.2, which was used in the experiment in chapter 6.

7.5.1 Results

The following figures present results from benchmarking on `WalkSAT`, `la-wsat1.0` and `la-wsat1.1`. Note that all solvers have the same initialization procedure, meaning the differences in unsatisfied clauses at time = 1 is strictly due to solver behavior.

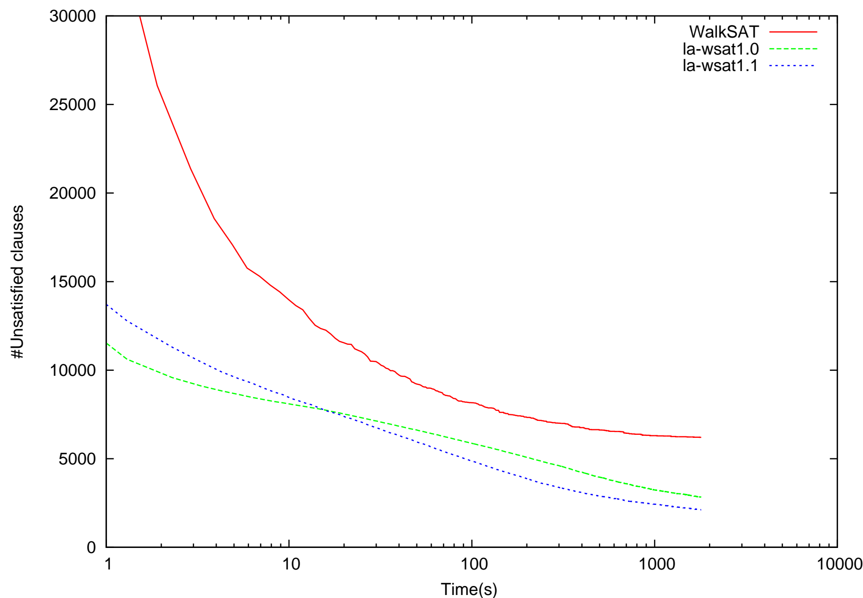

Figure 7.7: Log plot: Problem instance divider-problem.dimacs_3.filtered.cnf |variables| = 216 900, |clauses| = 711 249. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds.

Figure 7.7 and 7.8 reports that the `Learning Automata WalkSAT` algorithms outperforms `WalkSAT`. We see that `la-wsat1.0` and `la-wsat1.1` always lie significantly beneath `WalkSAT`.

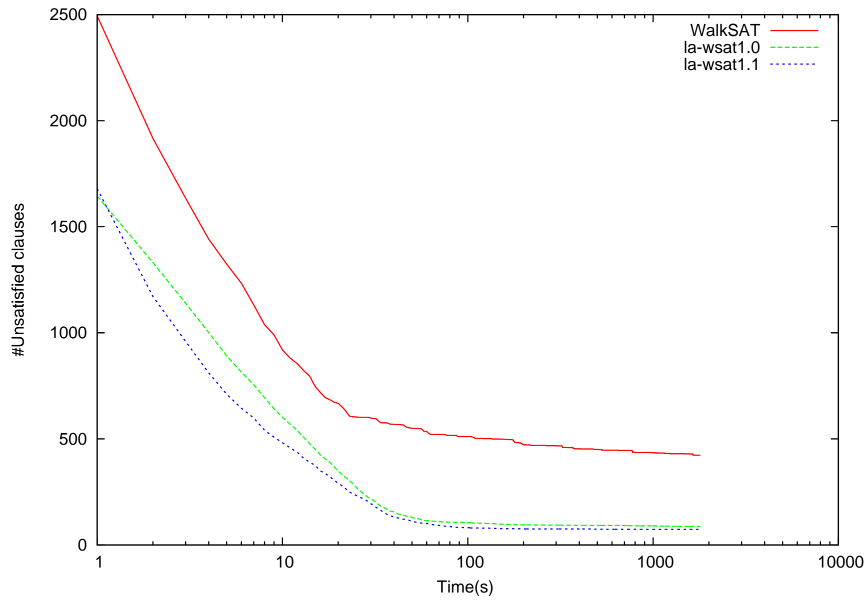

Figure 7.8: Log plot: Problem instance dividers6_hack.dimacs.filtered.cnf |variables| = 35 376, |clauses| = 132 699. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds.

In figure 7.9 we see that the `Learning Automata WalkSAT` algorithms compete with `WalkSAT` in the timespan time $= [10, 100]$. After the timespan, both `la-wsat1.0` and `la-wsat1.1` quickly drop to a lower final value than `WalkSAT`. We acknowledge that `la-wsat1.1` for all problems graphed above always gives better results than both `WalkSAT` and `la-wsat1.0`.

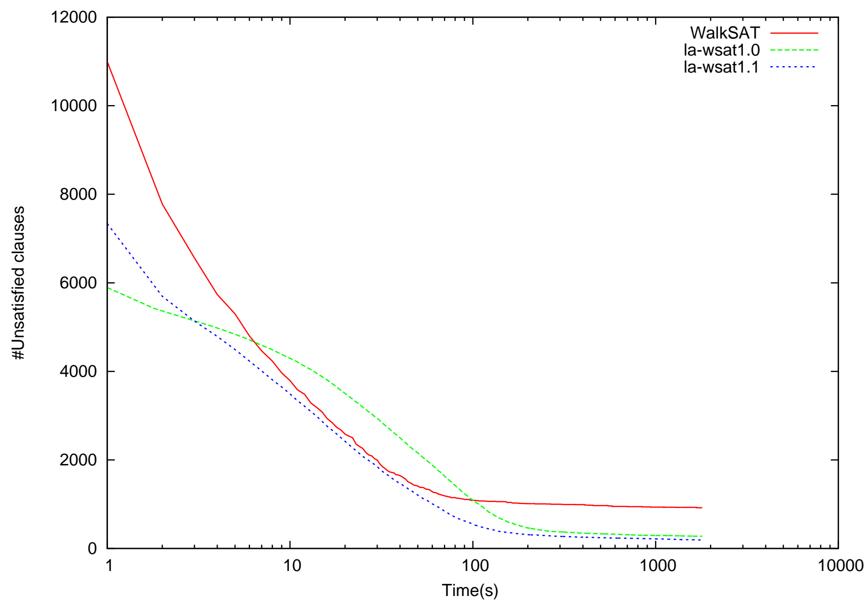

Figure 7.9: Log plot: Problem instance spi2.dimacs.filtered.cnf |variables| = 124 260, |clauses| = 515 813. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds.

Remaining results in table 7.4 show that both `Learning Automata WalkSAT` variants perform better than `WalkSAT` on almost all problems. Further, `la-wsat1.1` yields better results than `la-wsat1.0`. For more results see appendix A.2.

Problem instance	**Solver**	**#Mean unsatisfied clauses**	**Standard deviation**	**Variance**
spi2.dimacs.filtered.cnf	WalkSAT	919.20	39.67	1573.51
	la-wsat1.0	276.80	35.10	1231.96
	la-wsat1.1	191.70	22.69	514.68
wb_4m8s3.dimacs. filtered.cnf	WalkSAT	2675.70	71.38	5094.68
	la-wsat1.0	784.20	45.61	2080.40
	la-wsat1.1	285.20	25.18	634.18
rsdecoder2.dimacs. filtered.cnf	WalkSAT	669.80	16.63	276.62
	la-wsat1.0	1746.80	235.36	55396.62
	la-wsat1.1	1381.10	160.94	25900.10
i2c_master1.dimacs. filtered.cnf	WalkSAT	317.20	11.69	136.62
	la-wsat1.0	102.70	23.30	543.12
	la-wsat1.1	95.40	9.55	91.16
SM_RX_TOP.dimacs. filtered.cnf	WalkSAT	2900.60	113.65	12917.38
	la-wsat1.0	556.70	47.12	2220.23
	la-wsat1.1	550.50	35.32	1247.17
rsdecoder1_ blackbox_ CSEEblock-problem. dimacs_32.filtered.cnf	WalkSAT	3479.10	275.96	76156.32
	la-wsat1.0	2270.70	72.20	5212.90
	la-wsat1.1	1868.60	90.78	8240.27
divider-problem.dimacs_ 3.filtered.cnf	WalkSAT	6204.80	81.33	6613.96
	la-wsat1.0	2832.50	114.72	13161.39
	la-wsat1.1	2117.00	71.94	5175.33
spi-debug.dimacs.cnf	WalkSAT	1286.10	33.87	1147.43
	la-wsat1.0	287.40	35.78	1280.04
	la-wsat1.1	222.20	39.67	1573.51
divider-problem.dimacs_ 11.filtered.cnf	WalkSAT	6030.20	124.94	15611.07
	la-wsat1.0	2583.80	125.90	15851.07
	la-wsat1.1	1918.20	63.80	4071.07
dividers6_hack.dimacs. filtered.cnf	WalkSAT	423.20	25.42	645.96
	la-wsat1.0	87.10	8.17	66.77
	la-wsat1.1	73.60	6.79	46.04
dividers_multivec1. dimacs.filtered.cnf	WalkSAT	2953.90	30.67	940.54
	la-wsat1.0	340.60	26.33	693.16
	la-wsat1.1	250.50	24.08	580.06
fpu_fsm1-problem. dimacs_15.filtered.cnf	WalkSAT	3716.40	49.54	2454.49
	la-wsat1.0	499.60	26.26	689.60
	la-wsat1.1	422.70	44.16	1950.01
fpu8-problem.dimac s_24.filtered.cnf	WalkSAT	3585.90	61.94	3836.99
	la-wsat1.0	535.80	34.34	1179.29
	la-wsat1.1	412.50	35.87	1286.72

Table 7.4: Results from experiment. Learning Automata WalkSAT variants and WalkSAT.

Chapter 8

Combining WalkSAT with Multilevel Techniques and Learning Automata

In the preceding chapters we proposed methods for combining `WalkSAT` with multilevel techniques and Learning Automata, seperately. We now add both multilevel techniques and Learning Automata to `WalkSAT` at the same time.

The building blocks of `multilevel Learning Automata WalkSAT` are given in section 8.1. In section 8.2 we perform benchmarking of the proposed algorithm together with a selection of the other algorithms presented in this thesis.

8.1 Multilevel Learning Automata WalkSAT

The `multilevel Learning Automata WalkSAT` algorithm is completed by joining the best performing `multilevel WalkSAT` variant and `Learning Automata WalkSAT` variant. More specifically, the featured variants are `mlv-wsat1.0` and `la-wsat1.1`. This results in the algorithm `mlv-la-wsat`. The building blocks of `mlv-la-wsat` are listed in table 8.1, and the elements can be recognized from chapter 6 and 7. The three different moves, greedy move, random move and best move come from the structure of `WalkSAT` as presented in chapter 5.

Domain →	Learning Automata			Multilevel techniques		
Properties →	**Greedy move operation**	**Random move operation**	**Best move operation**	**Greedy move flip method**	**Best move flip method**	**Random move flip method**
Values →	Punish	Punish	Mirroring	Cluster	Cluster	Cluster

Table 8.1: Multilevel Learning Automata WalkSAT building blocks.

Instead of having a Learning automaton per variable, we now have a Learning automaton per cluster. For each level in the projection phase, every cluster is split into two smaller clusters. These two clusters inherit/clone the Learning automaton state of the parent cluster. After the projection from level 1 to level 0, each cluster is split to variables and inherits the Learning automaton state and the algorithm corresponds to `la-wsat1.1`.

8.2 Experiment - benchmarking of multilevel Learning Automata WalkSAT

In this experiment we perform benchmarking of `mlv-la-wsat` together with `WalkSAT`, `mlv-wsat1.0` and `la-wsat1.1`. The noise is set to 10% for all algorithms. `mlv-wsat1.0` and `mlv-la-wsat` have 4 levels, 2 variables per cluster, and the multilevel-multiplier set to 10. `la-wsat1.1` and `mlv-la-wsat` have 2 states per action. The time limit is 1800 seconds, and we use the problems given in table 6.2.

8.2.1 Results

Below we show plots of `WalkSAT`, `mlv-wsat1.0`, `mlv-la-wsat`, `la-wsat1.1` on selected problems from Appendix A.3.

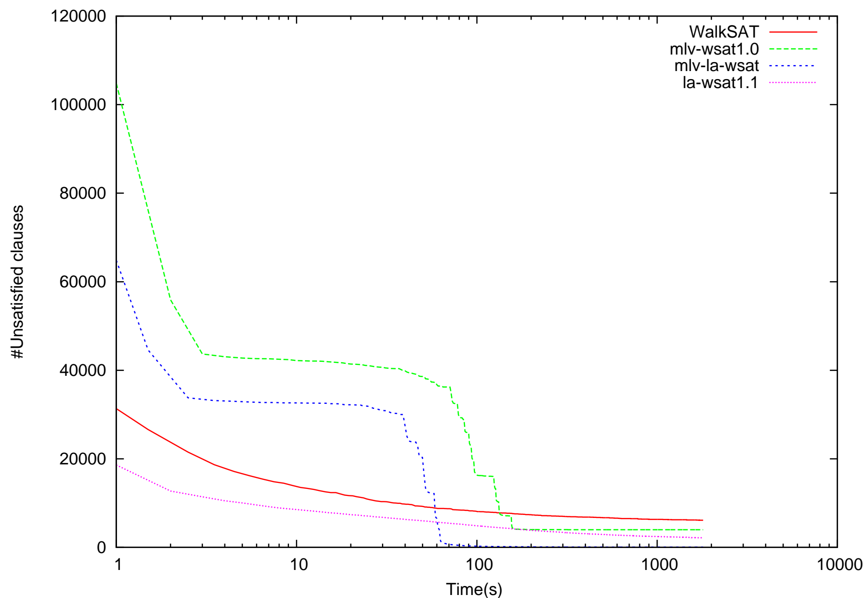

Figure 8.1: Log plot: Problem instance divider-problem.dimacs_3.filtered.cnf |variables| = 216 900, |clauses| = 711 249. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds.

In figures 8.1 and 8.2 we observe the two multilevel variants follow the same trajectory, although `mlv-la-wsat` has a significantly lower number of unsatisfied clauses during the entire search. This also applies for `WalkSAT` and `la-wsat1.1`.

We see that the multilevel characteristics as mentioned in section 6.4 are also present when Learning Automata is added to `mlv-wsat1.0`, because of the visible breaks and early convergence. This is evident from figures 8.3 and 8.4. From the graphs we even see that `mlv-la-wsat` converges earlier than `mlv-wsat1.0`.

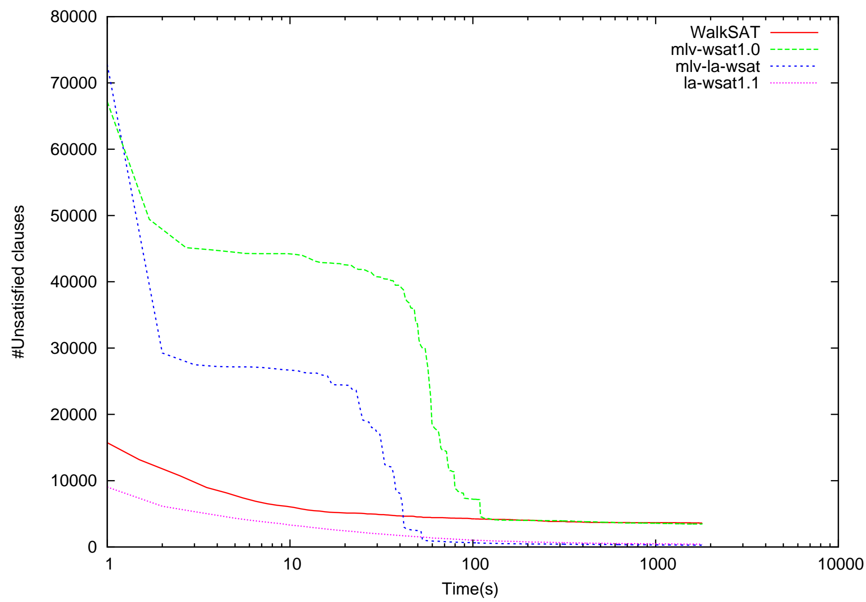

Figure 8.2: Log plot: Problem instance fpu8-problem.dimacs_24.filtered.cnf |variables| = 160 232, |clauses| = 548 848. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds.

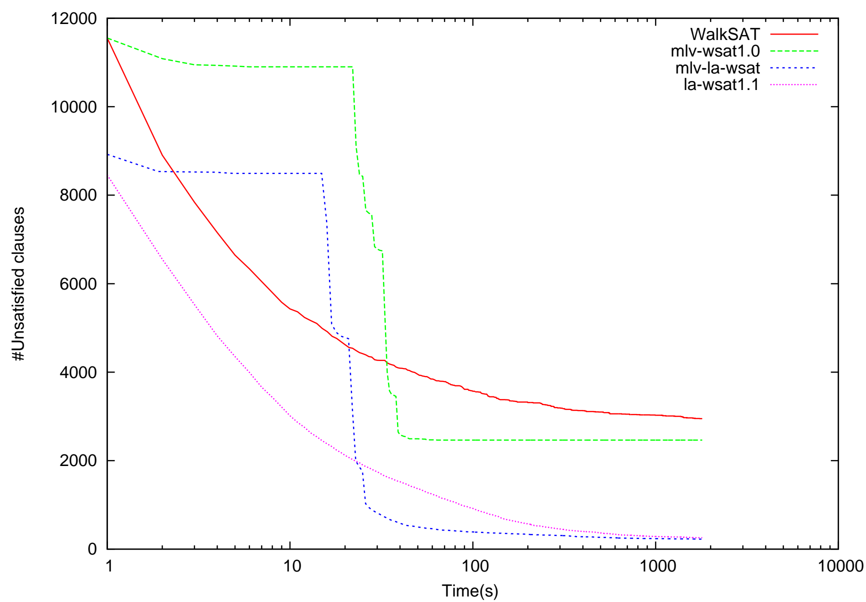

Figure 8.3: Log plot: Problem instance dividers_multivec1.dimacs.filtered.cnf |variables| =106 128, |clauses| = 397 650. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds.

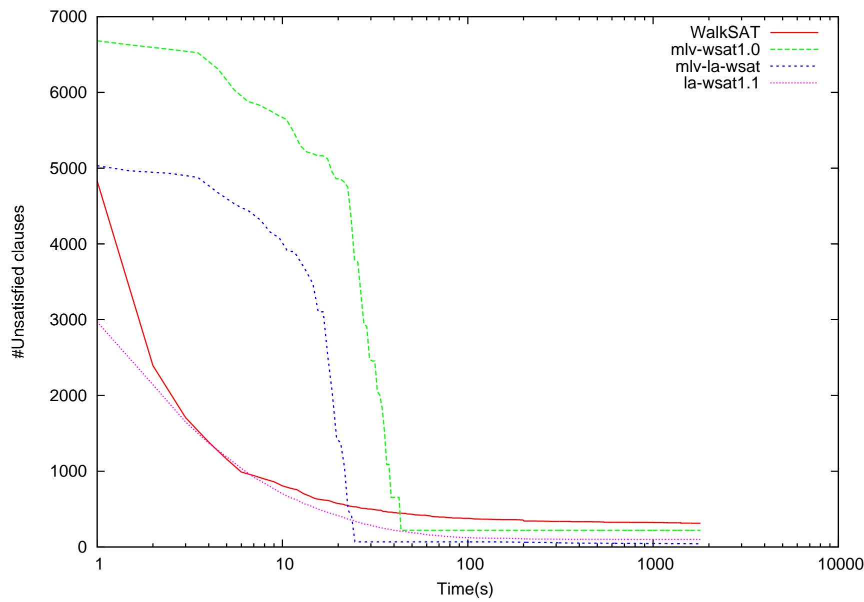

Figure 8.4: Log plot: Problem instance i2c_master1.dimacs.filtered.cnf |variables| = 82 429, |clauses| = 285 987. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds.

In table 8.2 we observe that in almost all cases `mlv-wsat1.0`, `mlv-la-wsat` and `la-wsat1.1` performs better than `WalkSAT`. Further, we see that `la-wsat1.1` achieves better results than `mlv-wsat1.0`. Finally, we acknowledge that `mlv-la-wsat` easily outperforms `la-wsat1.1` on most occasions.

Problem instance	Solver	#Mean unsatisfied clauses	Standard deviation	Variance
spi2.dimacs.filtered.cnf	WalkSAT	896.10	33.65	1132.10
	mlv-wsat1.0	589.10	39.74	1578.99
	mlv-la-wsat	147.40	7.24	52.49
	la-wsat1.1	184.30	13.71	188.01
wb__4m8s3.dimacs.filtered.cnf	WalkSAT	2665.90	77.37	5986.54
	mlv-wsat1.0	2514.00	68.41	4680.22
	mlv-la-wsat	178.20	17.87	319.51
	la-wsat1.1	282.50	28.27	799.17
rsdecoder2.dimacs.filtered.cnf	WalkSAT	666.20	13.85	191.96
	mlv-wsat1.0	444.70	26.49	701.57
	mlv-la-wsat	81.60	1.71	2.93
	la-wsat1.1	1323.50	380.11	144484.72
i2c__master1.dimacs.filtered.cnf	WalkSAT	313.60	15.51	240.71
	mlv-wsat1.0	220.70	29.20	852.90
	mlv-la-wsat	43.30	34.28	1174.90
	la-wsat1.1	99.80	8.73	76.18
SM__RX__TOP.dimacs.filtered.cnf	WalkSAT	2964.70	121.46	14752.01
	mlv-wsat1.0	2589.40	95.81	9180.04
	mlv-la-wsat	312.10	64.73	4189.88
	la-wsat1.1	531.10	31.56	996.10
rsdecoder1__blackbox__CSEEblock-problem.dimacs__32.filtered.cnf	WalkSAT	3561.60	235.88	55637.16
	mlv-wsat1.0	405.00	41.73	1741.78
	mlv-la-wsat	39.70	7.50	56.23
	la-wsat1.1	1832.00	66.54	4427.11
divider-problem.dimacs__3.filtered.cnf	WalkSAT	6143.70	128.13	16417.34
	mlv-wsat1.0	3999.60	98.01	9606.71
	mlv-la-wsat	54.30	28.53	814.23
	la-wsat1.1	2155.40	98.21	9644.71
spi-debug.dimacs.cnf	WalkSAT	1313.10	56.64	3208.54
	mlv-wsat1.0	1266.90	38.68	1496.10
	mlv-la-wsat	98.70	17.11	292.68
	la-wsat1.1	222.80	35.70	1274.40
divider-problem.dimacs__11.filtered.cnf	WalkSAT	6011.50	181.47	32932.72
	mlv-wsat1.0	4758.90	1070.41	1145773.43
	mlv-la-wsat	42.10	20.29	411.88
	la-wsat1.1	1861.10	87.65	7682.54
dividers6__hack.dimacs.filtered.cnf	WalkSAT	431.60	15.79	249.38
	mlv-wsat1.0	407.30	33.52	1123.34
	mlv-la-wsat	53.10	5.65	31.88
	la-wsat1.1	73.40	5.74	32.93

Table 8.2: Results from experiment. Multilevel WalkSAT, Learning Automata WalkSAT, multilevel Learning Automata WalkSAT and WalkSAT.

Chapter 9

Discussion

In this chapter we discuss results from all the experiments performed in this thesis. They will be discussed in the following order. Chapter 6: *Combining WalkSAT with Multilevel Techniques* in section 9.1, chapter 7: *Combining WalkSAT with Learning Automata* in section 9.2 and chapter 8: *Combining WalkSAT with Multilevel Techniques and Learning Automata* in section 9.3. We sum up important findings and observations in section 9.4, where we also link these to the research questions we put forward in the introduction chapter.

9.1 Multilevel WalkSAT

We discuss `multilevel WalkSAT` according to four categories that we see important: multilevel-multiplier, number of levels, single versus cluster and static versus dynamic noise. In general, we acknowledge that `mlv-wsat1.0` performs best of all the proposed multilevel algorithms presented in chapter 6.

9.1.1 Wasting computational resources

Preliminary testing during the benchmark period of `multilevel WalkSAT` showed that we needed a way to control how much time or amount of flips that was needed on each refinement phase, according to the multilevel paradigm as described in 6.2. Without this control mechanism `multilevel WalkSAT` would just continue to project to the next level either before finding a better solution, or finding a good solution and then continuing on for minutes. Wasting minutes on careless flipping would mean a worse end result due to a given max duration. We also observed that the closer `multilevel WalkSAT` was to level 0, the more time was needed to achieve convergence for each level. Naturally, this is because the amount of clusters in the higher levels is larger than in the lower levels.

Due to the compressing of variables and clusters, the number of unsatisfied clauses minima is possibly higher for level 1 than for level 0, level 2 than for level 1, and so on. Therefore, we introduced the multilevel-multiplier as described in section 6.3. Ideally, this parameter would be given its own evaluation, but due to time constraints and the thesis scope this was not included. We chose to set multilevel-multiplier = 10, and it seemed to give good results. However, increasing or decreasing this value might give better or worse performance.

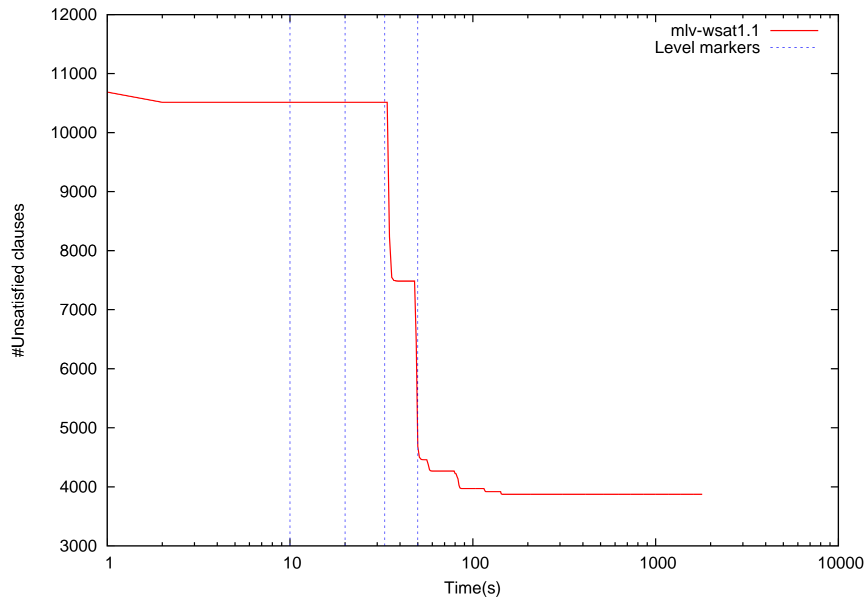

Figure 9.1: Log plot: Problem dividers_multivec1.dimacs.filtered.cnf, |variables| = 106 128, |clauses| = 397 650. Vertical axis gives the number of unsatisfied clauses, horizontal axis gives the time in seconds. The vertical bars indicate level transitions.

As an example, we will revisit a graph from section 6.4. We believe that the plateaus in figure 9.1 would be extended by increasing the multilevel-multiplier, and vice versa. However, this might also mean that `multilevel WalkSAT` could find a better solution.

Level	%PVA	#Variables/clusters
0	100	100
1	50	50
2	25	25
3	12.5	13
4	6.25	7
5	3.125	4
6	1.5625	2

Table 9.1: Multilevel consecutive reduction example. Contains levels, %PVA (percentage of possible variable assignments) and total amount of variables in a hypothetical MAXSAT problem with 100 variables at level 0. Note that we have 2 variables per cluster.

9.1.2 Number of levels

Regarding the amount of multilevel reduction of MAXSAT problem instances, we decided that 4 levels was sufficient as shown in table 9.1. As seen in the table, 4 levels is equal to 6.25% of the entire possible variable assignment of the problem. We see that at level 6 there are only two clusters which means that for `mlv-wsat1.0` and `mlv-wsat1.1` it would only be possible to change the truth value of two clusters. Of course, projecting from level 6 to level 5 will give 4 clusters, which is a bit more, but wasting time at these levels due to the low of amount of $\%PVA$ seems unnecessary. The situation is different for `mlv-wsat2.0` and `mlv-wsat2.1` due to the fact that they allow single variables to be flipped, hence breaking the cluster truth value.

9.1.3 Single versus cluster

In essence we introduced two different multilevel implementations of `WalkSAT`. The first one, `mlv-wsat1.0`, operates only on clusters whereas the second one, `mlv-wsat2.0`, operates on both variables and clusters. This means that `mlv-wsat1.0` will always have clusters that have subclusters and subvariables with the same truth value. Since `mlv-wsat2.0` might flip a single variable on a greedy move, as given in table 6.1, there might be a discrepancy between the truth values of subclusters and subvariables in a cluster. We have reason to believe that the `mlv-wsat1.0` performs better due to the multilevel-breakcount, see definition 8, compared to the regular `WalkSAT` breakcount, see definition 5. The multilevel-breakcount covers an entire cluster, but regular breakcount only covers a single variable. Further, looking at the experimental results we can confirm that `mlv-wsat1.0` outperforms `mlv-wsat2.0`. Therefore we believe that using the regular breakcount in a multilevel context is unfortunate.

9.1.4 Static versus dynamic noise

Additionally, we also experimented with dynamic noise. We noticed that almost all moves leading to a better solution happened at $noise = 0\%$, the first noise value, even though the noise would go through all noise values. Further, we observed that regular `multilevel WalkSAT` with static noise would stagnate at $noise = 0\%$ pretty fast, so we believe that going through the noise values must have changed the search space significantly. This could be verified by evaluating noise patterns.

9.2 Learning Automata WalkSAT

In section 7.4 we evaluated different number of states per action for each Learning automaton. Findings show that `la-wsat1.0` and `la-wsat1.1` gave best results on 3 and 2 states respectively. We know that Learning Automata eliminates some random moves for `la-wsat1.0`, and some random moves and greedy moves for `la-wsat1.1`. These moves are skipped because at each variable there is a Learning automaton that controls whether flipping should happen or not depending on the current state of the Learning automaton. We believe that these eliminated moves, a side-effect of learning, are part of the reason that `la-wsat1.0` and `la-wsat1.1` perform better than `WalkSAT`.

The further the distance from the action boundary, the more difficult it is to reach a state where it is possible to flip. As an example, if the number of states was 100 per action per Learning automaton, then the amount of punish needed would be at least 100 to be able to flip that variable, given that the Learning automaton was at state 100 or -100. Therefore a lower distance is preferred, which can be seen from the state evaluation in section 7.4.

In a way, Learning Automata could be seen as a bookkeeping method for classifying good and bad variable candidates for flipping. The algorithms choose how they want to use this information. We acknowledge that `la-wsat1.1` is the best performing algorithm of the two proposed algorithms in chapter 7.

9.3 Multilevel Learning Automata WalkSAT

We joined the best versions of `Learning Automata WalkSAT` and `multilevel WalkSAT`, namely `la-wsat1.1` and `mlv-wsat1.0`, together to form `mlv-la-wsat`. When these two were combined, they got better results than when separated. We believe multilevel techniques accelerates Learning Automata.

When Learning Automata are applied to the multilevel paradigm like in our research, there is a Learning automaton per cluster for level 1 to N. For level 0, there are only variables, so at level 0 the algorithm is essentially the same as `la-wsat1.1`. However, level 0 has a start solution from level 1.

It was not surprising that Learning Automata also works on clusters, due to a common truth value for all the variables. We believe that Learning Automata would not perform equally well on `mlv-wsat2.0` due to different truth values among cluster variables.

9.4 Summary

In this section we relate the results and observations to the research questions we formed in section 1.3. The questions are reproduced and answered based on findings conducted from the experiments and results.

1. **Learning Automata and multilevel techniques combined with other SAT solvers have recently been used to solve SAT. What will be the outcome when `WalkSAT` is extended with the aforementioned techniques for solving MAXSAT?**

   Our results have shown that `WalkSAT` combined with Learning Automata and multilevel techniques can be successfully applied to solve MAXSAT.

2. **Assuming we experiment with different combinations of `WalkSAT` with multilevel techniques, and `WalkSAT` with Learning Automata. What is the optimal combination in both aspects?**

   During our research we tried different combinations of `Learning Automata WalkSAT` and `multilevel WalkSAT`. Among these we found that `la-wsat1.1` and `mlv-wsat1.0` were the optimal compositions. `mlv-la-wsat` is the product of combining the best two algorithms from both `Learning Automata WalkSAT` and `multilevel WalkSAT`.

3. **If `WalkSAT` extended with Learning Automata and multilevel techniques separately yields good results, what will be the result of `WalkSAT` combined with both techniques?**

   `WalkSAT` extended with Learning Automata and multilevel techniques separately produced good results. The combination of these two techniques, resulting in `mlv-la-wsat`, gave even better results. This shows that Learning Automata can be successfully applied to the multilevel paradigm on clusters.

4. **Is there an optimal configuration of Learning Automata when coupled with `WalkSAT`?**

   Findings in experiment 7.4 confirm that the number of states play an important role of the performance of `Learning Automata WalkSAT`. Both variants preferred a rather low amount of states. `la-wsat1.0` and `la-wsat1.1` preferred 3 and 2 states per action respectively. The difference in state preference is due to a difference in algorithmic behavior.

Chapter 10

Conclusion of the Research

In this chapter we give the conclusion, and finally introduce possible future work.

10.1 Conclusion

The goal of this thesis was to extend the stochastic local search algorithm `WalkSAT` with multilevel techniques and Learning Automata to see if this could enhance the performance of `WalkSAT`. We have observed the effects of the employment of the two techniques when solving industrial MAXSAT problem instances. This has been accomplished in 4 steps:

(1) We implemented `WalkSAT`, performed noise evaluation, and found the optimal noise value when solving MAXSAT industrial problem instances.

(2) We combined `WalkSAT` with multilevel techniques and introduced four `multilevel WalkSAT` variants.

(3) We combined `WalkSAT` with Learning Automata and introduced two `Learning Automata WalkSAT` variants. We also performed Learning Automata state evaluation and found the optimal number of states per action.

(4) We combined `WalkSAT` with both multilevel techniques and Learning Automata, by using the best algorithms from (2) and (3) giving `multilevel Learning Automata WalkSAT`.

This work provides a comprehensive understanding of how multilevel techniques and Learning Automata can be applied to `WalkSAT`. Extensive experiments and benchmarking on 20 industrial MAXSAT problems from the MAXSAT Evaluation have been conducted. Results and observations have led to the following conclusions:

For `multilevel WalkSAT` the results show that the most successful algorithm was `mlv-wsat1.0`. On average it had 15% improvement over `WalkSAT` on our problem suite that consisted of 20 industrial MAXSAT problems. The results for the Learning Automata variants of `WalkSAT` show that `la-wsat1.1` was able to improve the quality of the solution compared to WalkSAT with up to 46%. For `multilevel Learning Automata WalkSAT` the results show that `mlv-la-wsat` achieved an improvement of 83% over WalkSAT.

Further, we conclude that multilevel techniques and Learning Automata combined with `WalkSAT` separately proves to be a better way of solving industrial MAXSAT problems than `WalkSAT` alone. When both these techniques are combined together with `WalkSAT` this provides even better performance as can be seen from the numbers above. Thus, we have presented a combination of the work presented in [22] and [8], namely Learning Automata for SAT and multilevel techniques for SAT, for applying Learning Automata in a multilevel paradigm for solving MAXSAT.

10.2 Further work

Further research on combining both Learning Automata and multilevel techniques is interesting. One idea is to combine `mlv-wsat2.0`, containing single and cluster flipping, with `la-wsat1.1`. Combining clusters with different truth values and Learning Automata could for instance be done by setting the Learning automaton's current action (and thus also state) to the majority truth value in the cluster. Choosing an appropriate mechanism for flipping, and how to reward each Learning automaton would then be important. In general, there are a multitude of ways to reward each Learning automaton, and this could also be researched further.

In our research we used a two-action Learning automaton. This automaton could be extended with stochastic properties. For example, the states immediately next to the action boundary, namely states -1 and 0, could be modeled such that a penalty was given a 90% success rate. This would mean that if the Learning automaton was situated at state -1, and a penalty was given, then there would be a 10% chance that it would not cross the action boundary like it normally would without the stochastic properties.

Bibliography

[1] Joao Marques-Silva. Practical applications of boolean satisfiability. In *Workshop on Discrete Event Systems (WODES'08)*. IEEE Press, May 2008.

[2] Michel Vasquez and Jin-Kao Hao. A logic-constrained knapsack formulation and a tabu algorithm for the daily photograph scheduling of an earth observation satellite. *Comput. Optim. Appl.*, 20:137–157, November 2001.

[3] Dawn M. Strickland, Earl Barnes, and Joel S. Sokol. Optimal protein structure alignment using maximum cliques. *Oper. Res.*, 53:389–402, May 2005.

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

[5] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *AAAI*, pages 440–446, 1992.

[6] Holger Hoos and Thomas Sttzle. *Stochastic Local Search: Foundations & Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[7] Ole-Christoffer Granmo and Noureddine Bouhmala. Enhancing local-search based sat solvers with learning capability. In Joaquim Filipe, Ana L. N. Fred, and Bernadette Sharp, editors, *ICAART (1)*, pages 515–521. INSTICC Press, 2010.

[8] Noureddine Bouhmala. A Multilevel Memetic Algorithm for Large SAT-Encoded Problems.

[9] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009.

[10] R.P. Grimaldi. *Discrete and combinatorial mathematics: an applied introduction*. Pearson Addison Wesley, 2004.

[11] Hui Xu, Rob A. Rutenbar, and Karem Sakallah. sub-sat: a formulation for relaxed boolean satisfiability with applications in routing. In *Proceedings of the 2002 international symposium on Physical design*, ISPD '02, pages 182–187, New York, NY, USA, 2002. ACM.

[12] Josep Argelich, Chu Min Li, Felip Manyá, and Jordi Planes. The first and second max-sat evaluations. *JSAT*, 4(2-4):251–278, 2008.

[13] Jun Gu. Efficient local search for very large-scale satisfiability problems. *SIGART Bull.*, 3:8–12, January 1992.

[14] Ian Gentles, Hans van Maaren, and Tory Walsh, editors. *Sat2000: Highlights of Satisfiability Research in the Year 2000*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 1st edition, 2000.

[15] Holger H. Hoos and Thomas Stützle. Systematic vs. local search for sat. In *Proceedings of the 23rd Annual German Conference on Artificial Intelligence: Advances in Artificial Intelligence*, KI '99, pages 289–293, London, UK, UK, 1999. Springer-Verlag.

[16] Adnan Darwiche and Knot Pipatsrisawat. Complete algorithms. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 99–130. IOS Press, 2009.

[17] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, July 1960.

[18] Joaäo P. Marques-silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999.

[19] Hantao Zhang. Sato: An efficient prepositional prover. In William McCune, editor, *Automated DeductionŮCADE-14*, volume 1249 of *Lecture Notes in Computer Science*, pages 272–275. Springer Berlin / Heidelberg, 1997. 10.1007/3-540-63104-6-28.

[20] James M. Crawford and Larry D. Auton. Experimental results on the crossover point in random 3sat. *Artificial Intelligence*, 81:31–57, 1996.

[21] Chu Min Li. Heuristics based on unit propagation for satisfiability problems. pages 366–371, 1997.

[22] Noureddine Bouhmala and Ole-Christoffer Granmo. Solving graph coloring problems using learning automata. In *Proceedings of the 8th European conference on Evolutionary computation in combinatorial optimization*, EvoCOP'08, pages 277–288, Berlin, Heidelberg, 2008. Springer-Verlag.

[23] Holger H. Hoos and Thomas Stützle. SATLIB: An Online Resource for Research on SAT, 2000.

[24] Pablo Moscato. A gentle introduction to memetic algorithms. In *Handbook of Metaheuristics*, pages 105–144. Kluwer Academic Publishers, 2003.

[25] Noureddine Bouhmala and Xing Cai. *A Multilevel Greedy Algorithm for the Satisfiability Problem*, chapter 3, pages 39–54. IN-TECH Education and Publishing, Vienna, 2008.

[26] Josep Argelich, Chu Min Li, Felip Manyá, and Jordi Planes. Analyzing the instances of the maxsat evaluation. In Karem A. Sakallah and Laurent Simon, editors, *SAT*, volume 6695 of *Lecture Notes in Computer Science*, pages 360–361. Springer, 2011.

[27] Chu M. Li, Universitè Picardie, Jules Verne, and Felip Manyà. New Inference Rules for Max-SAT. *In Journal of Artificial Intelligence Research*, 2007.

[28] Javier Larrosa, Federico Heras, and Simon de Givry. A logical approach to efficient max-sat solving. *Artif. Intell.*, 172:204–233, February 2008.

[29] Josep Argelich, Chu Min Li, Felip Manyà, and Jordi Planes. The first and second max-sat evaluations. *JSAT*, 4(2-4):251–278, 2008.

[30] Federico Heras, Javier Larrosa, and Albert Oliveras. MINIMAXSAT: An Efficient Weighted Max-SAT Solver. *Journal of Artificial Intelligence Research*, 31:1–32, 2008.

[31] Dave A. D. Tompkins and Holger H. Hoos. Scaling and probabilistic smoothing: Dynamic local search for unweighted max-sat. In Yang Xiang and Brahim Chaib-draa, editors, *Canadian Conference on AI*, volume 2671 of *Lecture Notes in Computer Science*, pages 145–159. Springer, 2003.

[32] Wayne Pullan and Holger H. Hoos. Dynamic local search for the maximum clique problem. *J. Artif. Int. Res.*, 25(1):159–185, February 2006.

[33] Satisfiability Suggested Format. Technical report, 1993.

[34] Donald J. Patterson and Henry Kautz. Auto-walksat: A self-tuning implementation of walksat. In *In Electronic Notes in Discrete Mathematics (ENDM*, page 2001, 2001.

[35] David McAllester, Bart Selman, and Henry Kautz. Evidence for invariants in local search. In *Proceedings of the fourteenth national conference on artificial intelligence and ninth conference on Innovative applications of artificial intelligence*, AAAI'97/IAAI'97, pages 321–326. AAAI Press, 1997.

[36] Holger H. Hoos. An adaptive noise mechanism for walksat. In *Eighteenth national conference on Artificial intelligence*, pages 655–660, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[37] C. Walshaw. A multilevel approach to the graph colouring problem. Technical Report 01/IM/69, School of Computing and Mathematical Science, Univeristy of Greenwich, London, UK, May 2001.

[38] Chris Walshaw. A multilevel approach to the travelling salesman problem. *Oper. Res.*, 50(5):862–877, September 2002.

[39] Noureddine Bouhmala. Combining local search with the multilevel paradigm for the traveling salesman problem. In *Hybrid Metaheuristics'04*, pages 51–58, 2004.

[40] K M. L. Tsetlin. *Automaton Theory and Modeling of Biological Systems*. Academic Press, 1973.

[41] M. A. L. Narendra, K. S. & Thathachar. *Learning Automata: An Introduction,*. Prentice Hall., 1989.

[42] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, December 2002.

[43] P. Nicopolitidis, G. I. Papadimitriou, A. S. Pomportsis, P. Sarigiannidis, and M. S. Obaidat. Adaptive wireless networks using learning automata. *IEEE Wireless Communications*, 18(2):75–81, 2011.

[44] Cem Ünsal, Pushkin Kachroo, and John S. Bay. Multiple stochastic learning automata for vehicle path control in an automated highway system. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 29(1):120–128, 1999.

[45] Javad Akbari Torkestani and Mohammad Reza Meybodi. Mobility-based multicast routing algorithm for wireless mobile ad-hoc networks: A learning automata approach. *Comput. Commun.*, 33(6):721–735, April 2010.

Appendix A

Benchmarking Results

This appendix contains results from benchmarking of all the proposed algorithms presented in chapter 6: *Combining WalkSAT with Multilevel Techniques*, chapter 7: *Combining WalkSAT with Learning Automata* and chapter 8: *Combining WalkSAT with Multilevel Techniques and Learning Automata*. The results are given in section A.1, A.2 and A.3 respectively.

A.1 WalkSAT combined with multilevel techniques

Problem instance	Solver	#Mean unsatisfied clauses	Standard deviation	Variance
rsdecoder_fsm1.d imacs.filtered.cnf	WalkSAT	271.20	18.76	351.96
	mlv-wsat1.0	185.70	19.19	368.23
	mlv-wsat1.1	405.30	83.22	6926.23
	mlv-wsat2.0	267.50	7.17	51.39
	mlv-wsat2.1	1428.20	172.45	29737.73
spi2.dimacs. filtered.cnf	WalkSAT	868.60	48.19	2321.82
	mlv-wsat1.0	599.00	32.03	1025.78
	mlv-wsat1.1	1184.00	45.99	2114.89
	mlv-wsat2.0	861.70	65.97	4351.57
	mlv-wsat2.1	1211.70	25.52	651.12
wb_4m8s3.dima cs.filtered.cnf	WalkSAT	2671.00	46.20	2134.67
	mlv-wsat1.0	2505.10	87.88	7723.66
	mlv-wsat1.1	3391.60	70.93	5030.93
	mlv-wsat2.0	2661.30	55.77	3110.68
	mlv-wsat2.1	3398.00	93.25	8695.33
rsdecoder2.dima cs.filtered.cnf	WalkSAT	671.80	14.22	202.18
	mlv-wsat1.0	442.50	41.23	1699.61
	mlv-wsat1.1	1002.50	211.72	44825.83
	mlv-wsat2.0	667.50	17.92	321.17
	mlv-wsat2.1	2904.00	83.19	6920.00
i2c_master1.dim acs.filtered.cnf	WalkSAT	306.00	14.48	209.56
	mlv-wsat1.0	214.20	26.89	723.07
	mlv-wsat1.1	557.50	114.32	13068.50
	mlv-wsat2.0	309.80	15.30	234.18
	mlv-wsat2.1	650.20	29.73	883.96
SM_RX_TOP.dim acs.filtered.cnf	WalkSAT	2914.70	149.36	22307.12
	mlv-wsat1.0	2563.20	99.33	9865.51
	mlv-wsat1.1	1620.40	48.75	2376.93
	mlv-wsat2.0	2905.70	195.21	38106.46
	mlv-wsat2.1	1647.40	57.40	3294.71
rsdecoder1_ blackbox_ CSEEblock-problem. dimacs_32.filtered.cnf	WalkSAT	3445.10	381.77	145745.21
	mlv-wsat1.0	422.40	28.85	832.49
	mlv-wsat1.1	1161.70	56.78	3224.23
	mlv-wsat2.0	3538.00	166.81	27824.22
	mlv-wsat2.1	1174.30	59.96	3595.57

Table A.1: Results from benchmarking. Multilevel WalkSAT variants and WalkSAT.

Problem instance	Solver	#Mean unsatisfied clauses	Standard deviation	Variance
divider-problem.dimacs_3.filtered.cnf	WalkSAT	6195.00	127.43	16238.44
	mlv-wsat1.0	4079.60	104.52	10924.93
	mlv-wsat1.1	5828.00	328.07	107632.89
	mlv-wsat2.0	6218.10	103.65	10743.43
	mlv-wsat2.1	6149.30	429.15	184167.79
spi-debug. dimacs.cnf	WalkSAT	1312.20	37.59	1412.84
	mlv-wsat1.0	1283.60	47.90	2294.04
	mlv-wsat1.1	2346.90	325.56	105991.66
	mlv-wsat2.0	1306.80	44.28	1960.40
	mlv-wsat2.1	2563.30	323.38	104574.90
divider-problem.dimacs_11.filtered.cnf	WalkSAT	5948.50	154.48	23864.72
	mlv-wsat1.0	4616.30	900.43	810767.79
	mlv-wsat1.1	5491.50	68.36	4672.50
	mlv-wsat2.0	6004.00	129.66	16812.00
	mlv-wsat2.1	6163.30	273.27	74679.12
dividers6_hack.dimacs.filtered.cnf	WalkSAT	426.70	19.83	393.34
	mlv-wsat1.0	412.50	22.65	513.17
	mlv-wsat1.1	991.80	27.93	779.96
	mlv-wsat2.0	432.40	13.89	192.93
	mlv-wsat2.1	979.90	43.67	1906.77
mrisc-debug .dimacs.cnf	WalkSAT	3772.60	532.69	283756.93
	mlv-wsat1.0	4905.70	887.27	787251.34
	mlv-wsat1.1	4812.20	1019.20	1038775.29
	mlv-wsat2.0	4157.00	494.79	244819.56
	mlv-wsat2.1	3845.70	457.26	209083.34
mem_ctrl-debug.dimacs.cnf	WalkSAT	152.80	9.68	93.73
	mlv-wsat1.0	143.50	9.71	94.28
	mlv-wsat1.1	195.80	21.88	478.84
	mlv-wsat2.0	153.80	10.13	102.62
	mlv-wsat2.1	216.20	26.25	689.29
dividers_multivec1.dimacs.filtered.cnf	WalkSAT	2912.20	78.78	6205.51
	mlv-wsat1.0	2464.90	56.23	3161.43
	mlv-wsat1.1	3875.10	84.06	7065.43
	mlv-wsat2.0	2943.00	62.73	3934.67
	mlv-wsat2.1	4062.70	179.80	32328.01

Table A.2: Results from benchmarking. Multilevel WalkSAT variants and WalkSAT.

Problem instance	Solver	#Mean unsatisfied clauses	Standard deviation	Variance
fpu_fsm1-problem.dimacs_15.filtered.cnf	WalkSAT	3690.70	54.52	2972.01
	mlv-wsat1.0	3284.00	715.02	511256.00
	mlv-wsat1.1	2706.60	113.24	12822.27
	mlv-wsat2.0	3686.30	67.00	4489.34
	mlv-wsat2.1	2949.80	69.61	4845.29
wb-debug. dimacs.cnf	WalkSAT	314.20	24.43	597.07
	mlv-wsat1.0	264.10	13.40	179.66
	mlv-wsat1.1	121.50	12.78	163.39
	mlv-wsat2.0	322.00	23.47	550.89
	mlv-wsat2.1	107.10	15.56	242.10
wb-problem.dimacs _ 45.filtered.cnf	WalkSAT	135.80	45.56	2075.51
	mlv-wsat1.0	159.70	28.84	831.57
	mlv-wsat1.1	199.30	74.34	5526.68
	mlv-wsat2.0	143.80	27.65	764.40
	mlv-wsat2.1	217.20	54.42	2961.07
c5_DD_s3_f1_e1_v1-bug-gate-0.dimacs. seq.filtered.cnf	WalkSAT	8.00	0.00	0.00
	mlv-wsat1.0	8.00	0.00	0.00
	mlv-wsat1.1	8.00	0.00	0.00
	mlv-wsat2.0	8.00	0.00	0.00
	mlv-wsat2.1	8.00	0.00	0.00
ac97_ctrl-debug.dimacs.cnf	WalkSAT	32.60	3.34	11.16
	mlv-wsat1.0	33.00	4.08	16.67
	mlv-wsat1.1	47.20	4.98	24.84
	mlv-wsat2.0	32.40	3.72	13.82
	mlv-wsat2.1	48.10	4.15	17.21
fpu8-problem.dimacs_24.filtered.cnf	WalkSAT	3608.10	53.66	2879.21
	mlv-wsat1.0	2822.90	629.11	395776.32
	mlv-wsat1.1	2737.50	210.93	44493.17
	mlv-wsat2.0	3560.40	70.42	4958.71
	mlv-wsat2.1	2845.20	150.06	22518.40

Table A.3: Results from benchmarking. Multilevel WalkSAT variants and WalkSAT.

A.2 WalkSAT combined with Learning Automata

Problem instance	Solver	#Mean unsatisfied clauses	Standard deviation	Variance
spi2.dimacs. filtered.cnf	WalkSAT	919.20	39.67	1573.51
	la-wsat1.0	276.80	35.10	1231.96
	la-wsat1.1	191.70	22.69	514.68
wb_4m8s3.dima cs.filtered.cnf	WalkSAT	2675.70	71.38	5094.68
	la-wsat1.0	784.20	45.61	2080.40
	la-wsat1.1	285.20	25.18	634.18
rsdecoder2.dima cs.filtered.cnf	WalkSAT	669.80	16.63	276.62
	la-wsat1.0	1746.80	235.36	55396.62
	la-wsat1.1	1381.10	160.94	25900.10
i2c_master1.dim acs.filtered.cnf	WalkSAT	317.20	11.69	136.62
	la-wsat1.0	102.70	23.30	543.12
	la-wsat1.1	95.40	9.55	91.16
SM_RX_TOP.dim acs.filtered.cnf	WalkSAT	2900.60	113.65	12917.38
	la-wsat1.0	556.70	47.12	2220.23
	la-wsat1.1	550.50	35.32	1247.17
rsdecoder1_ blackbox_ CSEEblock-problem. dimacs_32.filtered.cnf	WalkSAT	3479.10	275.96	76156.32
	la-wsat1.0	2270.70	72.20	5212.90
	la-wsat1.1	1868.60	90.78	8240.27
divider-problem.dim acs_3.filtered.cnf	WalkSAT	6204.80	81.33	6613.96
	la-wsat1.0	2832.50	114.72	13161.39
	la-wsat1.1	2117.00	71.94	5175.33
spi-debug. dimacs.cnf	WalkSAT	1286.10	33.87	1147.43
	la-wsat1.0	287.40	35.78	1280.04
	la-wsat1.1	222.20	39.67	1573.51
divider-problem.dim acs_11.filtered.cnf	WalkSAT	6030.20	124.94	15611.07
	la-wsat1.0	2583.80	125.90	15851.07
	la-wsat1.1	1918.20	63.80	4071.07
dividers6_hack.d imacs.filtered.cnf	WalkSAT	423.20	25.42	645.96
	la-wsat1.0	87.10	8.17	66.77
	la-wsat1.1	73.60	6.79	46.04
mrisc-debug .dimacs.cnf	WalkSAT	4064.00	542.67	294490.00
	la-wsat1.0	1636.00	515.76	266010.22
	la-wsat1.1	997.70	111.18	12361.34
mem_ctrl-deb ug.dimacs.cnf	WalkSAT	150.40	11.33	128.27
	la-wsat1.0	28.80	5.18	26.84
	la-wsat1.1	45.50	3.95	15.61

Table A.4: Results from benchmarking. Learning Automata WalkSAT variants and WalkSAT.

Problem instance	**Solver**	**#Mean unsatisfied clauses**	**Standard deviation**	**Variance**
dividers__multivec1. dimacs.filtered.cnf	WalkSAT	2953.90	30.67	940.54
	la-wsat1.0	340.60	26.33	693.16
	la-wsat1.1	250.50	24.08	580.06
ac97__ctrl-de bug.dimacs.cnf	WalkSAT	33.50	3.27	10.72
	la-wsat1.0	8.20	2.30	5.29
	la-wsat1.1	7.50	0.85	0.72
fpu__fsm1-problem.di macs__15.filtered.cnf	WalkSAT	3716.40	49.54	2454.49
	la-wsat1.0	499.60	26.26	689.60
	la-wsat1.1	422.70	44.16	1950.01
wb-debug. dimacs.cnf	WalkSAT	307.80	23.18	537.51
	la-wsat1.0	237.40	14.08	198.27
	la-wsat1.1	220.40	15.42	237.82
wb-problem.dimacs __ 45.filtered.cnf	WalkSAT	143.30	24.71	610.68
	la-wsat1.0	871.00	85.13	7247.33
	la-wsat1.1	276.80	40.14	1610.84
rsdecoder__fsm1.d imacs.filtered.cnf	WalkSAT	270.30	9.18	84.23
	la-wsat1.0	568.00	139.26	19392.00
	la-wsat1.1	438.30	96.76	9363.12
c5__DD__s3__f1__e1__v1-bug-gate-0. dimacs.seq.filtered.cnf	WalkSAT	8.00	0.00	0.00
	la-wsat1.0	18.60	31.14	969.60
	la-wsat1.1	8.00	0.00	0.00
fpu8-problem.dimac s__24.filtered.cnf	WalkSAT	3585.90	61.94	3836.99
	la-wsat1.0	535.80	34.34	1179.29
	la-wsat1.1	412.50	35.87	1286.72

Table A.5: Results from benchmarking. Learning Automata WalkSAT variants and WalkSAT.

A.3 WalkSAT combined with multilevel techniques and Learning Automata

Problem instance	Solver	#Mean unsatisfied clauses	Standard deviation	Variance
spi2.dimacs. filtered.cnf	WalkSAT	896.10	33.65	1132.10
	mlv-wsat1.0	589.10	39.74	1578.99
	mlv-la-wsat	147.40	7.24	52.49
	la-wsat1.1	184.30	13.71	188.01
wb_4m8s3.dimacs.filtered.cnf	WalkSAT	2665.90	77.37	5986.54
	mlv-wsat1.0	2514.00	68.41	4680.22
	mlv-la-wsat	178.20	17.87	319.51
	la-wsat1.1	282.50	28.27	799.17
rsdecoder2.dimacs.filtered.cnf	WalkSAT	666.20	13.85	191.96
	mlv-wsat1.0	444.70	26.49	701.57
	mlv-la-wsat	81.60	1.71	2.93
	la-wsat1.1	1323.50	380.11	144484.72
i2c_master1.dimacs.filtered.cnf	WalkSAT	313.60	15.51	240.71
	mlv-wsat1.0	220.70	29.20	852.90
	mlv-la-wsat	43.30	34.28	1174.90
	la-wsat1.1	99.80	8.73	76.18
SM_RX_TOP.dimacs.filtered.cnf	WalkSAT	2964.70	121.46	14752.01
	mlv-wsat1.0	2589.40	95.81	9180.04
	mlv-la-wsat	312.10	64.73	4189.88
	la-wsat1.1	531.10	31.56	996.10
rsdecoder1_blackbox_CSEEblock-problem.dimacs_32.filtered.cnf	WalkSAT	3561.60	235.88	55637.16
	mlv-wsat1.0	405.00	41.73	1741.78
	mlv-la-wsat	39.70	7.50	56.23
	la-wsat1.1	1832.00	66.54	4427.11
divider-problem.dimacs_3.filtered.cnf	WalkSAT	6143.70	128.13	16417.34
	mlv-wsat1.0	3999.60	98.01	9606.71
	mlv-la-wsat	54.30	28.53	814.23
	la-wsat1.1	2155.40	98.21	9644.71
spi-debug. dimacs.cnf	WalkSAT	1313.10	56.64	3208.54
	mlv-wsat1.0	1266.90	38.68	1496.10
	mlv-la-wsat	98.70	17.11	292.68
	la-wsat1.1	222.80	35.70	1274.40
divider-problem.dimacs_11.filtered.cnf	WalkSAT	6011.50	181.47	32932.72
	mlv-wsat1.0	4758.90	1070.41	1145773.43
	mlv-la-wsat	42.10	20.29	411.88
	la-wsat1.1	1861.10	87.65	7682.54

Table A.6: Results from benchmarking. Multilevel WalkSAT, Learning Automata WalkSAT, multilevel Learning Automata WalkSAT and WalkSAT.

Problem instance	**Solver**	**#Mean unsatisfied clauses**	**Standard deviation**	**Variance**
dividers6_hack.d imacs.filtered.cnf	WalkSAT	431.60	15.79	249.38
	mlv-wsat1.0	407.30	33.52	1123.34
	mlv-la-wsat	53.10	5.65	31.88
	la-wsat1.1	73.40	5.74	32.93
mrisc-debug .dimacs.cnf	WalkSAT	3942.20	524.07	274653.07
	mlv-wsat1.0	5159.80	912.30	832291.73
	mlv-la-wsat	1248.60	341.20	116419.16
	la-wsat1.1	979.10	115.18	13267.21
mem_ctrl-deb ug.dimacs.cnf	WalkSAT	147.80	11.84	140.18
	mlv-wsat1.0	146.00	8.06	64.89
	mlv-la-wsat	6.70	2.54	6.46
	la-wsat1.1	45.50	4.01	16.06
dividers_multivec1. dimacs.filtered.cnf	WalkSAT	2949.90	33.22	1103.88
	mlv-wsat1.0	2462.80	86.15	7421.73
	mlv-la-wsat	225.80	22.89	523.73
	la-wsat1.1	256.50	11.60	134.50
ac97_ctrl-de bug.dimacs.cnf	WalkSAT	33.50	2.88	8.28
	mlv-wsat1.0	32.50	3.89	15.17
	mlv-la-wsat	7.40	0.97	0.93
	la-wsat1.1	7.30	0.82	0.68
fpu_fsm1-problem.di macs_15.filtered.cnf	WalkSAT	3684.40	72.25	5219.82
	mlv-wsat1.0	2865.30	716.68	513633.12
	mlv-la-wsat	323.70	106.23	11284.46
	la-wsat1.1	376.40	38.48	1480.49
wb-debug. dimacs.cnf	WalkSAT	319.80	22.31	497.96
	mlv-wsat1.0	250.20	14.45	208.84
	mlv-la-wsat	116.70	11.18	124.90
	la-wsat1.1	218.40	15.20	230.93
wb-problem.dimacs _45.filtered.cnf	WalkSAT	147.00	29.27	856.89
	mlv-wsat1.0	134.20	32.77	1073.96
	mlv-la-wsat	52.50	9.85	96.94
	la-wsat1.1	270.30	48.72	2374.01
c5_DD_s3_f1_e1_v1- bug-gate-0. dimacs.seq.filtered.cnf	WalkSAT	8.00	0.00	0.00
	mlv-wsat1.0	8.00	0.00	0.00
	mlv-la-wsat	8.00	0.00	0.00
	la-wsat1.1	34.00	44.28	1960.89

Table A.7: Results from benchmarking. Multilevel WalkSAT, Learning Automata WalkSAT, multilevel Learning Automata WalkSAT and WalkSAT.

Problem instance	**Solver**	**#Mean unsatisfied clauses**	**Standard deviation**	**Variance**
fpu8-problem.dimac s_24.filtered.cnf	WalkSAT	3613.80	44.66	1994.18
	mlv-wsat1.0	3467.60	451.46	203814.93
	mlv-la-wsat	279.80	100.34	10068.18
	la-wsat1.1	424.30	32.27	1041.57
rsdecoder__fsm1.d imacs.filtered.cnf	WalkSAT	268.70	11.59	134.23
	mlv-wsat1.0	176.60	17.39	302.49
	mlv-la-wsat	10.90	1.37	1.88
	la-wsat1.1	443.50	192.86	37196.50

Table A.8: Results from benchmarking. Multilevel WalkSAT, Learning Automata WalkSAT, multilevel Learning Automata WalkSAT and WalkSAT.