International Journal of Computer Science & Applications
Vol. 4 Issue 3, pp 15-29
© 2007 Technomathematics Research Foundation

Solving the Satisfiability Problem Using Finite Learning Automata

Ole-Christoffer Granmo[1] ole.granmo@hia.no
Noureddine Bouhmala[2] noureddine.bouhmala@hive.no

[1]Agder University College, Grooseveien 36, N-4876 Grimstad, Norway
[2]Vestfold University College, Box 2243, N-3103 Tønsberg, Norway

Abstract

A large number of problems that occur in knowledge-representation, learning, VLSI-design, and other areas of artificial intelligence, are essentially satisfiability problems. The satisfiability problem refers to the task of finding a truth assignment that makes a Boolean expression true. The growing need for more efficient and scalable algorithms has led to the development of several SAT solvers. This paper reports the first approach based on combining finite learning automata with metaheuristics. In brief, we introduce a new algorithm that combines finite learning automata with traditional random walk. Furthermore, we present a detailed comparative analysis of the new algorithm's performance, using a benchmark set containing instances from randomized distributions, as well as SAT-encoded problems from various domains.

Keywords: SAT, learning automata, combinatorial optimization

1 Introduction

The Satisfiability problem (SAT), which is known to be NP-complete [1], plays a central role in many applications in the fields of VLSI Computer-Aided design, Computing Theory, and Artificial Intelligence. Generally, a SAT problem consists of a propositional formula:

$$\Phi = \bigwedge_{j=1}^{m} C_j ,$$

with m clauses and n Boolean variables. A Boolean variable $x_i, i \in \{1,\ldots,n\}$, is a variable that can take one of the two values: *True* or *False*. Each clause C_j is a disjunction of Boolean variables and has the form:

$$C_j = \left(\bigvee_{k \in I_j} x_k \right) \vee \left(\bigvee_{l \in \overline{I}_j} \overline{x}_l \right),$$

where $I_j, \overline{I}_j \subseteq \{1,\ldots n\}, I_j \cap \overline{I}_j = \emptyset$, and $\overline{x}_i$ denotes the negation of x_i. The task is to determine whether the propositional formula Φ evaluates to *True*. Such an assignment, if it exists, is called a satisfying assignment for Φ, and Φ is called satisfiable. Otherwise, Φ is said to be unsatisfiable. Note that since we have two choices for each of the n Boolean variables, the size of the search space S of truth assignments is $|S| = 2^n$. That is, the size of the search space grows exponentially with the number of variables.

Most SAT solvers use a Conjunctive Normal Form (CNF) representation of the propositional formula. In CNF,

the formula is represented as a conjunction of clauses, where each clause is a disjunction of literals, and a literal is a Boolean variable or its negation. For example, $P \vee Q$ is a clause containing the two literals P and Q. This clause is satisfied if either P is *True* or Q is *True*. When each clause in Φ contains exactly k literals, the resulting SAT problem is called *k*-SAT. In this paper, we focus on the 3-SAT problem, where each clause contains exactly 3 literals.

The paper is organized as follows. In Section 2, we review various algorithms for SAT problems. Section 3 explains the basic concepts of learning automata and introduces our new approach – the Learning Automata Random Walk (LARW). In Section 4, we present the results from testing LARW on a suit of problem instances. Finally, in Section 5 we present a summary of our work and provide pointers to further work.

2 Methods for SAT

The SAT has been extensively studied due to its simplicity and applicability. The simplicity of the problem coupled with its intractability, makes it an ideal platform for exploring new algorithmic techniques. This has led to the development of several algorithms for solving SAT problems, which usually fall into two main categories: *systematic search algorithms* and *local search algorithms*. Systematic search algorithms are guaranteed to return a solution to a problem if one exists, and prove it unsolvable otherwise. The most popular and efficient systematic search algorithms for SAT are based on the Davis-Putnam (DP) [2] procedure, which essentially provides a recursive, depth first enumeration of all possible variable assignments. However, due to the NP-hardness of SAT, large and complex SAT problems are hard to solve using systematic algorithms. One way to overcome the combinatorial explosion of SAT is to give up completeness. Local search algorithms are techniques which use this strategy.

Local search algorithms are based on what is perhaps the oldest optimization method – *trial and error*. Typically, they start with an initial assignment of truth values to variables, randomly or heuristically generated. Satisfiability can then be formulated as an *iterative* optimization problem in which the goal is to minimize the number of unsatisfied clauses. Thus, the optimum is obtained when the value of the objective function equals zero, which means that all clauses are satisfied. During each iteration, a new value assignment is selected from the "neighborhood" of the present one, by performing a "move". Most local search algorithms use a 1-flip neighborhood relation, which means that two truth value assignments are considered to be neighbors if they differ in the truth value of *only one* variable. Performing a move, then, consists of switching the present value assignment with one of the neighboring value assignments, e.g., if the neighboring one is better (as measured by the objective function). The search terminates if no better neighboring assignment can be found. Note that choosing a fruitful neighborhood, and a method for searching it, is usually guided by intuition – theoretical results that can be used as guidance are sparse.

One of the most popular local search algorithms for solving SAT is GSAT [3]. Basically, GSAT begins with a random generated assignment of truth values to variables, and then uses *the steepest descent heuristic* to guide the switching between neighboring truth value assignments. Essentially, moves that maximize decrease in the numbers of unsatisfied clauses are sought. After a fixed number of moves, the search is restarted from a new random assignment. The search continues until a solution is found or a fixed number of restarts have been performed. An extension of GSAT, referred as GSAT with random-walk [4], has been realized with the purpose of escaping from local optima. In a random walk step, a randomly unsatisfied clause is selected. Then, one of the variables appearing in that clause is flipped, thus effectively forcing the selected clause to become satisfied. The main idea is to decide at each search step whether to perform a standard GSAT move or a random-walk move. The probability of performing a random-walk move is called the walk probability.

Another widely used variant of GSAT is the Walk-SAT algorithm, originally introduced in [5]. The Walk-SAT algorithm first randomly picks an unsatisfied clause. One of the variables in the clause - the one with the lowest *break count* - is then randomly selected. The break count of a variable is defined as the number of clauses that would be unsatisfied by flipping that variable. In order to diversify the search, one of the other variables in the clause may also be selected with a certain probability. The random choice of unsatisfied clauses, combined with the randomness in the selection of variables, enable Walk SAT to avoid local minima

and to better explore the search space. Note that many versions of the algorithm exist.

Recently, several new algorithms [5, 6, 7, 8, 9] have emerged using history-based variable selection strategies in order to avoid repetitive looping among a limited number of truth value assignments. Apart from GSAT and its variants, several clause weighting based local search algorithms [10, 11] have been proposed to solve SAT problems. The key idea is to associate the clauses of the given CNF formula with weights. Although these clause-weighting local search algorithms differ in the manner clause weights are updated (probabilistic or deterministic), they all choose to increase the weights of all the unsatisfied clauses as soon as a local minimum is encountered. Clause weighting acts as a diversification mechanism rather than a way of escaping local minima. Finally, many other local search algorithms have been applied to SAT. These include techniques such as Simulated Annealing [12], Evolutionary Algorithms [13], and Greedy Randomized Adaptive Search Procedures [14].

3 Solving SAT Using Finite Learning Automata

We base our work on the principles of Learning Automata [15, 16]. Learning Automata have been used to model biological systems [17], and have attracted considerable interest in the last decade because they can learn the optimal actions when operating in (or interacting with) unknown stochastic environments. Furthermore, they combine rapid and accurate convergence with low computational complexity. Although the SAT problem has not been addressed from a Learning Automata point of view before, Learning Automata solutions have recently been proposed for several other combinatorial optimization problems. In [18, 19] a so-called Object Migration Automaton is used for solving the classical equipartitioning problem. An order of magnitude faster convergence is reported compared to the best known algorithms at that time. A similar approach has also been discovered for the Graph Partitioning Problem [20]. Finally, the list organization problem has successfully been addressed by Learning Automata schemes, supporting converge to the optimal arrangement with probability arbitrary close to unity [21].

Inspired by the success of the above schemes, we will in the following propose a Learning Automata based scheme for solving SAT problems.

3.1 A Learning SAT Automaton

Generally stated, a finite learning automaton performs a sequence of actions on an environment. The environment can be seen as a generic unknown medium that responds to each action with some sort of reward or penalty, perhaps stochastically. Based on the responses from the environment, the aim of the finite learning automaton is to find the action that minimizes the expected number of penalties received. Figure 1 illustrates the interaction between the finite learning automaton and the environment. Because we treat the environment as unknown, we will here only consider the definition of the finite learning automaton.

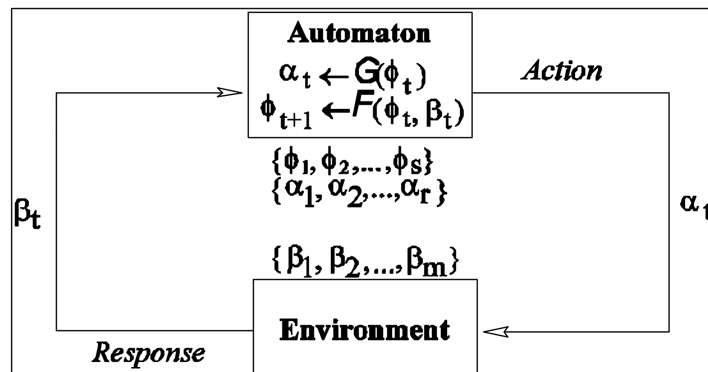

Figure 1: A learning automaton interacting with an environment.

The finite learning automaton can be defined in terms of a quintuple [15]:

$$\{\underline{\Phi}, \underline{\alpha}, \underline{\beta}, F(\cdot,\cdot), G(\cdot,\cdot)\}.$$

$\underline{\Phi} = \{\phi_1, \phi_2, \ldots, \phi_s\}$ is the set of internal automaton states. $\underline{\alpha} = \{\alpha_1, \alpha_2, \ldots, \alpha_r\}$ is the set of automaton actions. And, $\underline{\beta} = \{\beta_1, \beta_2, \ldots, \beta_m\}$ is the set of inputs that can be given to the automaton. An output function $\alpha_t = G[\phi_t]$ determines the next action performed by the automaton given the current automaton state. Finally, a transition function $\phi_{t+1} = F[\phi_t, \beta_t]$ determines the new automaton state from: (1) the current automaton state and (2) the response of the environment to the action performed by the automaton.

Based on the above generic framework, the crucial issue is to design automata that can learn the optimal action when interacting with the environment.

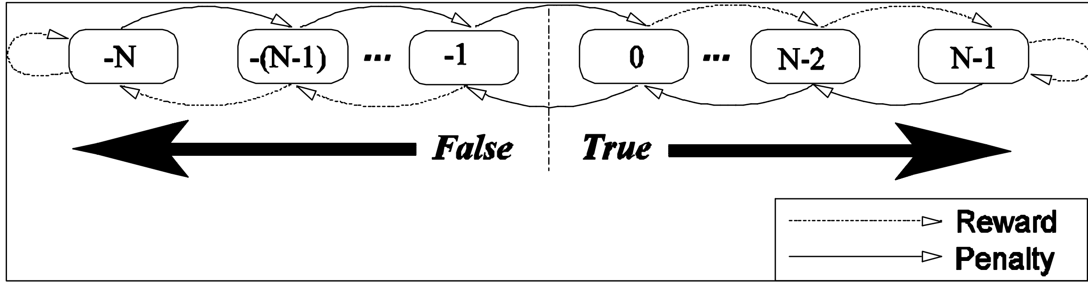

Figure 2: The state transitions and actions of the Learning SAT Automaton.

Several designs have been proposed in the literature, and the reader is referred to [15, 16] for an extensive treatment. In this paper we target the SAT problem, and our goal is to design a team of Learning Automata that seeks the solution of SAT problem instances. We build upon the work of Tsetlin and the linear two-action automaton [15, 17]. Briefly stated, for each variable in the SAT problem instance that is to be solved, we construct an automaton with

- States: $\underline{\Phi} = \{-N, -(N-1), \ldots, -1, 0, \ldots, N-2, N-1\}$.
- Actions: $\underline{\alpha} = \{\text{True}, \text{False}\}$.
- Inputs: $\underline{\beta} = \{\text{reward}, \text{penalty}\}$.

Figure 2 specifies the F and G matrices. The G matrix can be summarized as follows. If the automaton state is positive, then action True will be chosen by the automaton. If the state is negative, on the other hand, action False will be chosen. Note that since we initially do not know which action is optimal, we set the initial state of each Learning SAT Automaton randomly to either '-1' or '0'.

The state transition matrix F determines how learning proceeds. As seen from the finite automaton in the figure, providing a reward input to the automaton strengthens the currently chosen action, essentially by making it less likely that the other action will be chosen in the future. Correspondingly, a penalty input weakens the currently selected action by making it more likely that the other action will be chosen later on. In other words, the automaton attempts to incorporate past responses when deciding on a sequence of actions.

3.2 Learning Automata Random Walk (LARW)

Overview: In addition to the definition of the Learning SAT Automaton, we must define the environment that the Learning SAT Automata interact with. Simply put, the environment is a SAT problem instance as defined in Section 1. Each variable of the SAT problem instance is assigned a dedicated Learning SAT Automaton, resulting in a team of n Learning SAT Automata. The task of each Learning SAT Automaton is to determine

the truth value of its corresponding variable, with the aim of satisfying all of the clauses where that variable appears. In other words, if each automaton reaches its own goal, then the overall SAT problem at hand has also been solved.

Pseudo-code: With the above perspective in mind, we will now present the details of the LARW. Figure 3 contains the complete pseudo-code for solving SAT problems using a team of n Learning SAT Automata. As seen from the figure, the LARW corresponds to an ordinary Random Walk (RW), however, both satisfied and unsatisfied clauses are used in the search. Furthermore, the assignment of truth values to variables is indirect, governed by the states of the automata.

```
Procedure learning_automata_random_walk()
Begin
  /* Initialization */
  For i = 1 To n Do
    /* The initial state of each automaton is set to either '-1' or '1' */
    state[i] = random_element({-1,0});
    /* And the respective variables are assigned corresponding truth values */
    If state[i] == -1 Then x_i = False Else x_i = True;
  /* Main loop */
  While Not stop() Do
    /* Draw unsatisfied clause randomly */
    C_j = random_unsatisfied_clause();
    /* Draw variable index randomly from clause */
    i = random_variable_index(I_j ∪ Ī_j);
    /* The corresponding automaton is penalized for choosing the ``wrong" action */
    If i ∈ I_j Then
      state[i]++;
      /* Flip variable when automaton changes its action */
      If state[i] == 0 Then
        flip(x_i);
    Else If i ∈ Ī_j Then
      state[i]--;
      /* Flip variable when automaton changes its action */
      If state[i] == -1 Then
        flip(x_i);
    /* Draw satisfied clause randomly */
    C_j = random_satisfied_clause();
    /* Draw variable index randomly from clause */
    i = random_variable_index(I_j ∪ Ī_j);
    /* Reward corresponding automaton if it contributes to the satisfaction of the clause */
    If i ∈ I_j And state[i] >= 0 And state[i] < N-1 Then
      state[i]++;
    Else If i ∈ Ī_j And state[i] < 0 And state[i] > -N Then
      state[i]--;
  Ewhile
Emethod
```

Figure 3: Learning Automata Random Walk Algorithm.

At the core of the LARW is a punishment/rewarding scheme that guides the team of automata towards the optimal assignment. In the spirit of automata based learning, this scheme is incremental, and learning is performed gradually, in small steps. To elaborate, in each iteration of the algorithm, we randomly select a single clause. A variable is randomly selected from *that* clause, and the corresponding automaton is identified. If the clause is unsatisfied, the automaton is punished. Correspondingly, if the clause is satisfied, the automaton is rewarded, however, only if the automaton makes the clause satisfied.

Remark 1: Like a two-action Tsetlin Automaton, our Learning SAT Automaton seeks to minimize the expected number of penalties it receives. In other words, it seeks finding the truth assignment that minimizes the number of unsatisfied clauses among the clauses where its variable appears.

Remark 2: Note that because multiple variables, and thereby automata, may be involved in each clause, we are dealing with a *game* of learning automata [15]. That is, multiple learning automata interact with the same environment, and the response of the environment depends on the actions of several automata. In fact, because there may be conflicting goals among the automata involved in the LARW, the resulting game is competitive. The convergence properties of general competitive games of learning automata have not yet been successfully analyzed, however, results exists for certain classes of games, such as the Prisoner's Dilemma game [15]. In our case, the Learning SAT Automata involved the LARW are non-absorbing, i.e., every state can be reached from every other state with positive probability. This means that the probability of reaching the solution of the SAT problem instance at hand is equal to 1 when running the game infinitively. Also note that the solution of the SAT problem corresponds to a Nash equilibrium of the game.

Remark 3: In order to maximize speed of learning, we initialize each Learning SAT Automaton randomly to either the state '-1' or '0'. In this initial configuration, the variables will be flipped relatively quickly because only a single state transition is necessary for a flip. Accordingly, the joint state space of the automata is quickly explored in this configuration. However, as learning proceeds and the automata move towards their boundary states, i.e., states '-N-1' and 'N', the flipping of variables calms down. Accordingly, the search for a solution to the SAT problem instance at hand becomes increasingly focused.

4 Experimental Results

4.1 Benchmark Instances

As a basis for the empirical evaluation of LA, we selected a benchmark suite from three different domains: uniform random 3-SAT, SAT-encoded graph coloring, and finally SAT-encoded instances from AI planning domains, in particular, from the blocks World domain. All these benchmark instances are known to be hard and difficult to solve. They are available from the SATLIB website (*http://www.informatik.tu-darmstadt.de/AI/SATLIB*). Note that the benchmark instances used in this experiment are satisfiable instances that have been used widely in the literature in order to give an overall picture of the performance of different algorithms. The benchmark library used in this work contains 605 problem instances. Due to the randomization of the algorithm, the number of flips required for solving a problem instance varies widely between different runs. Therefore, for each problem instance, we run LARW and RW 100 times, with a cutoff parameter (maxflips) setting which is high enough (10^7) to guarantee a success rate close to 100%.

4.2 Search Space

The manner in which the Learning SAT Automata converge to a stable assignment is crucial for a better understanding of LARW's behavior. In Figure 4, we show how the best-found and current assignment evolve during the search on a random 3-SAT problem with 150 variables and 645 clauses, taken from the SAT benchmark library. Figure 5 shows the corresponding results for a random 3-SAT problem with 600 variables and 2550 clauses.

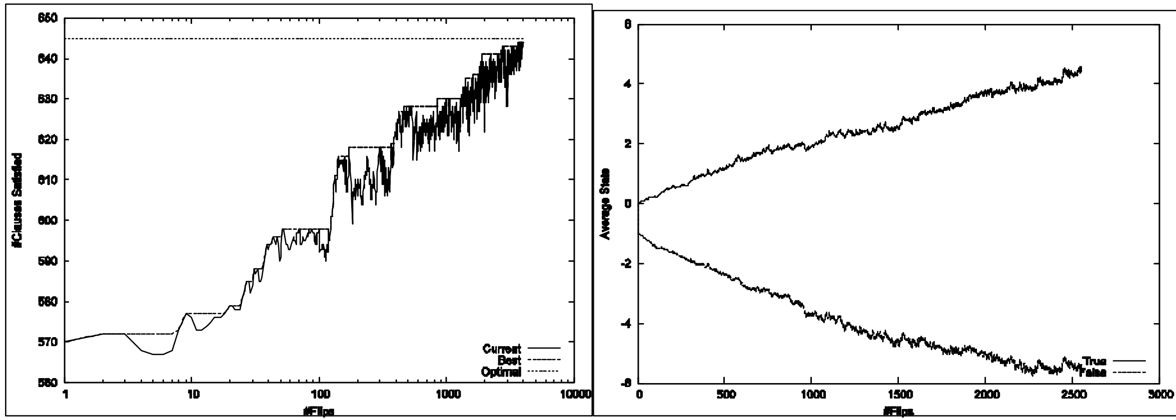

Figure 4: (Left) LARW's search space on a 150 variable problem with 645 clauses (uf150-645). Along the horizontal axis we give the number of flips, and along the vertical axis the number of satisfied clauses. (Right) Average state of automaton. Horizontal axis gives the number of flips, and the vertical axis shows the average state of automaton.

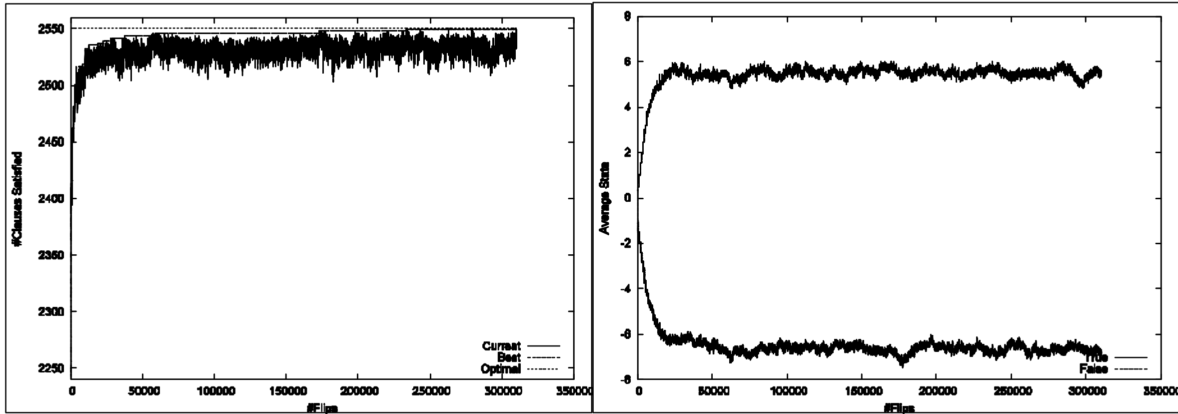

Figure 5: (Left) LARW's search space on a 600 variable problem with 2550 clauses (f600.cnf). Along the horizontal axis we give the number of flips, and along the vertical axis the number of satisfied clauses. (Right) Average state of automaton. Horizontal axis gives the number of flips, and the vertical axis shows the average state of automaton.

The plots located on the left in both figures suggest that problem solving with LARW happens in two phases. In the first phase, the LARW behaves as a hill-climbing method. In this phase, which can be described as a relatively short one, up to 95% of the clauses are satisfied. As seen, the currently best found assignment improves rapidly at first, and then flattens off as we approach a plateau, which introduces the second phase. The plateau spans a region in the search space where flips typically leave the best assignment unchanged, however, the search continues with the purpose of escaping local optima.

To further investigate the behavior of LARW once on the plateau, we looked at the corresponding average state of the automata as the search progresses. The plots located on the right in Figure 4 and Figure 5 report the resulting observations. At the start of the plateau, search coincides in general with an increase in the average state. The longer the plateau is, the higher the average state becomes. An automaton with high average state needs to perform a series of actions before its current state changes to either '-1' or '0', thereby making the flipping of the corresponding variable possible. The transition between each plateau corresponds to a change to the region where a small number of flips gradually improves the score of the current solution ending with an improvement of the best assignment. The search pattern brings out an interesting difference between LARW and the standard use of local search methods. In the latter, one generally stops the search as soon as no more improvements is found. This can be appropriate when looking for a near-solution. On the

other hand, when searching for a global maximum (i.e., a satisfying assignment) stopping when none of flips offer an immediate improvement, is a poor strategy.

4.3 Run-Length-Distributions (RLDs)

As an indicator of the behavior of the algorithm on a single instance, we choose the median cost when trying to solve a given instance in 100 trials, and using an extremely high cutoff parameter setting of $Maxsteps = 10^7$ in order to obtain a maximal number of successful tries. The reason behind choosing the median cost rather than the mean cost is due to the large variation in the number of flips required to find a solution.

To get an idea of the variability of the search cost, we analyzed the cumulative distribution of the number of search flips needed by both LARW and RW for solving single instances. Due to non-deterministic decisions involved in the algorithm (i.e., initial assignment and random moves), the number of flips needed by both algorithms to find a solution is a random variable that varies from run to run. More formally, let k denote the total number of runs, and let $f'(j)$ denote the number of flips for the j-th successful run (i.e., run during which a solution is found) in a list of all successful runs, sorted according to increasing number of flips. Then the cumulative empirical RLD is defined by:

$$P[f'(j) \le f] = \frac{|\{j \mid f'(j) \le f\}|}{k}.$$

For practical reasons we restrict our presentation here to the instances corresponding to small, medium, and large size from the underlying test-set.

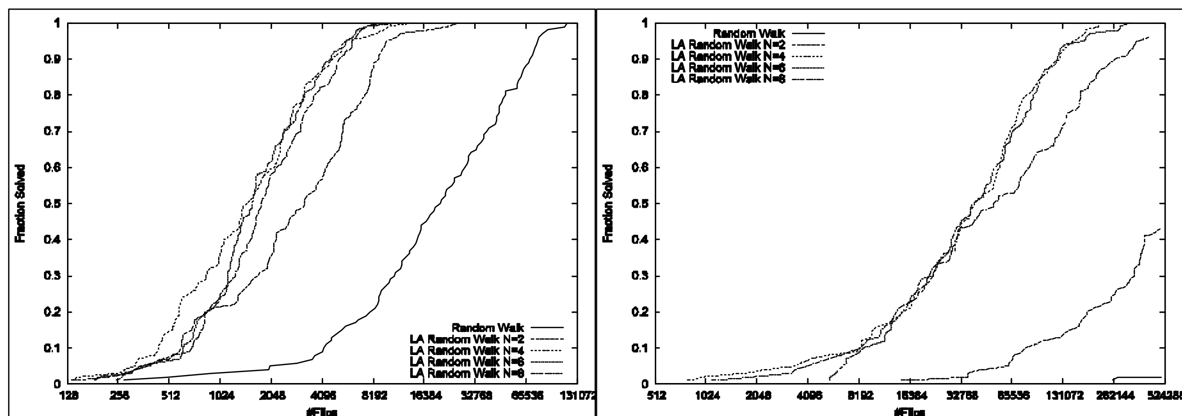

Figure 6: (Right) LARW Vs RW: Cumulative distributions for a 50-variable *random* problem with 218 clauses (uf50-218). (Left) Cumulative distribution for a 125-variable random problem with 538 clauses (uf125-538). Along the horizontal axis we give the number of flips, and along the vertical axis the fraction of problems solved for different values of N.

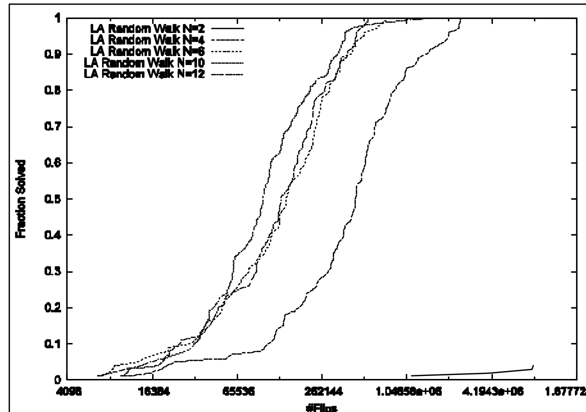

Figure 7: (Right) LARW Vs RW: cumulative distributions for a 255-variable random problem with 960 clauses (uf225-960). Along the horizontal axis we give the number of flips, and along the vertical axis the fraction of problems solved for different values of N.

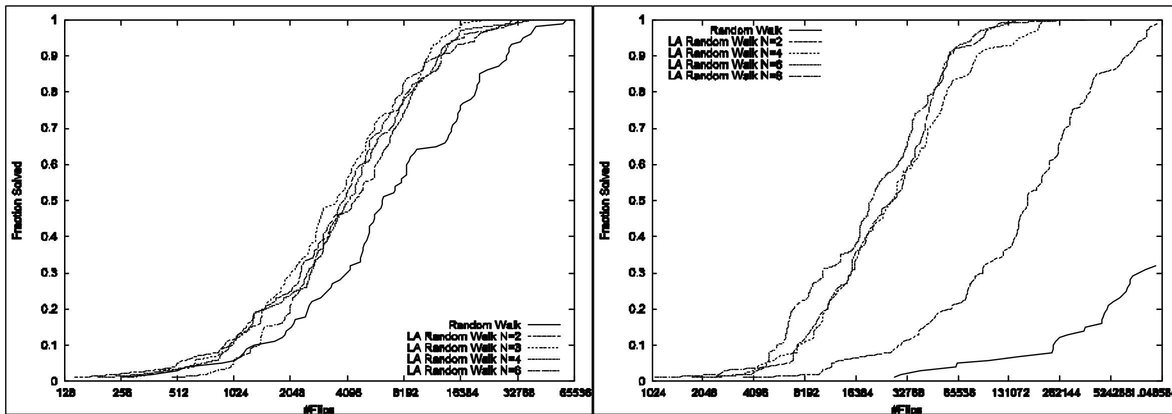

Figure 8: (Right) LARW Vs RW: Cumulative distributions for a 90-variable graph coloring problems with 300 clauses(flat90-300). (Left) Cumulative distribution for a 150-variable graph coloring problem with 545 clauses (flat375-1403). Along the horizontal axis we give the number of flips, and along the vertical axis the fraction of problems solved for different values of N.

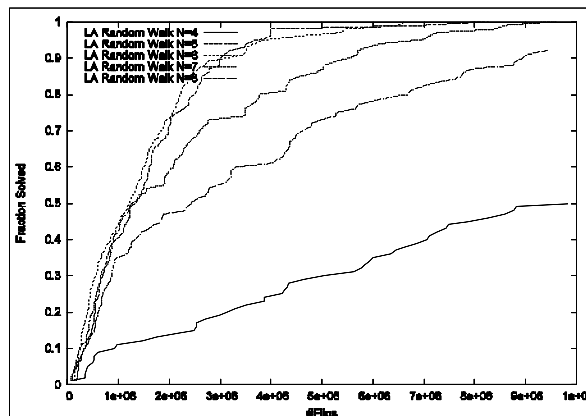

Figure 9: LARW Vs RW: cumulative distributions for a 375-variable graph coloring problems with 1403 clauses (flat375-1403). Along the horizontal axis we give the number of flips, and along the vertical axis the fraction of problems solved for different values of N.

Figures 6 and 7 show RLDs obtained by applying RW and LARW to individual Random-3-SAT problem instances. As can be seen from Figure 6, we observe that on the small size instance, the two algorithms show no CMOS-over in their corresponding RLDs. This provides evidence for the superiority of LARW compared to RW (i.e., N = 1) as it gives consistently higher success probabilities, regardless of the number of search steps. On the medium size instance, we observe a stagnation behavior with extremely low asymptotic solution probability corresponding to 0.04. As can be easily seen, both methods show the existence of an initial phase below which the probability for finding a solution is 0. Both methods start the search from a randomly chosen assignment which typically violates many clauses. Consequently, both methods need some time to reach the first local optimum which possibly could be a feasible solution. The plot in Figure 7 shows that the performance of RW for the large instance uf225-960 is dramatically poor – the probability of finding a feasible solution within the required number of steps is 0. The value of the distance between the minimum and the maximum number of search steps needed for finding a feasible solution using RW is higher compared to that of LARW and increases with the hardness of the instance. Note also, that there is a huge variability in the length of different runs of RW compared to LARW in all instances.

Looking at Figures 8 and 9 and applying the RLD analysis to SAT-encoded graph coloring problems, LARW again dominates RW in all the instances, with a success probability equal to 1. Both algorithms have an improved performance when applied to SAT-encoded graph coloring problems. Taking the medium size instance flat150-545 as an example, we observe that RW achieves a success probability of 0.31 compared to 0.04 when attempting to solve the medium size random instance uf125-538. Similarly, LARW was able to solve the instance flat150-545 within 70,000 local search steps compared to 141,000 when attempting to solve the large random instance uf225-960. The performance difference of the two methods when applied to SAT-encoded graph coloring problems is characterized by a factor up to 65 times in favor of LARW in the number of steps required to find a solution within a given probability, compared to a factor of 16 when applied to Random-3-SAT problem instances.

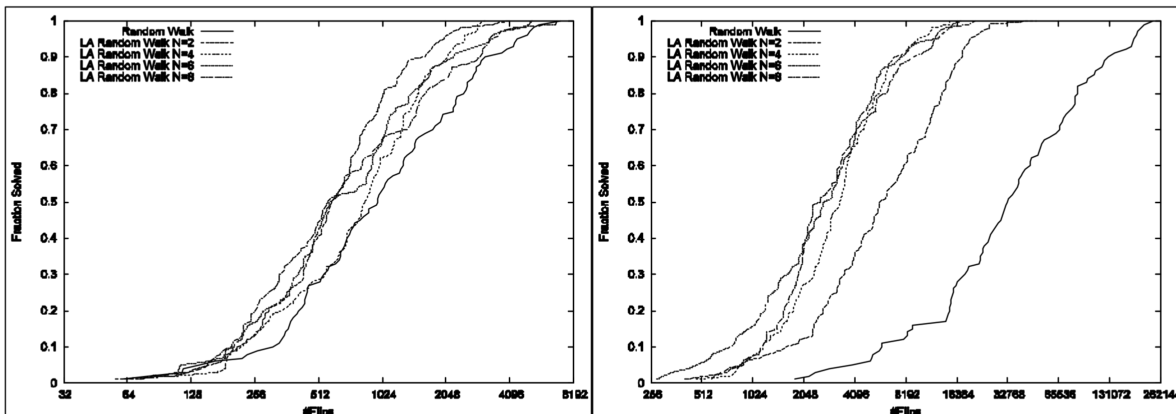

Figure 10: (Right) LARW Vs RW: cumulative distributions for a 48-variable Blocks World problem with 261 clauses (bw-anomaly). (Left) LARW cumulative distribution for a 116-variable Blocks World problem with 953 clauses (medium). Along the horizontal axis we give the number of flips, and along the vertical axis the fraction of problems solved for different values of N.

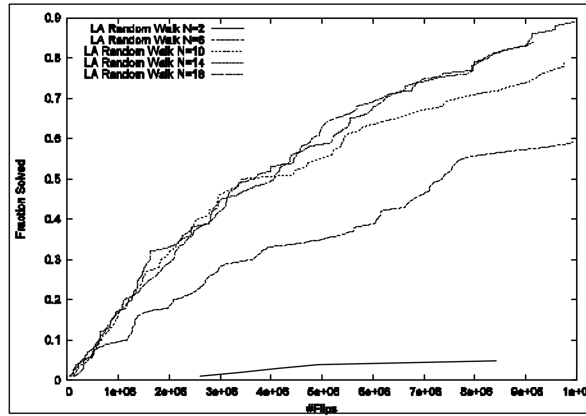

Figure 11: (Right) LARW Vs RW: cumulative distributions for a 459-variable Blocks World problems with 4675 clauses (bw-large.a). Along the horizontal axis we give the number of flips, and along the vertical axis the fraction of problems solved for different values of N.

Finally, we now turn to single SAT-encoded instances from the Blocks World Planning domain. The plots in Figures 10 and 11 indicate that RW is again outperformed by LARW. While both methods reach a probability success of 1, LARW requires a relatively fewer number of steps needed for a finding a solution within a given probability. In the case of the small instance bw-anomaly, the factor is up to ca. 3, while for the medium instance bw-medium, the factor is up to ca. 16. Figure 11 shows the results for the large instance bw-large-a. We observe that LARW achieved a success probability of ca. 0.9, while RW was incapable of solving this instance in any of the 100 trials.

To summarize, in the above these experiments, LARW shows robustness w.r.t. all the instances from the three domains. Obviously, the harder the instance the better it performs compared to RW. The learning automaton mechanism used in RW pays off as the instance gets harder, and the probability of success gets higher as N increases and reaches an optimal value.

4.4 McNemar's Test

We now compare the significance of conclusions made in the previous section (i.e. which of the two algorithms is significantly more successful than the other) using the McNemar test [22]. For each instance, we record the result of each possible run and construct the following contingency table:

N_{00}	N_{01}
N_{10}	N_{11}

Here, N_{10} is the number of times the pair *(success, failure)* have been observed (i.e. success for LARW and failure for RW), and N_{01} is the number of times the pair *(failure, success)* have been observed (i.e. failure for LARW and success for RW), both observed over 100 runs. Thus, experiments that provide the same outcome for both methods, i.e., *(success, success)* or *(failure, failure),* are eliminated. Under the null hypothesis, the two methods should have the same error rate, which means that $E[N_{10}] = E[N_{01}]$. McNemar test is based on the X^2-test for goodness-of-fit that compares the distribution of counts under the null hypothesis to the observed counts. The expected counts under the null hypothesis are:

N_{00}	$(N_{10} + N_{01})/2$
$(N_{10} + N_{01})/2$	N_{11}

The following statistic is distributed as $X^2 = \frac{(|N_{01} - N_{10}| - 1)^2}{N_{01} + N_{10}}$ with one degree of freedom. The McNemar's test accepts the null hypothesis at significance level α if $X^2 \leq X^2_{\alpha,1}$. The value of α is set to 0.05 and $X^2_{0.05,1} = 3.84$.

Problem-Instance	X^2	Two-tailed P value	Null hypothesis
UF125-538	56.017	$P < 0.0001$	Reject
UF225-960	96.010	$P < 0.0001$	Reject
Flat325-1403	68.014	$P < 0.0001$	Reject
Flat600-2237	98.010	$P < 0.0001$	Reject
Bw-large.a	93.011	$P < 0.0001$	Reject

Table 1: Results of McNemar's test.

Hence, we conclude that the difference in performance between the two methods is considered to be extremely statistically significant, leading to the conclusion that LARW performs better than Random-Walk.

4.5 Hardness Distributions

In this section, we focus on the behavior of the two algorithms using 100 instances of size small, medium, and large from Random-3-SAT and SAT-encoded graph coloring problems. The SAT-encoded instances from the Blocks Worlds domain were not included due to the existence of only one instance within any test-set per problem. For each instance the median search cost (number of local search steps) is measured and we analyze the distribution of the mean search cost over the instances from each test-set. The different plots show the cumulative hardness distributions produced by 100 trials on 100 instances from a test-set. Several observations can be made from the plots in Figure 12, which show the hardness distributions for Random-3-SAT.

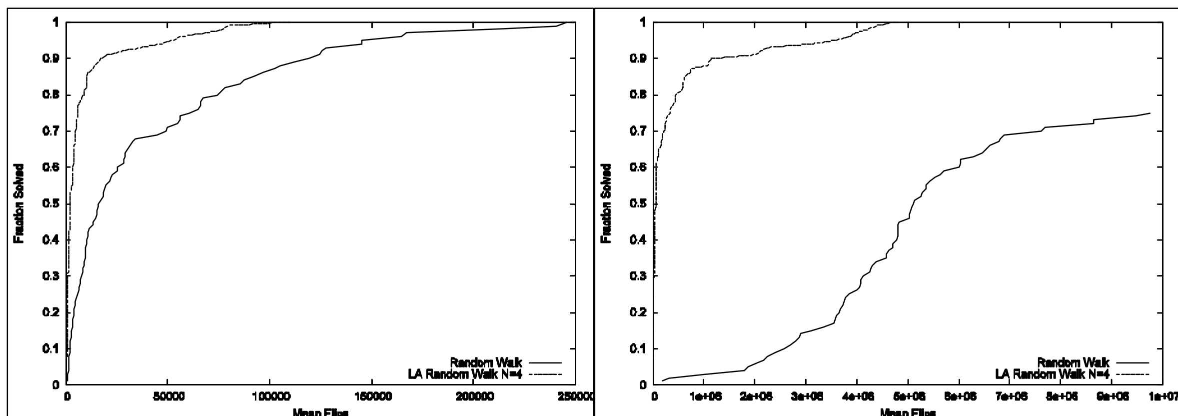

Figure 12: (Left) Hardness distribution across test-set uf50-218. (Right) Hardness distribution across test-set uf125-538. Along the horizontal axis we give the median number of flips per solution, and along the vertical axis the fraction of problems solved.

There exist no cross-over in either of the plots in the figure, which makes LARW the clear winner. Also note that the RW shows a higher variability in search cost compared to LARW between the instances of each test-set. The distributions of the two algorithms confirms the existence of instances which are harder to solve than others. In particular, as can be seen from the long tails of these distributions, a substantial part of problem instances are dramatically harder to solve with RW than with LARW. This observations is already present in the small size test (uf50-218) and is confirmed in the medium size test-set. This confirms the hypothesis that instances with a low solution density (i.e., a few number of solutions) tend to be much easier to solve with LARW than its counterpart. The harder the instance, the higher the difference between the average search

costs of two algorithms (a factor of approximately up to 50). We conjecture that the finite automaton learning mechanism employed in LARW offers an efficient way to escape from highly attractive areas in the search space of hard instances, leading to a higher probability of success as well as reducing the average number of local search steps to find a solution.

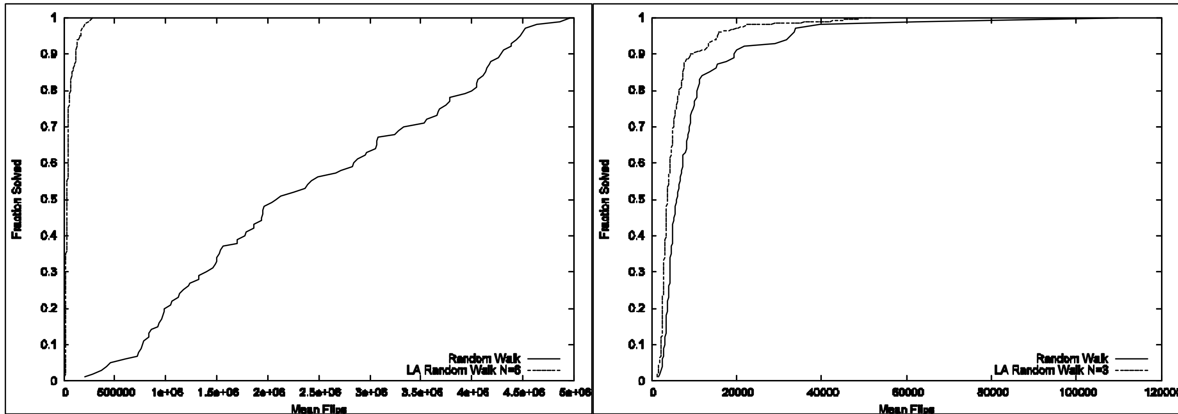

Figure 13: Hardness distribution across test-set flat150-545. (Left) Hardness distribution across test-set for flat90-300. Along the horizontal axis we give the median number of flips per solution, and along the vertical axis the fraction of problems solved.

The empirical hardness distribution of SAT-encoded graph coloring problems in Figure 13 show that it was rather easy for both algorithms to find a feasible solution in each trial across the test set flat90-300, with LARW showing on average a lower search cost within a given probability compared to RW. The plot reveals the existence of some instances on which RW suffers from a strong search stagnation behavior. The plot located on the left of Figure 13 shows a striking poor average performance of RW compared to LARW on the test set flat150-545. Conversely, LARW shows a consistent ability to find solutions across the instances on this test set. For LARW, we observe a small variability in search cost indicated by the distance between the minimum and the maximum number of local search steps needed to find a solution. The differences in performance between these two algorithms can be characterized by a factor of ca. 10 in the median. The performance differences observed between the two algorithms for small size instances are still observed and very significant for medium size instances. This suggests that LARW is considerably more effective for larger instances.

4 Conclusions

In this work, we have introduced a new approach based on combining learning finite automata with random walk. Thus, in order to get a comprehensive picture of the new algorithm's performance, we used a set of benchmark problems containing different types of problem instances, including sets of randomly generated SAT instances and other structured SAT-encoded problems from other domains. All the selected problem instances have been widely used by researchers in the context of evaluating the performance of metaheuristics. RW suffers from stagnation behavior which directly affects its performance. This same phenomenon is however observed with LARW only for large instances. Based on the analysis of RLD's, we observe that the probability of finding a solution within a any arbitrary number of search steps is higher compared to that of RW. To get an idea of the variability of the solution cost between the instances of the test sets, we analyzed the cumulative distribution of the median search cost. Results indicated that the harder the instance, the higher the difference between the average search costs of two algorithms. The difference can be several order of magnitude in favor of LARW. An obvious topic for further work would be the combinations of finite learning automaton with Walk-SAT and GSAT variants.

References

[1] S. A. Cook, The complexity of theorem-proving procedures, *Proc. of the Third ACM Symposium on Theory of Computing*, ACM Press, 151-158, 1971.

[2] M. Davis and H. Putnam, A computing procedure for quantification theory, *Journal of the ACM*, 7:201-215, 1960.

[3] B. Selman, H. Levesque, and D. Mitchell, A New Method for Solving Hard Satisfiability Problems, *Proc. of AAA'92*, MIT Press, 440-446, 1992.

[4] B. Selman, Henry A. Kautz, and B. Cohen, Noise Strategies for Improving Local Search, *Proc. of AAAI'94*, MIT Press, 337-343, 1994.

[5] D. McAllester, B. Selman, and H. Kautz, Evidence for Invariants in Local Search, *Proc. of AAAI'97*, MIT Press, 321-326, 1997.

[6] F. Glover, Tabu Search-Part 1, *ORSA Journal on Computing*, 1(3):190-206, 1989.

[7] P. Hansen and B. Jaumand, Algorithms for the Maximum Satisfiability Problem, *Computing*, 44:279-303, 1990.

[8] I. Gent and T. Walsh, Unsatisfied Variables in Local Search. In *Hybrid Problems, Hybrid Solutions*, IOS Press, 73-85, 1995.

[9] L. P. Gent and T. Walsh, Towards an Understanding of Hill-Climbing Procedures for SAT, *Proc. of AAAI'93*, MIT Press, 28-33, 1993.

[10] B. Cha and K. Iwama, Performance Tests of Local Search Algorithms Using New Types of Random CNF Formula, *Proc. of IJCAI'95*, Morgan Kaufmann Publishers, 304-309, 1995.

[11] J. Frank, Learning Short-term Clause Weights for GSAT, *Proc. of IJCAI'97*, Morgan Kaufmann Publishers, 384-389, 1997.

[12] W. M. Spears, Simulated Annealing for Hard Satisfiability Problems, Technical Report, Naval Research Laboratory, Washington D.C., 1993.

[13] A. E. Eiben and J. K. van der Hauw, Solving 3-SAT with Adaptive Genetic Algorithms, *Proc. of the 4th IEEE Conference on Evolutionary Computation*, IEEE Press, 81-86, 1997.

[14] D. S. Johnson and M. A. Trick, editors, Cliques, Coloring, and Satisfiability, Volume *26 of DIMACS Series on Discrete Mathematics and Theoritical Computer Science*, American Mathematical Society, 1996.

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

[16] M. A. L. Thathachar and P. S. Sastry, *Network of Learning Automata: Techniques for Online Stochastic Optimization*, Kluer Academic Publishers, 2004.

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

[18] B. J. Oommen and D. C. Y. Ma, Deterministic Learning Automata Solutions to the Equipartitioning Problem, *IEEE Transactions on Computers*, 37(1):2-13, 1988.

[19] W. Gale, S.Das, and C.T. Yu. Improvements to an Algorithm for Equipartitioning. IEEE Transactions on Computers, 39(5):706-710, 1990.

[20] B. J. Oommen and E.V. St. Croix, Graph partitioning using learning automata, *IEEE Transactions on Computers*, 45(2):195-208, 1996.

[21] B. J. Oommen and E.R. Hansen, List organizing strategies using stochastic move-to-front and stochastic move-to-rear operations, *SIAM Journal on Computing*, 16:705-716, 1987.

[22] B. S. Everit, *The analysis of contigency tables*, Chapman and Hall, London, 1977.