

Multilevel Stochastic Local Search for SAT

Camilo Rostoker
University of British Columbia
Department of Computer Science
rostokec@cs.ubc.ca

Chris Dabrowski
University of British Columbia
Department of Computer Science
kdabrows@cs.ubc.ca

April 22, 2005

Abstract

Satisfiability (SAT) problems are a hot topic in the field of combinatorial optimization, having strong theoretical foundations as well as many practical applications. Moreover, approximate SAT algorithms have gained widespread attention because they offer a computationally feasible approach to finding high-quality solutions to NP-hard problems in a scalable and efficient manner. Complete SAT algorithms, which are guaranteed to find a solution in bounded time (if one exists), are particularly important when a guaranteed solution is required, but are of limited practical use because of exponential worst-case bounds. In this paper we propose a Multilevel Stochastic Local Search (MLSLS) algorithm that combines power of a complete algorithm with the efficiency and scalability of a Stochastic Local Search (SLS) method. We probabilistically assign values to a varying-sized subset of the variables and then use an efficient unit-propagation algorithm to produce a coarsened approximation of the original problem instance. We then construct an initial candidate solution from the reduced SAT formula and perform a subsequent local search using a state-of-the-art stochastic local search algorithm. Results achieved from our experiments showed our algorithm to be slower than the competitors, but through empirical testing we have determined the bottleneck to be the systematic component, and with future work directed at improving its efficiency, we believe that positive results are a realistic possibility.

1 Introduction

Many research avenues have focused on finding efficient methods for solving Boolean satisfiability (SAT) problems, a fundamentally important topic in combinatorial optimization. As a result, a wide variety of techniques have been developed, all of which can be classified as either complete or approximate. Complete methods systemically examine the entire solution space defined by the problem instance and are guaranteed to find the optimal solution (if one exists) in bounded time, or otherwise return that the formula is unsatisfiable[9]. These algorithms are appropriate when the problem can be solved in polynomial-time, or when there are no resource constraints (i.e. time or compute cycles), but for large problem instances, or when there are resource constraints, complete algorithms scale exponentially and are therefore of limited practical use[3]. Because of this, approximate search algorithms have become the focus of attention as they offer a means of specifying a trade-off between the degree of accuracy and efficiency.

More recently, advances in local search methods have proven to be state-of-the-art for a large variety of SAT problems. Local search methods include a wide variety of techniques that exploit certain properties of the underlying search space topology to find the optimal value within a given subspace (neighborhood); this is referred to as the local optimum. Local search algorithms that incorporate randomness are more commonly known as Stochastic Local Search (SLS) algorithms. SLS algorithms generally involve taking a candidate solution and performing some sort of perturbation which results in one or more new candidate solutions. An evaluation function is then used to determine which of the candidate solutions should be accepted. All state-of-the-art SLS algorithms include two types of mechanisms within the search strategy, namely intensification and diversification. Intensification is a means of greedily improving solution quality within a small area of the search space for a local optimum, while diversification helps to prevent stagnation by ensuring that the search process explores a broad area of the search space, rather than a confined area that may contain only suboptimal solutions [10]. Incorporating some form of randomness has proved to be an efficient diversification mechanism, while intensification can be achieved through a variety of techniques such as Iterative Improvement or the selection step in a genetic algorithm.

One of the main decisions when designing a search algorithm is the relative amount of effort to spend thoroughly exploring the entire search space relative to the amount of effort exploiting small subspaces to find local optima. One approach that has recently gained popularity is the idea of a multi-level stochastic local search. The main idea behind this approach is to perform a rough search in order to produce an acceptable candidate solution as a starting point for a subsequent local search (in contrast to other methods such as random initialization or greedy construction). The multilevel paradigm was first introduced by Barnard and Simon [1] to speed up their spectral bisection methods. Shortly after, Hendrickson and Leland [6] augmented this strategy with a local search algorithm, demonstrating the first generalized methods for multilevel combinatorial optimization. Since then, multilevel methods have been successfully applied to problems such as the Graph Partition Problem (GPP), Graph Coloring Problem (GCP) and the Traveling Salesperson Problem (TSP) [21]. The main reason why these problems were found to benefit from the multilevel paradigm is that they exhibit a natural structure from which the problem instance can be continuously coarsened or refined, which translates to switching between large-scale exploration (diversification) and local search (intensification).

2 Existing Work

Multi-level SAT methods are a relatively new concept to the field of combinatorial optimization, and as such, related work in this area has resulted in a diverse range of approaches.

Several hybrid approaches that combine elements of Stochastic Local Search and systematic exploration have yielded a variety of algorithms that have both strengths and weaknesses. For example, Constrained Local Search (CLS) by Prestwich can be seen as a stochastic local search in a constrained space[15]. More specifically, CLS performs a randomized backtracking to escape from local minima, and like pure systematic approaches, it is *constructive*: it violates no constraints and therefore can exploit techniques such as forward look-ahead. The systematic component of our algorithm can be seen as a generalization of this since we use Satz’s look-ahead technique[13] with backtracking; however, our algorithm only assigns a subset of the variables in this fashion before allowing the underlying SLS procedure to take control.

In another hybrid approach called WalkSatz[5], a WalkSat[17] procedure is performed at each

node of a DPLL search tree. The key to this approach is that they use Satz’s powerful branching rule[13] to provide a valid path to a solution, then create a literal implication graph of the current DPLL node, and use that as the sub-problem to solve by the underlying WalkSat procedure. The intuition here is that the strongly connected components in the implication graph are essentially equivalence classes and can be substituted by a unique literal in order to reduce the size of the formula that WalkSat must solve. The result of this WalkSat procedure is used by the DPLL procedure in calculating a branching point based on a Satz rule.

Some of the most recent research in this area are SLS algorithms incorporating Markov-based Multiresolution Analysis [4, 18]. Until recently, multiresolution analysis was mainly used in such research fields as signal processing, compression, numerical analysis and computer graphics [4]. In its simplest form, the goal of multiresolution analysis is to recursively coarsen to create a hierarchy of approximations to the original problem [21]. In most cases, wavelet theory is adopted which permits a natural definition of a function that decomposes the search space into arbitrarily coarse approximations (thus, in this context, the term multiresolution is synonymous to multi-level). In the context of SAT problems, this function can be considered as the neighborhood selection function. However, it has been empirically shown that these methods are generally most efficient if that target element(s) are located in high-average fitness regions of the search space [18].

In another similar line of work, researchers have shown a great potential for solving large-scale, complex combinatorial problems by harnessing the power of multiple cooperating agents. Toulouse et al. have developed a series of parallel stochastic local search methods called ”Multi-Level Cooperative Search”, where individual agents (i.e. the search programs) work in parallel to implicitly decompose the search space according to their specific search strategies. The agents interact by exchanging information in order to exploit previously explored areas of the search space [20]. The collective information is gathered and strategically reorganized using techniques from Scatter Search[2] and Vocabulary Building[2]. While there seems to be great potential in utilizing the collective intelligence of multiple communicating search agents, the task of engineering efficient control structures is an ongoing research area.

3 Algorithm Description

In our proposed algorithm we wish to exploit variable dependencies by performing a Satz look-ahead procedure[13] and a heuristic based on Unit Propagation, the UP heuristic[12], to ”drill-down” to a certain level in the search tree in order to minimize the size of the search space for the underlying SLS procedure. We do this by iteratively assigning truth values to a subset of variables, calculated in such a way as to detect failed literals as early as possible, thus reducing the amount of backtracking needed. The selection and ordering of this subset, is based on the $PROP_z$ heuristic[13] (refer to Section 3.1) and the Satz look-ahead technique respectively. Intuitively, the less variables that are set by the systematic search, the more coarse the approximation is (and vice versa), and the more work the SLS procedure must do to find a local minimum. However, determining the number of variables to set is difficult and varies largely with the problem instance. Therefore, we introduce a simple heuristic that is used to dynamically adjust the size of the variable subset depending on how many truth values were propagated in previous iterations. The pseudo-code for our proposed algorithm is outlined in Algorithm 1.

As outlined in Algorithm 1, we iteratively approximate the problem instance π to some *level* of coarseness by using the $PROP_z$ heuristic and Satz look-ahead technique. Each time a new variable

Algorithm 1 Multilevel Stochastic Local Search for SAT

input: problem instance $\pi \in \prod$
output: solution $\hat{s} \in S(\pi)$ **or** \emptyset
while termination criteria not satisfied **do**
 $level \leftarrow$ choose number of variables to assign
 $\pi_r \leftarrow \pi$
 $currentLevel \leftarrow 0$
 while $currentLevel < level$ **do**
 $VS \leftarrow$ calculate and order variable set using $PROP_z$ heuristic
 $v_{best} \leftarrow$ best variable from VS
 assignTruthValue(v_{best})
 $\pi_r \leftarrow$ unitPropagation(π_r)
 $currentLevel \leftarrow currentLevel + 1$
 end while
 $s_c \leftarrow$ construct(π_r)
 $s_l \leftarrow$ localSearch(π_r, s_c)
 if $f(s_l) < f(\hat{s})$ **then**
 $\hat{s} \leftarrow s_l$
 end if
end while
if $\hat{s} \in S$ **then**
 return \hat{s}
else
 return \emptyset
end if

v_{best} is chosen to be assigned, we perform unit propagation which results in a simplified problem instance π_r . The variable v_{best} , and its truth value, is determined by the Satz look-ahead procedure; more details on this can be found in the following section.

After the systematic process has set *level* variables, we construct an initial candidate solution s_c , and then performs a local search starting in the neighbourhood induced by s_c *w.r.t.* the current step function. The level of coarseness at which the systematic procedure works is directly related to the size of the variable subset: the less variables that are set, the more coarse the approximation is, since fewer dependencies will be discovered which directly translates to a less simplified Boolean formula. Although the level of coarseness can go either way (i.e. variable subset size can increase or decrease), our results indicate that the level usually follows a monotonically decreasing or increasing function and eventually converges at a value specific to the problem instance.

After performing a systematic search using Satz look-ahead and UP heuristic, the result is a partially assigned Boolean CNF formula. We then complete the partial solution by performing a construction step where the remaining variables are assigned truth values, resulting in an initial candidate solution from which a local search is started. Our algorithm uses the "Averaging In" construction heuristic[16] in order to take advantage of information from previous iterations. The intuition behind this technique is that by looking at past incumbent solutions, we can effectively reinforce decisions that previously led to high-quality solutions. More details for this technique are described in Section 3.5.

3.1 Variable Ordering

So far we have only considered one form of variable ordering, which is a heuristic based on unit propagation, or more specifically, a simple look-ahead technique[13]. Given a formula F , the algorithm picks a subset S of variables using a heuristic called *PROP*. We first describe the overall algorithm and then describe *PROP* in detail. The algorithm iterates through all of the variables in S . For each variable x , it computes two modified formula's resulting from performing unit propagation using x and \bar{x} ; i.e $F' \leftarrow \text{UnitPropagation}(F \cup \{x\})$ and $F'' \leftarrow \text{UnitPropagation}(F \cup \{\bar{x}\})$. If both F' and F'' contain an empty clause then F is unsatisfiable with the current assignment of variables and we have to backtrack. If the first variable we consider gives us both F' and F'' with an empty clause then we know that F is unsatisfiable. If F' contains an empty clause then we know that x has to be set to 0. Similarly, if F'' contains an empty clause then we know that x has to be set to 1. If neither F' nor F'' contain an empty clause then a score is assigned to the variable x . If all of the variables have been looked at, the variable x with the greatest score is picked. This variable is randomly set to 0 or 1 and $F \leftarrow \text{UnitPropagation}(F \cup \{\bar{x}\})$ or $F \leftarrow \text{UnitPropagation}(F \cup \{x\})$ respectively. This process is continued for as many iterations as desired, where each iteration corresponds to setting one variable. Intuitively, the more iterations that are performed, the more variables that are set, and thus the less variables that will be left for the local search stage. This setting is a parameter that can be tuned in our algorithm.

The heuristic *PROP* is defined as follows: $PROP(x, i)$ is true iff x occurs both positively and negatively in binary clauses and has at least i binary occurrences in F . Given an integer T , $PROP_z(x)$ is defined as the first of the three predicates $PROP(x, 4)$, $PROP(x, 3)$, *true* (in this order) whose denotational semantics contains more than T variables[13]. According to [13], empirical tests had shown 10 as a good value for T . Basically, S is the subset of variables that appear negatively in at least one binary clause and positively in at least one binary clause, and appear in a total of at least i binary clauses, such that there is at least T variables satisfying this

condition. In our algorithm, we do not use 4 and 3 for the values of i as originally proposed. Through empirical analysis, we have found that with larger instances, using 4 and 3 for i creates a set S so big that it defeats the purpose, since it doesn't narrow our choice down much at all. Therefore we have been experimenting with setting i higher, such as 50 and 35.

One more definition that is needed in calculating the score of a variable in the algorithm is $diff(F_1, F_2)$, which is a function which returns the number of clauses of minimum size in F_1 but not in F_2 . The pseudo-code for the algorithm is given below. It is taken directly from [13].

Algorithm 2 Variable Ordering Algorithm

```

for each free variable  $x$  such that  $PROP_z(x)$  is true do
  let  $F'$  and  $F''$  be two copies of  $F$ 
   $F' := \text{UnitPropagation}(F' \cup \{x\})$ 
   $F'' := \text{UnitPropagation}(F'' \cup \{\bar{x}\})$ 
  if both  $F'$  and  $F''$  contain an empty clause then
    do backtracking procedure
  end if
  if  $F'$  contains an empty clause then
     $x := 0$ 
     $F := F''$ 
  end if
  if  $F''$  contains an empty clause then
     $x := 1$ 
     $F := F'$ ;
  end if
  if neither  $F'$  nor  $F''$  contains an empty clause then
     $w(x) := diff(F', F)$ 
     $w(\bar{x}) := diff(F'', F)$ 
     $H(x) := w(\bar{x}) * w(x) * 1024 + w(\bar{x}) + w(x)$ 
  end if
end for
Choose variable  $x$  such that  $H(x)$  is greatest

```

3.2 Unit Propagation

Right now our algorithm has a very naive form of unit propagation implemented. Basically, the algorithm looks at every clause, and if the clause has not been satisfied it looks at the number of unassigned literals. If there is only one unassigned literal, then the function sets the literal accordingly and repeats the process until either all of the clauses have been satisfied or there are no more unit clauses. Our original intent was to implement zChaff's lazy data structure approach to efficient Unit Propagation; however, this was never achieved in the current version of our algorithm. Please see the Discussion and Future Work section for more details related to this issue.

3.3 Level Selection Mechanism

The Level Selection Mechanism (LSM) is invoked at the end of every systematic run to determine the number of variables to assign in the next iteration. Given the total number of variables n , number of variables to assign a , and the total number of variables assigned t . We calculate $X = t/n$, the percentage of the total variables assigned, and $Y = t/a$, the percentage of variables that were set compared to the number we specified to set. Our rule is that if $X < 0.05$ and $Y > 10$, we increment n by one, otherwise we decrement n by one.

When we freeze variables, the given assignment may not be consistent with an optimal solution, and thus when freezing more variables we increase the probability of this occurring. Therefore, we try to keep X small, i.e. $X < 0.05$.

The calculation of Y attempts to determine how structured the instance is. If an instance is structured, then Y should be greater than 10. If Y is less than 10, then the instance can be considered unstructured, and thus we want to set as few variables as possible. The intuition here is that if it is an unstructured instance, then we would like the SLS algorithm to do most of the work. Figure 1 shows how our algorithm performs with and without the Level Selection Mechanism. The values in the legend correspond to the initial value for the number of variables to set. These RTDs are averaged over 50 runs, and the implementation details and execution environment are described in section 4.

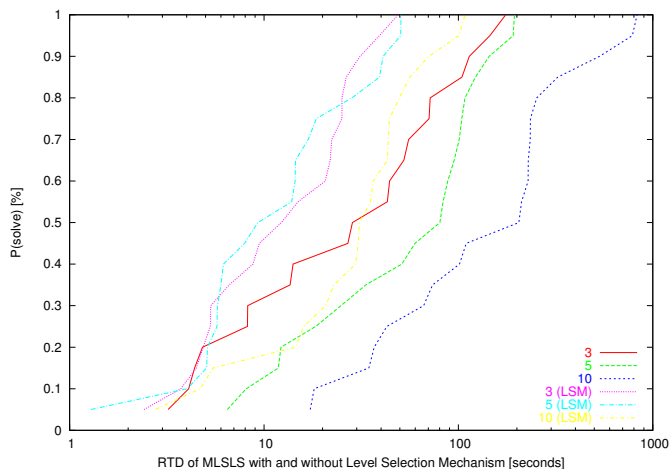


Figure 1: RTD's for MLSLS with/without Level Selection Mechanism on the logistics.d problem instance

Figure 1 shows the RTDs of our algorithm with different starting *levels*, with and without the Level Selection Mechanism. This indicates robustness in our algorithm, as it becomes less important to know *a priori* the optimal *level* for different problem instances. In other words, our Level Selection Mechanism provides robustness by dynamically adapting the *level* at which our systematic algorithm works in order to optimize the balance between the systematic and SLS components for different types of problem instances.

3.4 Backtracking

Our algorithm backtracks when the current assignment of variables leads to an empty clause. That is, if the number of variables to assign is a and we have assigned some $b < a$ variables, and when searching for the $b + 1$ st variable to assign we encounter a situation where no matter if we set a variable to 0 or 1 we get an empty clause, then we have to backtrack. Backtracking is achieved by choosing the last variable to be assigned and flipping its truth value. Once a variable has been assigned both 0 and 1, we then flip the variable that was set prior to that one. This is achieved by keeping track of the number of times a variable has been flipped; we call this the *branching factor*. When a variable is assigned for the first time, its branching factor is set to one. When the same variable is chosen as a branch point during the backtracking process, the branch factor becomes two. While backtracking, if a variable has a branch factor of two, we unset it, as well as all the variables that were propagated because of it, and then continue backtracking to the variable before it. If we have backtracked to the first variable after assigning both values for it, then we know that the instance is unsatisfiable. This gives our algorithm the property that if the number of variables to assign is equal to the total number of variables, then the systematic component of our algorithm is complete.

3.5 Construction Heuristic

Once the instance has been reduced by Satz and unit propagation, a construction heuristic is used to generate an initial candidate solution. Our algorithm employs the "Averaging In" construction heuristic, proposed by Selman and Kautz as a domain-independent extension to GSAT for solving large structured SAT problems[16]. The basic idea of this technique is to exploit information gathered from previous iterations in order to guide the construction process. More specifically, after each local search, a bitwise average of the resulting candidate solution and the previous best solution is performed. A bitwise average of two truth assignments is an assignment which agrees with the assignment of those letters on which the two given truth assignments are identical; the remaining letters are randomly assigned truth values. For example, if $S_1 = 101001001$ and $S_2 = 001010011$ then the bitwise average, S_a would be 01001 . A * indicates that the variable is assigned a random truth value. Because this process biases specific regions of the search space, it is important to ensure that we do not force the search to get trapped in sub-optimal local minimum; however, this issue is trivially addressed by the fact that the diversification mechanism within the SLS procedure allows the search to escape from a local optimum. Due to the fact that some of the variables will be "frozen" during the construction step and subsequent local search stage, our algorithm incorporates a small modification to the Averaging In strategy. In our modified version, only the free variables (i.e. not frozen) will be included in the bitwise average computation. Algorithm 3 outlines this approach. Through careful empirical analysis, we have concluded that this strategy does not offer significant advantages, at least for the smaller problem instances that we tested. Figure 2 illustrates the performance of our algorithm with and without the Averaging In construction heuristic. These RTDs are averaged over 50 runs, and implementation details and execution environment are described in section 4.

3.6 Stochastic Local Search

One of the main purposes of our proposed algorithm is to explore how Stochastic Local Search can be effectively combined with systematic search. This is achieved by applying systematic search up

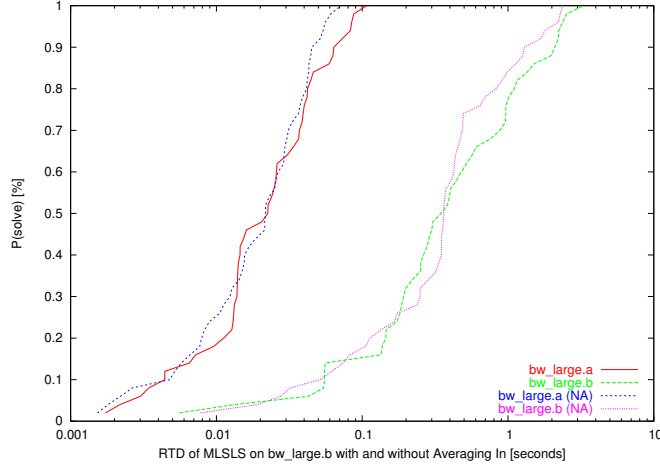


Figure 2: Effects of the "Averaging In" Construction Heuristic

Algorithm 3 Averaging In Strategy

$T_1^{init} \leftarrow$ assign all unset variables using random construction
 Do a round of systematic with Satz look-ahead starting with T_1^{init}
 $T_2^{init} \leftarrow$ bitwise average of T_1^{init} and T_1^{best} , excluding frozen variables
 Do a round of systematic with Satz look-ahead starting with T_2^{init}
while (termination criteria not satisfied) **do**
 $T_i^{init} \leftarrow$ bitwise average of T_{i-1}^{best} and T_{i-2}^{best} , excluding frozen variables
 Do a round of systematic with Satz look-ahead starting with T_i^{init}
end while

to a specified "level", and then performing a subsequent local search to explore a confined region of the search space. Our algorithm utilizes Novelty+, a relatively recent high-performance SLS algorithm[7]. One of the main reasons for choosing Novelty+ was that it was easy to integrate into our algorithm. Regardless of what algorithm we might have chosen, a minor modification was necessary, which involves checking the flip candidate after each step to ensure that it was not "frozen" by the prior systematic search stage. Novelty+ uses a history-based, randomized greedy mechanism for selecting a variable to flip and uses the same scoring function as GSAT[10].

As shown in Algorithm 4, Novelty+ either performs a random walk with probability *randwalk*, or it uses the GSAT scoring function, which tries to minimize the number of unsatisfied clauses that would occur if the variable was chosen, to determine a variable to flip from a randomly selected clause. If the best variable is not the youngest, it is selected immediately – this is the history-based part of the algorithm. Otherwise, with probability *noise*, we choose the second best variable and with probability $1 - noise$ choose the best variable.

The inclusion of a check for frozen variables was straightforward, and was done so in a way to avoid destroying the underlying semantics of the original algorithm. To accomplish this, we essentially ignore frozen variables when determining best and second best variables; thus, we can guarantee the chosen variable is never frozen, and the best free variable (i.e. unfrozen) is selected.

Algorithm 4 Novelty+ Step Function

```
if RandomProbability(randwalk) then
  C  $\leftarrow$  choose a false clause uniformly at random
  flipCandidate  $\leftarrow$  choose a variable in C uniformly at random
else
  C  $\leftarrow$  choose a false clause uniformly at random
  bestVar  $\leftarrow$  variable in C that minimizes the number of unsatisfied clauses if flipped
  secondBestVar  $\leftarrow$  next best variable in C after bestVar
  youngestVar  $\leftarrow$  variable in C that was most recently flipped
  if bestVar  $\neq$  youngestVar then
    flipCandidate  $\leftarrow$  bestVar
  else
    if RandomProbability(noise) then
      flipCandidate  $\leftarrow$  secondBestVar
    else
      flipCandidate  $\leftarrow$  bestVar
    end if
  end if
end if
```

instance(s)	level	maxRunStepsPerIter	# of frozen picks	# of steps	% of frozen of picks
uf250	2	2000000	68	582832	0.01164
uf250	5	2000000	70	604924	0.01149
uf100	2	2000000	195	1036870	0.01882
uf100	5	2000000	207	1003293	0.02065
logistics.d	10	1000000	2	814591	0.00146
logistics.d	20	1000000	14	1154374	0.00371
bw_large.b	4	1000000	55	547677	0.01344

Table 1: Statistics on number of frozen variables chosen during local search

If Novelty+ chooses to perform a random walk portion, we just check to see if the selected variable is frozen, and if so, we don't flip it. Although one would tend to think this would result in many "wasted" steps (i.e. no variable is flipped because the chosen one was frozen), empirical observation has shown that frozen variables were selected less than 0.014% out of the total number of steps. Table 1 shows statistics on how many times a frozen variable was selected for different problem instances. The *maxRunStepsPerIter* refers to how many Novelty+ run steps are performed in each iteration before returning control to the systematic search. The entries with uf250 and uf100 are values averaged across the entire instance ensembles, with 100 independent runs performed on each individual instance. The logistics.d and bw_large.b entries are single problem instances averaged over 50 independent runs.

4 Implementation and Execution Environment

Our implementation is built within the UBCSAT framework, an experimental environment for Stochastic Local Search algorithms for SAT and MAX-SAT [19]. UBCSAT provides a flexible, feature-rich environment that includes many highly optimized data structures for efficient implementation, as well as an array of valuable testing and evaluation tools for thorough empirical analysis. More importantly, the UBCSAT system contains implementations of many of the state-of-the-art SLS algorithms, which makes it conveniently straightforward to compare against our own algorithm. Unfortunately, there is no support for systematic solvers, and thus our comparison against complete algorithms is done outside of UBCSAT.

Our experiments were performed by submitting jobs to a compute cluster running a Sun Grid Engine (SGE) version 2.4.1. Cluster nodes had Intel Xeon 3.06GHz CPUs running SuSE Linux 9.1 with kernel version 2.6.5-7.111.30-smp with 2 GB of RAM.

5 Testing and Evaluation

Our test sets include both structured and random instances. In particular, we used instances from the Block's World planning domain, Logistics, Uniform Random and flat graph colouring. All problem instances were retrieved from the SATLIB website[8], the official repository for SAT benchmark problem instances.

We compare our algorithm against current state-of-the-art algorithms for both random and structured instances. In particular, we choose Novelty+[7] and SAPS[11] as the SLS algorithms to compare against, and Satz[13] and zChaff[14] for the systematic algorithms. We chose Novelty+ and Satz because we use both as a foundation to our algorithm, while zChaff and SAPS because they are state-of-the-art, and zChaff is the most recent winner of the SAT challenge in the structured category. The source code for both Satz and zChaff was easily accessible from the authors website, while SAPS and Novelty+ are conveniently included in the UBCSAT.

We performed many experiments on a variety of different instances and generated Run-Time Distributions (RTDs) and Run-Length Distributions (RLDs) in order to compare against both systematic and SLS algorithms. We also show the RTDs averaged across the entire ensembles for uf100, uf250 and flat200, which characterizes the variation of run-time over a distribution of problem instances[10]. Because our algorithm combines both systematic and SLS methods, we use a cut-off time as the termination criteria rather than search steps.

5.1 Empirical Analysis

We believe that our algorithm is Probabilistically Approximately Complete (PAC). That is, it is guaranteed to solve any solvable problem instance with arbitrarily high probability when allowed to run long enough. We believe our algorithm has the PAC property because when we set variables, unless a variables must be set to 0 or 1 to avoid an empty clause, it is randomly set to 0 or 1. Therefore any combination of values for the variables is possible unless that combination leads to an empty clause which could not be a solution anyway. Once we have set a number of variables we freeze them and let the SLS algorithm run. The SLS algorithm cannot flip any of the frozen variables but if we let it run long enough and the frozen variables are consistent with a solution it will find the solution. If the frozen variables are not consistent with a solution then eventually we will stop the SLS algorithm and run the systematic again. Since the systematic part of our algorithm can set the variables to any possible consistent combination, eventually one will be found and the SLS algorithm will be run and find a solution.

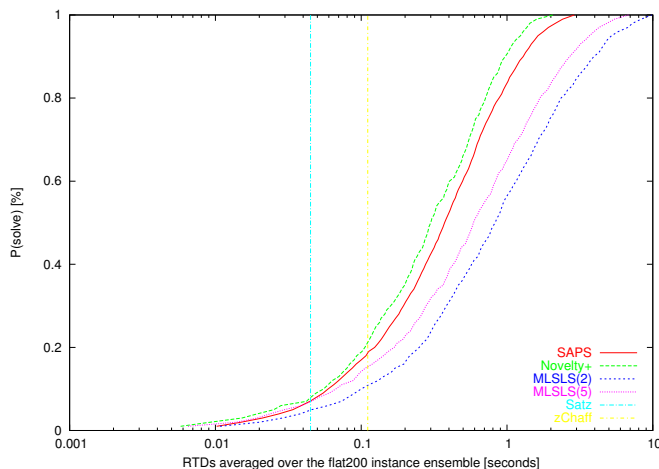


Figure 3: RTDs averaged across the flat200 instance ensemble

As shown in the results, our algorithm is always outperformed by SAPS, zChaff and Satz, but is sometimes competitive with Novelty+. It appears that in most cases any benefit that the systematic component of the algorithm may provide to the subsidiary local search is not justified by its lengthy running time. As further proof to this argument, refer to Figure 6, which shows RTDs for MLSLS on the logistics.d test instance, starting at different levels, averaged over 50 independent runs. As the figure shows, the algorithm performs better when less variables are set (i.e. small level), indicating again that the systematic component is the bottleneck of the algorithm.

One result that shows promise is the RTD for the flat200 instance ensemble (Figure 3), where the distribution curves for our algorithm do not contain "humps", as observed in the other figures. This indicates a potential for achieving an exponential distribution for the flat200 problem instances with optimal parameter settings. Figure 4 and 5 are RTDs of uniform random instances with 250 and 100 variables respectively, obtained by averaging corresponding quantiles from 100 independent runs per instance in the ensemble (Figure 3 was also obtained using this method). In both cases, Novelty+ and SAPS perform as expected with smooth exponential RTDs. On the other hand, the RTDs for our algorithm exhibit a multi-modal behaviour. We attribute this to the systematic component

of our algorithm, where each of the "humps" indicates the span of one or more systematic search phases. Intuitively, when the systematic search is running, the run-time is constantly increasing but the solution quality is not, since a full variable assignment is never reached until the construction step is called at the end of the systematic search stage.

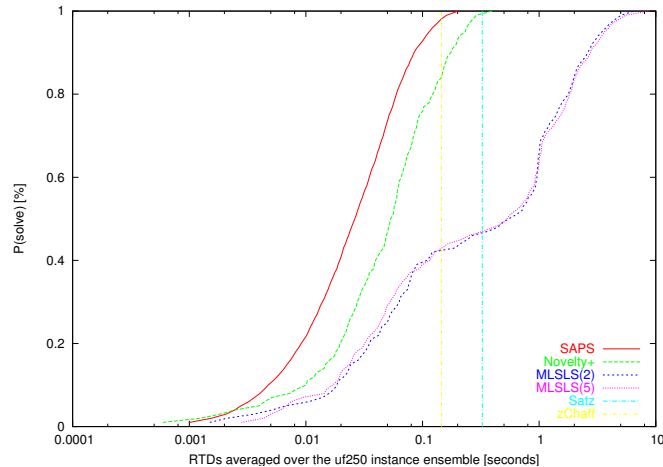


Figure 4: RTDs averaged across the uf250 instance ensemble

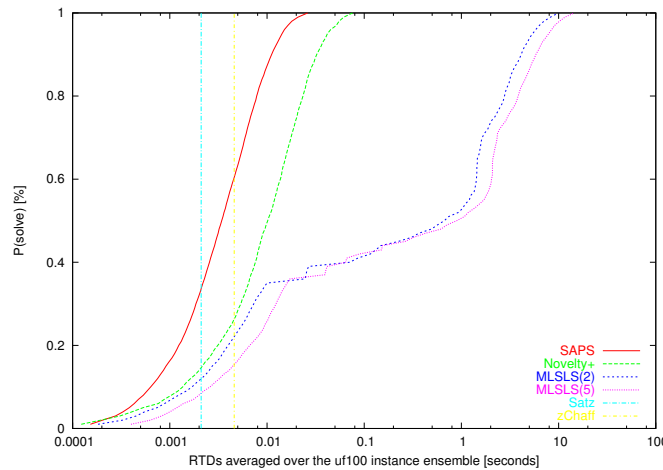


Figure 5: RTDs averaged across the uf100 instance ensemble

While most of our experiments focused on RTDs, in order to include systematic algorithms, we analyzed several RLDs in order to better understand what benefits, if any, the systematic component of our algorithm is providing for the underlying local search. Figure 7 shows an RLD of our algorithm against Novelty+ on the logistics.d problem instance. This shows that MLSLS takes fewer steps to find higher-quality solutions, indicating that the systematic component reduced the formula in such a way that the subsidiary SLS procedure needed less steps than Novelty+ did to find similar solutions.

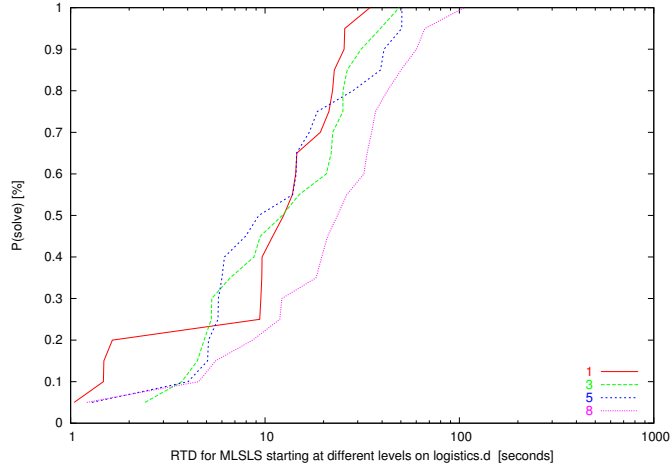


Figure 6: RTDs for MLSLS with varying starting levels on logistics.d

Figure 8 shows RTDs on the `bw_large.b` problem instance. While SAPS and Satz easily outperform their rivals, MLSLS seems to outperform Novelty+ on runs that took longer than 0.06 seconds. The problem instance is structured, and thus we believe this experiment indicates that the systematic component is exploiting the variable dependencies inherent in the structured instance, and thus providing some benefit to the underlying SLS procedure.

6 Discussion and Future Work

There are a number of improvements and future work considerations that have become apparent throughout the duration of the project. This section briefly lists some of these issues and describes possible avenues for future work. We separate this section into discussion pertaining to the SLS and systematic components of our algorithm.

6.1 SLS Discussion and Future Work

1. One improvement that we thought of is to study the variables that Novelty+ picks when deciding which variable to flip. Sometimes Novelty+ will pick a variable that has been frozen by the systematic part of the algorithm. Currently when this happens we do nothing. It might be a good idea to keep track of which frozen variables are picked by Novelty+. If a variable is picked more than some threshold of times maybe we would allow it to be flipped because this might be indicating that it is set to the wrong value.
2. Another improvement along the same lines would be to maybe have a data structure that only stores the non frozen variables so that Novelty+ would never pick a frozen variable. This would make our code more efficient as we would not be wasting any steps to pick frozen variables. At the same time if we do this then we would not be able to implement the suggestion mentioned above since we would not know which frozen variables Novelty+ would pick if it could. Therefore we could only implement one of these suggestions, but both are worth investigation.

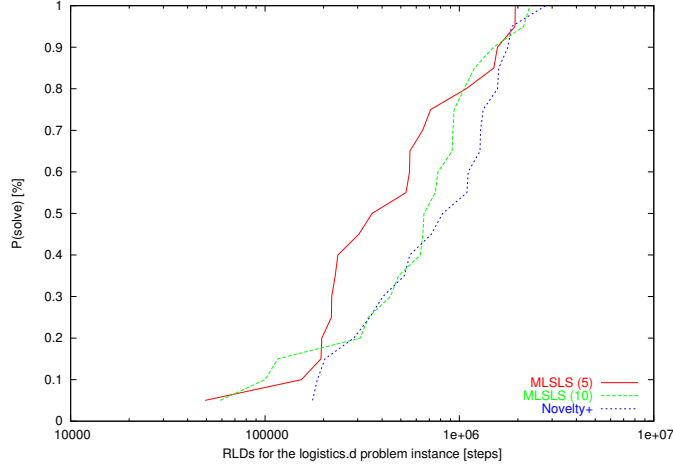


Figure 7: RLDs on the logistics.d instance, averaged over 50 independent runs

3. An obvious improvement would be to try different underlying SLS methods. Right now we have only tried Novelty+ but maybe some other SLS methods would be more effective in this kind of system.

6.2 Systematic Discussion and Future Work

1. One obvious way to speed up our algorithm is implement a more efficient unit propagation algorithm. For instance, by using the unit propagation approach of 'watched literals' implemented in zChaff we anticipate a major speed-up in performance.
2. We have also not done any algorithm profiling. This would be a valuable endeavour as it would show us which parts of the algorithm are the slowest and therefore where we should focus our attention when optimizing the code. This is a very general suggestion for improvement but we feel a necessary one as we did not have any time to try to optimize our code.
3. Right now our level selection mechanism is a rough approximation with very little empirical testing behind it. It would be a good idea to do more testing and come up with a better way to decide the number of variables to assign at each run of the systematic part of our algorithm.
4. The $PROP_z$ heuristic uses two parameters which determine the size of the set to be returned for use by the systematic part of the algorithm. Basically, when the parameters are set to a high value, the resulting set contains fewer variables, as fewer variables meet the necessary criteria. For instance, with many binary clauses we would like these parameters to be high in order to distinguish which variables could most effectively detect failed literals early in the search. If the parameters are too low then too many of the variables meet the criteria. If the parameters are too high then too few of the variables meet the criteria and instead the set of all unfrozen variables is returned. An ideal situation would be to adjust the parameters automatically so that a reasonably sized set is always returned for the systematic search. If the set is too small, then there is not enough opportunity for randomness in the algorithm,

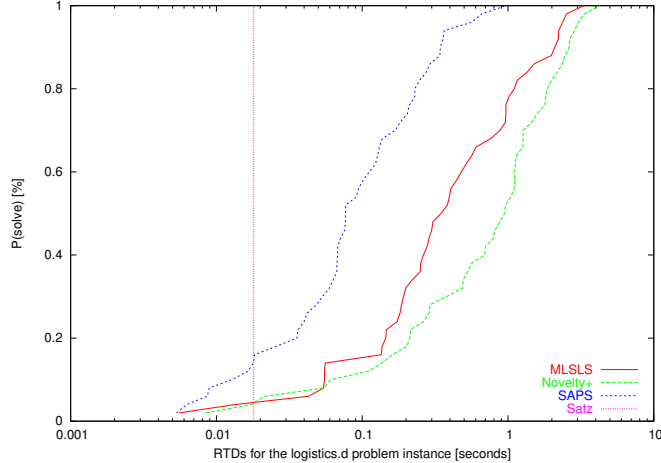


Figure 8: RTDs on the bw_large.b instance, averaged over 50 independent runs

while if the set is too big, then the algorithm has to look through too many variables and the optimal variables are not separated from sub-optimal variables.

5. Another issue is how long to run the SLS component of our algorithm with regards to the other parameters, since we want to allow the SLS stage to run long enough so that it has a good probability of finding high-quality solutions. On the other hand, we do not want to run the SLS procedure for too long, because the frozen variables picked by the systematic search may not be consistent with the global optimum. In this case we want to stop the SLS component so that the systematic search has another chance at setting a different subset of variables. Therefore, the number of steps that Novelty+ performs has to take into account both of these issues and try to balance between them.
6. All of these parameters described above are most likely dependent on each other. Therefore we cannot just find the optimal values for any one parameter individually; we have to test all combinations of these parameters and to determine the relationship between them. This could be done through a rigorous empirical analysis or some other parameter selection mechanism such as automatic parameter tuning.
7. Another improvement concerns our method of backtracking. We believe that more research is necessary to see if a better method of backtracking is possible. One method we considered was backjumping. If we had set variables $v_1, v_2, v_3, \dots, v_n$, and while trying to set the $n + 1$ th variable we encountered an empty clause, we examine the set of variables starting at v_n and see if they were in one of the empty clauses. If some variable v_k was the first variable encountered that was in an empty clause, we unset all variables v_{k+1}, \dots, v_n as well as the variables that were propagated by these variables, flip variable v_k and continue the search from there. The reasoning behind this is that there is no point flipping a variable that is not in an empty clause because it cannot affect the clauses' satisfaction status. This idea was scrapped because we realized that it did not take into account the fact that even if a variable was not in an empty clause, one of the variables that it set through unit propagation might

be in an empty clause. Moreover, when we flip a variable we have to unset all the variables that it previously propagated and run unit propagation again with its new truth value, and thus this process may affect one of the empty clauses even if the variable is not in an empty clause.

7 Conclusion

The initial intent of this project was to design an algorithm that would outperform SLS methods on structured instances, while being competitive on random instances. Similarly, we aimed to outperform complete methods on random instance while being competitive with them on structured instances. Furthermore, we believe such a multi-level approach could serve as a good all around algorithm for situations when the nature of the problem instance at hand is unknown.

The reasoning behind our plan was as follows: If we encountered a structured instance the systematic part of our algorithm would set a number of cleverly chosen variables (using the SATZ heuristic), which would propagate to set (possibly many) more variables. Once all of these variables were set, including the propagated dependent ones, we would pass a much smaller problem to the SLS procedure to solve. Our intuition was that the work performed in the systematic stage of the algorithm would lead to a significant speed up in the SLS algorithm, because some of the dependencies would have been removed, which have been found to cause problems for most SLS algorithms. In this way, the algorithm would be exploiting the structured nature of the instance to help the SLS procedure to find the solution more efficiently. On random instances our algorithm would set a small number of variables and basically let the SLS part do all of the work. The systematic stage would not be essential to the performance of the algorithm, because there would be no dependencies to exploit. We thought this could be achieved by intelligently choosing *a*) the number of variables to assign and *b*) the number of SLS steps to perform, based on the number of variables that were propagated. If a large number of variables were propagated, then this would indicate a structured instance, and thus we would increase the number of variables to assign. If few variables were propagated, this would indicate a random instance and we would decrease the number of variables to assign and increase the number of steps for the SLS procedure.

Based on the empirical results of our tests we did not achieve this goal. All of the existing algorithms we compared against, Novelty+, SAPS, zChaff, and SATZ, outperformed our algorithm on nearly all problem instances - on the random and structured instances. We are still waiting on results for various large (hard) structured instances that were not finished at the time of this writing, but even from the partial results we have seen, it seems like our algorithm was still not as good.

Our conclusion is that the systematic part of our algorithm we implemented was much too slow, and thus did not allow us to truly test our hypothesis. Our algorithm spends most of its time on the systematic part deciding on which variables to set, followed by a call to Unit Propagation. Therefore, we believe that any advantage that might have been gained by setting the variables and propagating the truth values is completely lost due to the fact that the systematic stage of the algorithm is so slow. Our tests have indicated that assigning truth values to less variables in the systematic stage of the algorithm always results in better performance, regardless of the problem instance (refer to Figure 6). In conclusion, although the results obtained so far are relatively poor, we believe that by replacing our Unit Propagation procedure with a more efficient one, and addressing some of the other issues described in the discussion section, there is still hope for this

algorithm to reach our original goals.

References

- [1] S. T. Barnard and H. D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice & Experience*, 6(2):101–117, 1994.
- [2] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268 – 308, 2003.
- [3] Hirsch Edward and Kojevnikov Arist. Unitwalk: A new sat solver that uses local search guided by unit clause elimination. *Annals of Mathematics and Artificial Intelligence*, 43(1):91–111, 2005.
- [4] James. D Fix. Combinatorial optimization through multiresolution analysis. Written for authors general examination, 1996.
- [5] Djamel Habet, Chu Min Li, Laure Devendeville, and Michel Vasquez. *A Hybrid Approach for SAT*, volume 2470. Springer-Verlag, 2002.
- [6] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In S. Karin, editor, *Proc. Supercomputing '95, San Diego*, New York, NY 10036, 1995. ACM Press.
- [7] Holger Hoos. On the run-time behaviour of stochastic local search algorithms for sat. In *Proceedings of AAAI-99*, pages 661–666. MIT Press, 1999.
- [8] Holger Hoos and Thomas Thomas Stützle. Satlib website. <http://www.satlib.org>.
- [9] Holger H. Hoos and Thomas Stützle. Systematic vs. local search for sat. In *KI*, pages 289–293, 1999.
- [10] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search Foundations and Applications*. Morgan Kaufmann, 2005.
- [11] Frank Hutter, Dave A.D. Tompkins, and Holger H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for sat. In *Proceedings of Constraint Programming 2002*, pages 233–248. Springer Verlag, 2002.
- [12] C. Li and Anbulagan. Heuristic based on unit propagation for satisfiability problems. In *In Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 366 – 371, 1997.
- [13] C.M. Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, 1997.
- [14] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 530–535. ACM Press, 2001.

- [15] Steve Prestwich. A hybrid search architecture applied to hard random 3-sat and low-autocorrelation binary sequences. In *The Sixth International Conference on Principles and Practice of Constraint Programming*, volume 1894, pages 337–352. Springer-Verlag, 2000.
- [16] Bart Selman and Henry A. Kautz. Domain-independent extensions to gsat: Solving large structured satisfiability problems. In *IJCAI-93*, 1993.
- [17] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *AAAI '94: Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*, pages 337–343, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.
- [18] Marc Thuillard. Emergence of collective intelligence in stochastic local search-and-optimization systems. In *EUNITE 2003*, 2003.
- [19] Dave A.D. Tompkins and Holger H. Hoos. Ubsat: An implementation and experimentation environment for sls algorithms for sat and max-sat. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, pages 37–46, 2004.
- [20] Michel Toulouse, Krishnaiyan Thulasiraman, and Fred Glover. Multi-level cooperative search: A new paradigm for combinatorial optimization and an application to graph partitioning. In *Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, pages 533 – 542, 1999.
- [21] Chris Walshaw. Multilevel refinement for combinatorial optimisation problems. In *Annals of Operations Research*. Kluwer Academic Publishers, The Netherlands, 2004.