A Comparative Study of Algorithms for Propositional Satisfiability

Ravi Kiran S S and **Gaurav R. Bhaya**
{kiran, gbhaya}@cs.washington.edu
Department of Computer Science and Engineering,
University of Washington.

Abstract

Propositional satisfiability is the problem of determining if there exists an assignment of truth values for which a given propositional formula evaluates to 'true'. Treating this problem as a search through the space of possible assignments, we describe two popular algorithms – DPLL and WALKSAT – as instances of complete and incomplete methods respectively. We examine the effect of incorporating heuristics – reported in literature and our own – by analyzing their performance on various types of problems. In particular, we explore using Genetic Algorithms as a viable bootstrapping mechanism for WALKSAT and provide supporting results.

We verify hypothesis presented in the literature on identifying *hard* problems. We show in our study, that while WALKSAT works well for most satisfiable problems and DPLL works well even for unsatisfiable ones, both these algorithms struggle to find a solution when presented with hard satisfiable problems. This is observed by increased run-times in case of DPLL and decreased correctness in case of WALKSAT. Based on our results, we present some possible directions for future work.

1 Introduction

Propositional satisfiability is the problem of determining, for a formula of the propositional calculus, if there is an assignment of truth values to its variables for which the formula evaluates to *true*. SAT is a common shorthand for referring to the problem of propositional satisfiability in conjunctive normal form (CNF).

The first, and one of the simplest, of the many problems which have been shown to be NP-complete, SAT holds a central position in the study of computational complexity. Consequently, an efficient SAT algorithm is a good pointer to the potential tractability of many practical problems which are currently considered NP-hard. A related aspect which motivates the analysis of SAT problems is the $A.I.$ paradigm of planning. Many problems in planning can be "compiled" into SAT problems. Therefore, a good solution for the SAT version would assist in solving the corresponding planning problem efficiently.

Before proceeding, we provide some terminology and definitions for the SAT problem. A *formula* or a *sentence* is a conjunction of clauses. A clause is a disjunction of literals and a literal is a propositional variable or its negation. Let $U = u_1, u_2, \ldots u_n$ be a set of n boolean variables. A (partial) truth assignment for U is a (partial) function $T : U \rightarrow \{\mathbf{true}, \mathbf{false}\}$. Corresponding to each variable are two literals, u and $\bar{u}$. A literal u ($\bar{u}$) is **true** under T iff $T(u) = \mathbf{true}$ ($T(u) = \mathbf{false}$). We call a set of literals a *clause*, and a set of sequence (tuple) of clauses, a *formula* or a *sentence*. The restriction of SAT to instances where all clauses have length k is denoted $k - SAT$ (Cook & Mitchell 1997). Of special interest are problems in the category $3 - SAT$[1], and for the purposes of our evaluation, we have considered only problems of this category.

A procedure for SAT is *sound* if every input on which it returns **yes** is satisfiable and *complete* if in addition, it returns **yes** (in finite time) on every satisfiable input. In practice, we often need a procedure to return a solution when the input is a satisfiable formula - the *search* or *function* version of SAT. Under the aforesaid taxonomy, therefore, the methods attempting to solve a SAT problem fall into two categories, *complete methods* and *incomplete methods*.

- **Complete methods:** Since resolution is refutation complete, a simple method for resolvents, and then check if an empty clause has been generated. Davis and Putnam (Davis & Putnam 1960) introduced a method in which the variables are eliminated one-by-one from the formula by generating all possible resolvents based on a chosen variable and then deleting all clauses mentioned in that variable, all in one step. Each such step generates a subproblem with one fewer variable, but possibly quadratically more clauses. The above procedure is guaranteed to be complete. However, most implementations typically involve setting up a search tree and traversing it in a depth-first fashion and backtracking as required. For large problem instances, this generates considerable space-time overhead, which not only slows down the search but also limits the size of problems which can be solved using complete methods.
- **Incomplete methods:** As the name suggests, these meth-

[1] 3 is the smallest value of k for which $k - SAT$ is NP-complete.

ods are *not guaranteed* to find a satisfying solution even if one did exist. Typically, these methods employ some notion of a randomized local search[2]. In local search, an objective function is defined over truth assignments such that global minima correspond to satisfying assignments. Most algorithms typically start by guessing the truth values and then try to improve the guess incrementally by checking truth assignments within a neighbourhood of the current one with a lower value for the objective function. Typically, the initial guess is a random truth assignment, the objective function is the number of clauses not satisfied by the current truth assignment, and the neighbourhood is the set of truth assignments at Hamming distance one from the current guess.

Incomplete methods can be thought of as *model finders:* they cannot prove unsatisfiability, but are often much better than the known complete methods at finding satisfying assignments when they exist. In addition, because they involve local computations, they are faster. These attractive features have tended to overshadow the incompleteness factor, leading to an ever-increasing emphasis in SAT related literature – a trend which continues to-date (Selman, Kautz, & Cohen 1996) (Selman, Kautz, & Cohen 1997).

This report is organized as follows: Sections 2 and 3 describe an instance from each of the two methods mentioned above. In particular, we look at DPLL and WALKSAT as instances of complete and incomplete methods respectively. We also try various heuristics – reported from literature and our own – and analyze their effects on the aforementioned algorithms. In Section 4, we briefly describe concepts central to Genetic Algorithms and illustrate how they can be used to bootstrap DPLL and WALKSAT. In Section 5, we describe the results of various experiments and analyze the performance of the algorithms. Section 6 provides the conclusion and directions for future research. Appendix A describes the work done by each of the authors.

2 DPLL algorithm

The Davis-Putnam algorithm (DPLL) –named after its authors, Davis, Putnam, Logemann, and Loveland (Davis & Putnam 1960)– is a commonly used complete backtracking algorithm for boolean satisfiability problems. It improves the traditional depth first search over the variables of the problem by using various simple search space pruning techniques described below:

- **Early Termination:** The DPLL algorithm determines the truth value of a *sentence* for a partial assignment of its variables. In particular, if a literal is assigned to be true any clause that contains the literal is true irrespective of the assignment to its other variables. Furthermore, if any clause in a sentence is false, then the entire sentence is false, irrespective of all other clauses in the sentence. The DPLL algorithm uses Early Termination to avoid examination of entire subtrees in the search process.

- **Pure symbol heuristic:** A symbol is said to be pure if it is present in the same sign that it appears in. For example, in the sentence $(A \vee B) \wedge (A \vee \neg C) \wedge (\neg B \vee C)$ A is a pure symbol as it always appears in the form A and never as $\neg A$. According to the Pure symbol heuristic, assignment of true to the pure symbol literal (i.e., the literal form in which the literal appears) cannot make any clauses false. Moreover, the Pure symbol heuristic can ignore the clauses which are already true at the current depth of the search tree.

- **Unit clause heuristic:** A unit clause is a clause which has only one literal which must be assigned true in order to make the clause true. In particular, this includes all clauses that contain only a single literal or clauses in which all but one literal have been assigned false in the partial assignment in the search space. The unit clause heuristic assigns all such symbols to be true before branching on the remaining symbols. Note that assigning a unit clause can create another unit clause. This cascade of forced assignments is called as *unit propagation*. This process represents forward chaining in horn clauses.

We shall now discuss the data structure that we used in order to enable the DPLL algorithm to execute efficiently.

Data Structures for DPLL The data structures that we build primarily focus on making individual operations performed by DPLL more efficient. In particular, the data-structures presented in this section enable quick identification of clauses that are affected by assignment to the literal. Furthermore, they aid in identification of unit and pure literals without searching through all the literals in individual clauses. On the other hand, every backtrack operation require restoring the data structure to its original state. The individual components of the data structure, as shown in Figure 1 are created as follows:

- **Structure for each Symbol:** The DPLL algorithm essentially selects a literal and then propagates assigned value to individual clauses. In doing so, each literal requires to modify the *status* of each clause it occurs in. Further, some heuristics (discussed later) require the knowledge of the number of clauses in which the symbol occurs in either as the literal itself or its complement. Therefore, the data structure built for each symbol provides the information in the form of a table lookup. Each symbol provides the count of clauses that contain the literal and the number of clauses that contain the complement of the literal. In addition, it provides *pointers* to all clauses in which the symbol occurs. (This structure is modified as each symbol gets assigned.)

- **Structure for Clauses:** The DPLL algorithm requires information such as the unassigned literals in each clause and the status of the clause (such as, clause is satisfied or not yet satisfied).

- **Unit literal Queue:** In addition to the above, we maintain a list of unsatisfied clauses that contain a unit literal. This allows easy look up of unit literals.

[2]That this is a promising approach for SAT is a relatively recent discovery, made independently by Selman (Selman, Levesque, & Mitchell 1992) and Gu (Gu 1992).

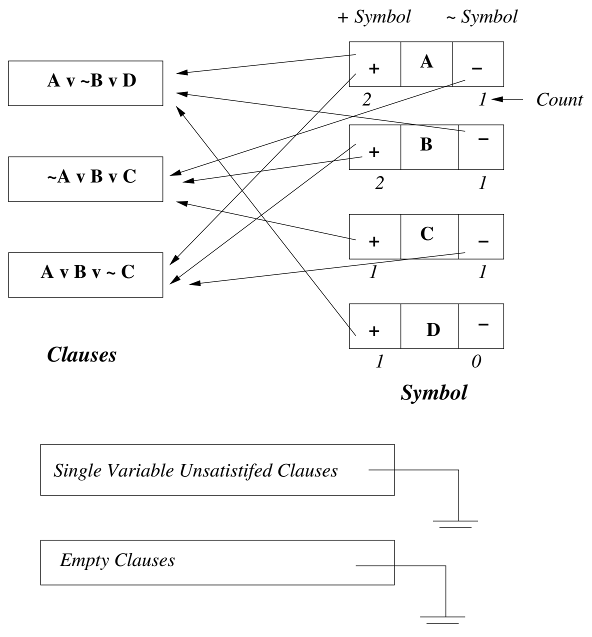

Figure 1: DPLL Data Structures.

The following operations need to be performed during every iteration:

1. For all clauses containing the literal, the clauses are marked satisfied and all clauses that contain the negation of the literal mark the symbol as deleted. If the clause becomes a unit clause or an empty clause after the deletion of the symbol the clause is pushed in the appropriate queue. Further, for each clause that was satisfied, the unsatisfied clause count maintained by its other symbols needs to be updated.
2. This allows easy look up of unit literals. Pure literals are easily looked up using the count structure maintained for each symbol.
3. On backtracking all the above operations need to be reversed which is done by a *clean up* routine.

Heuristics for DPLL Although DPLL is a complete search algorithm, its performance can be improved using good ordering on symbols for partial assignment. We shall now discuss a few heuristics that can be used with DPLL:

1. Random Heuristic: This heuristic selects a symbol at random to assign value. Thus, symbols are assigned in a random order while the completeness of DPLL is retained.
2. Greedy Heuristic: In this heuristics we choose the literal that satisfies most number of unsatisfied clauses with the goal of finding the solution at the earliest. This heuristic is based on the greedy approach but retains the completeness property of the DPLL algorithm.
3. Random Heuristic with re-starts: Although the random heuristic does better that a static ordering on the symbols it suffers when the ordering on the literals is a relatively undesired one, thus exploring a part of the such tree which is unlikely to yield a solution. In order to overcome this drawback, we added the re-start property which restarts the search for the solution from the beginning if the solution is not found within a some number of steps. However, the price for this improvement is loss of DPLL's completeness property, or in other words, DPLL with restarts may return without a solution even given sentence has a solution.

Preliminary evaluations suggest that the Greedy heuristics outperforms the rest. Therefore, in the interest of time and space we shall use the performance of Greedy DPLL in our analysis.

3 WALKSAT

WALKSAT belongs to the class of algorithms which perform a local search in the space of complete assignments. A local search procedure moves in a search space where each point is a truth assignment to the given literals. A solution is an assignment in which each clause of the CNF formula evaluates to true (McAllester, Kautz, & Selman 1997). Local search has been shown to be surprisingly good at finding completely satisfying assignments for CNF problems (Selman, Levesque, & Mitchell 1992; Gu 1992). WALKSAT starts with a randomly generated truth assignment. On every iteration, it picks an unsatisfied clause and picks an unsatisfied literal in this clause to flip. It chooses randomly between two ways to pick which symbol to flip: (1) a "min-conflicts" step that minimizes the number of unsatisifed clauses in the new state, and (2) "a random walk" step that picks the literal randomly (Russell & Norvig 2003). The first corresponds to the evaluation of an objective function. The objective function that WALKSAT attempts to minimize is the total number of unsatisfied clauses. The second corresponds to a random, possibly non-optimal move that enables WALKSAT to escape from local minima. This randomized strategy helps WALKSAT avoid the most common pitfall that often plagues combinatorial problems. Algorithm 1 is the pseudocode representation of WALKSAT.

WALKSAT is an incomplete search algorithm, in the sense that it may not always terminate with a satisfying assignment. However, WALKSAT is usually faster than its systematic counterparts since, at any given time, it involves flipping a single variable from an unsatisfied clause, a task of much smaller complexity than the sub-tasks in systematic search. The performance of a stochastic local search procedure critically depends upon the setting of the parameters that determine its likelihood of escaping from local minima by making non-optimal moves. This characteristic of a search strategy that causes it to make non-optimal moves – moves which increase or fail to decrease the objective function even when better moves are available – is called "noise". Adding noise perturbs the search space and enables the search to "jump" out of local minima. In WALKSAT, this "noise" parameter is the probability p. The optimal setting for this parameter depends both upon the problem instances and on the specfic details of the search procedure, which may be influenced by other parameters and in general,

```
/* clauses = A set of clauses in propositional logic */
/* p = probability of random-walk */
/* MAX_FLIPS = maximum number of flips per try */
/* MAX_RESTARTS = maximum number of restarts
   before terminating */
/* model = random assignment of true/false to literals
   in clauses */
/* result = true if WALKSAT finds a solution, false
   otherwise */
input  : clauses,p,MAX_FLIPS,MAX_RESTARTS,model
output: bool result

WALKSAT(clauses,p,MAX_FLIPS,MAX_RESTARTS,model)
for $i \leftarrow 1$ to MAX_RESTARTS do
    for $j \leftarrow 1$ to MAX_FLIPS do
        if model satisfies clauses then
            return true ;
        end
        falseClause = A randomly selected clause
        from clauses that is false in model ;
        rNum = Generate a random number in $[0, 1]$;
        if rNum $\leq$ clauses then
            Flip the value in model of a randomly
            selected literal in falseClause ;
        else
            Flip whichever symbol in falseClause
            maximizes the number of satisfied clauses
            ;
        end
    end
end
```

Algorithm 1: The WALKSAT algorithm

requires a non-trivial amount of tuning to achieve an agreeable setting. However, the general performance of WALKSAT can be improved by using heuristics. A crucial step is selecting which literal in the randomly selected unsatisfied clause. For our experiments, we applied some heuristics in making this selection, which are detailed below. The first heuristic is our own and the subsequent ones have been detailed in (McAllester, Kautz, & Selman 1997).

Heuristics for WALKSAT

- $GREEDY$: Strictly speaking, this heuristic applies to the selection of an unsatisfied clauses from among the unsatisfied ones. In this, instead of randomly selecting the unsatisfied clause, the selection is performed in a greedy manner. As soon as it is determined that the given sentence is unsatisfied, the first clause that renders the sentence unsatisfied is greedily chosen. This simple strategy was generally found to be the fastest algorithm to obtain a satisfiable solution , as our experimental results in section demonstrate. This is a natural consequence of the greedy choice as described.
- $NOVELTY$: In this heuristic, the literal whose flip causes the greatest reduction in number of unsatified clauses and the second best such literal (if it exists) in the selected clause are considered. If the best literal is not the most recently flipped literal, it is selected. Otherwise, with a probability p, the second best literal is selected, and with probability $1 - p$, the first literal is selected.
- $R\_NOVELTY$: This heuristic is similar to NOVELTY, except in the case where the best literal is the most recently flipped one. In this case, let n be the difference in the objective function between the best and second-best literal. There are then, four cases:
  1. When $p < 0.5$ and $n > 1$, pick the best.
  2. When $p < 0.5$ and $n = 1$, then with probability $2 * p$, pick the second-best, otherwise pick the best.
  3. When $p \geq 0.5$ and $n = 1$, pick the second best.
  4. When $p \geq 0.5$ and $n > 1$, then with probability $2 * (p - 0.5)$ pick the second-best, otherwise pick the best.

  The intuition for $NOVELTY$ and $R\_NOVELTY$ is to avoid repeatedly flipping between the same literal back and forth.
- $SELFTUNE$: This heuristic is related to obtaining the optimal setting for the parameter p. Let us define the *normalized noise level* of a search procedure on a given problem instance as the *mean value of the objective function* during a run on that instance. This quantity has been observed to be approximately constant across WALKSAT implementations employing heuristics such as those described above. In (McAllester, Kautz, & Selman 1997), the authors describe a preliminary principle for setting noise parameters based on statistical properties of search. More specifically, many short runs of the search procedure are made and the mean-to-variance ratio of unsatisfied clauses is recorded and subsequently, averaged over the runs. We modifed this heuristic to arrive at a self-tuning version of WALKSAT. The idea is to start a run at a arbitrary noise level. Then the averaged mean-to-variance ratio of unsatisfied clauses is obtained every few runs. If this ratio is too low, it means we are reaching states with low numbers of unsatisfied clauses. But, this can also mean that the variance is also very small, and in fact, so small, that the algorithm seldom reaches a state with zero unsatisfied clauses. When this occurs, the algorithm is stuck in a deep minima. In this situation, the noise levels are increased. If the ratio is too high, the variance is large but the average number of unsatisfied clauses is even larger. We would need to "cool down" the system and decreased the perturbations induced by the current p setting. Therefore, the noise levels are decreased. In this manner, by dynamically adjusting noise level, one can obtain optimal performance.

4 Genetic Algorithm with Local Search

Genetic Algorithms have been to used to obtain *near* optimal solutions for many hard problems (Hao, Lardeux, & Saubion ; Lardeux, Saubion, & Hao 2004). We use Genetic Algorithms as a bootstrap for local search algorithms such as WALKSAT, Tabu Search. While Genetic Algorithms aid in quick search for near satisfiable assignment, local search heuristics then use these solution to obtain the optimal solution if it exists. Like most randomized algorithms, this

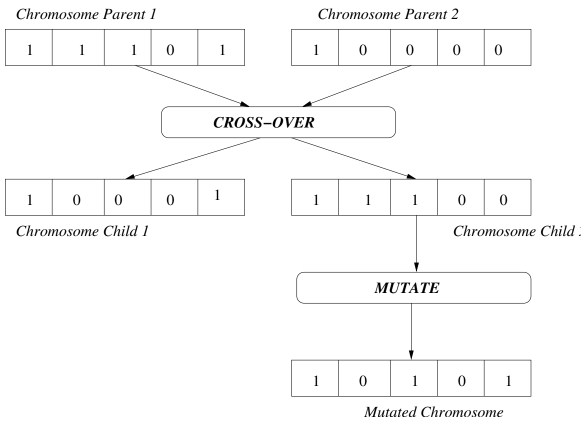

Figure 2: Genetic Algorithms, representation and operations.

heuristic is not guaranteed to find the solution even if it exists; therefore Genetic Algorithm with local search is not a complete search algorithm.

Characterizing Genetic Algorithms Representation: The most obvious way to represent a solution is stream of bits which represent individual assignment to the symbols of the satisfiability problem. Therefore, the i^{th} bit turned on, represents that the i^{th} symbol is assigned true while a turned off bit represents that the i^{th} symbol is assigned false. Each string in the population thus represents a *chromosome*.

Fitness Function: The fitness of a chromosome can be defined as number of clauses in given sentence that the chromosome satisfies. The fitness function induces the selection procedure on the chromosomes.

Cross over Function: The crossover function represents the generation of new chromosomes from existing chromosomes. Once the parent chromosomes are selected based on the fitness function the crossover operation is performed to generate two children. For each *gene* in the chromosome (i.e., each symbol in the assignment) we randomly exchange each gene between the parents to create new children. Figure 2 demonstrates the crossover operation.

Mutation Function: In order to rescue Genetic Algorithm from local minimas we add the Mutation function which randomly flips a small number of *genes* from randomly selected chromosomes from the population.

The Algorithm The initial population for a given satisfiability problem is randomly generated. Each successive generation is generated by the crossover and mutation operations on the current generation. The newly created generation is evaluated for fitness and the fittest chromosome is used as a bootstrap to the local search algorithms presented in the next section. If at any point in the execution a solution is found the search terminates and the solution is returned. Otherwise, Genetic Algorithm continues generating further generations and evaluating the fitness of individual chromosomes. If no solution is found within the $MAX\_GENERATIONS$ then the search terminates declaring that the given sentence is unsatisfiable.

Local Search Heuristics We tried two different local search heuristics to complement Genetic Algorithm.

Tabu Search: We use tabu search to flip the symbol that satisfies the most number of unsatisfied clauses. Whenever, a symbol is flipped it is pushed into a fixed size *tabu-list* in order to prevent flipping it back again. Using Tabu Search prevents the local search algorithm from evaluating the same solution over and over again by flipping the same symbols over and over again. The intuition behind flipping the symbol that satisfies the most number of unsatisfied clauses is maximize the gain and reach the optimal solution at the earliest.

WALKSAT: We bootstrap the WALKSAT local search heuristic using the fittest chromosome of the genetic algorithm generation. Using a good solution to bootstrap WALKSAT provides a better initial solution to WALKSAT reducing the number of flips required to find the optimal solution. We restrict WALKSAT to use initial assignment generated by Genetic Algorithm by setting the number of restarts to 1.

5 Evaluation

Characterisation of Randomly Generated 3 SAT problems

In this section we present some important characteristics for satisfiability problems. In particular, we answer the following questions:

- For a given clauses-to-literals ratio what fraction of problems have a solution?
- Does clauses-to-literals ratio have any bearing on the hardness of a problem?

As the number of clauses increases the solution space becomes more and more constrained. The first question attempts to answer if the clauses-to-literals ratio is a good indicator of the complexity of the problem. In other words, if there exists a point such that the probability of finding a problem with a valid solution is the same as probability of finding a problem without a valid solution then most problems near this point are such that they have very few (if any) solutions thus making them hard problems. The second question tries to justify the argument that problems which are hard in theory are indeed hard in practice.

Figure 3 answers the first question about the fraction of problems that are satisfiable for a given clauses-to-literals ratio. In particular, we observe that for a low clauses-to-literals ratio the SAT problem is not constrained enough so that any randomly generated problem is satisfiable. On the other hand at high clauses-to-literals ratio the any randomly generated problem is highly constrained and hence mostly unsatisfiable. Of particular interest is the region between 4.0 and 4.5 where probability that a randomly generated problem is satisfiable changes drastically. This region represents the region of high *entropy* where any randomly generated

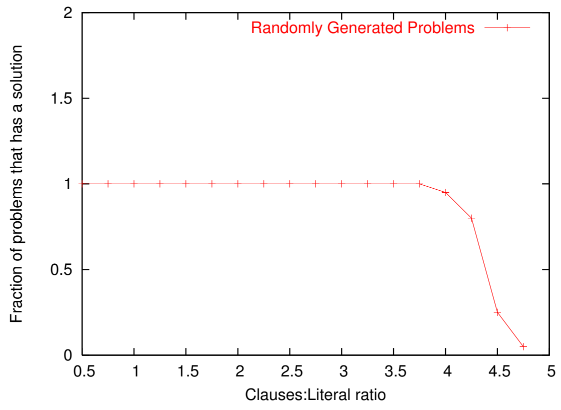

Figure 3: Fraction of satisfiable problems for a given clauses:literals ratio.

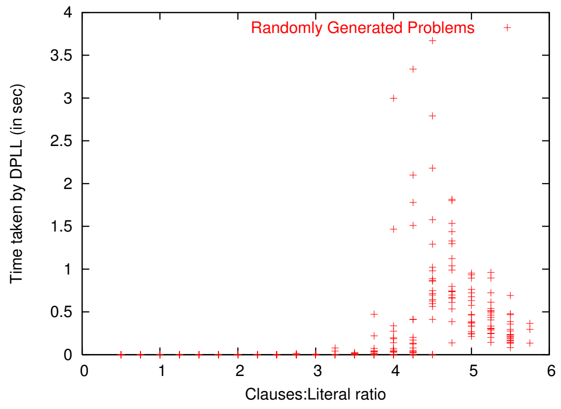

Figure 4: Distribution of run-times of DPLL for randomly generated problems.

problem is equally likely to be either satisfiable or unsatisfiable. This suggests that the satisfiable problems in this region are likely to have very few solutions and hence would require clever heuristics rather than brute force search.

Figure 4 justifies the above conclusion by showing that a practical complete search algorithm such as DPLL stuggles in the region of high entropy i.e., around clauses-to-literals ratio of 4.3. Towards the left most of the problems are satisfiable and hence DPLL can find a satisfiable solution quickly. On the other hand, towards the right most of the problems are unsatisfiable and hence DPLL finds a contradiction quickly, thus pruning a significant fraction of the search space. It is around the region of high *entropy* that DPLL needs to search a large portion of the search space resulting into high running times. Thus, the region around clauses-to-literals ratio of 4.3 represents *hard* problems.

Performance Characteristics

As a preliminary experiment, we compared WALKSAT,R_WALKSAT, WALKSAT_SELFTUNE,DPLL,GA+TABU and WALKSAT+GA algorithms on various $3-CNF$ hard problems. The results are given in Table 1. For each strategy, we give the average time in seconds it took to find a satisfying assignment, the average number of flips it required, and R, the average number of restarts needed before finding a solution. For each strategy, we used atleast 30 random restarts on each problem instance. If we needed more restarts, then we performed the restarts for a maximum of 100 times. A "*" in the table indicates that no solution was found after running for more than 4 hours or using more than 100 restarts. The results are averaged over the various parameter settings. In our case, we ran the experiments for $MAX\_FLIPS = 1000, 5000, 10000$, $MAX\_RETRIES = 5, 10, 25, 1000$, $p = 0.2, 0.5, 0.7$ and TABU list lengths ranging from 10 to 100.

The results suggest that DPLL performs poorly at finding solutions to large instances of unsatisfiability problems within a reasonable period of time. WALKSAT, inspite of the incomplete nature of its search, performs better. The greedy WALKSAT performs competitively with WALKSAT which uses the NOVELTY heuristic. We implemented WALKSAT-SELFTUNE as a means of verifying the hypothesis given in (McAllester, Kautz, & Selman 1997) that a dynamically parameter adjusting version would be equivalent to a hand-crafted optimal version. While the results do not indicate the same, we believe that the internal parameters for changing the "noise" factor (p) could be tweaked to obtain the same. We defer this to a future version of the work.

Completeness vs Correctness

Although the execution time for WALKSAT (for large problems) is much smaller than complete search algorithms such as DPLL, such heuristics sacrifice the important property of completeness. In this section we study the ability of such incomplete search heuristics to find the solution when it exists. We define, *correctness* of a heuristic as the fraction of times it returns the right answer.

Clearly, correctness of a heuristic depends on the hardness of the problems it attempts to solve. It might be easy to for the most naive heuristic that randomly explores the solution space to almost always find a solution for a sentence with many solutions. On the other hand, for a problem with just one valid solution, that best of the heuristics may stuggle to find it. Therefore, we study the correctness of a heuristic in the light of the hardness of the problem. While it may not be easy to quantify the hardness of a problem in absolute terms, previous studies (Selman, Levesque, & Mitchell 1992) show that for randomly generated problems, a clauses-to-literals ratio of 4.3 tends to be hard. Problems with clauses-to-literals ratio less than 4.3 tend to have multiple solutions and hence tend to be easy. On the other hand clauses-to-literals ratio of greater than 4.3 tend to be unsolvable and hence may not be of interest.

Figure 5 shows the correctness of WALKSAT and Genetic Algorithm based heuristics for various values of clauses-to-literals ratio. Each point is generated using 20 sets of randomly generated problems for a given clauses-to-literals ratio. Using DPLL (a complete search algorithm) we solve the problem completely to determine is a valid solution

Formula		WGY			WNY			WGA			WST			DPLL	GTU
Vars	Clauses	T	F	R	T	F	R	T	F	R	T	F	R	T	T
10	43	0.126	8	1	0.399	31	1	0.319	4	1	0	5	1	0.339	4
50	215	9	89	1	10	128	1	26	3	1	3	24	1	882	16
100	413	74	607	1	37	221	1	69	5	1	367	1250	1	21000	473
500	2150	4934	7893	2	5888	7802	2	17000	100	9	49904	32423	4	*	*
1000	4300	416598	328927	30.5	204877	138150	13.6	*	500000	100	1164388	309277	31	*	*

Table 1: Performance of various algorithms on hard random 3CNF instances. WGY = WALKSAT_GREEDY, WNY = WALKSAT_NOVELTY , WGA = WALKSAT_GA, WST = WALKSAT_SELFTUNE, DPLL = Davis Putnam, GTU = GA+TABU. T = Running time (in milli-s), F = Number of flips, R = Number of retries.

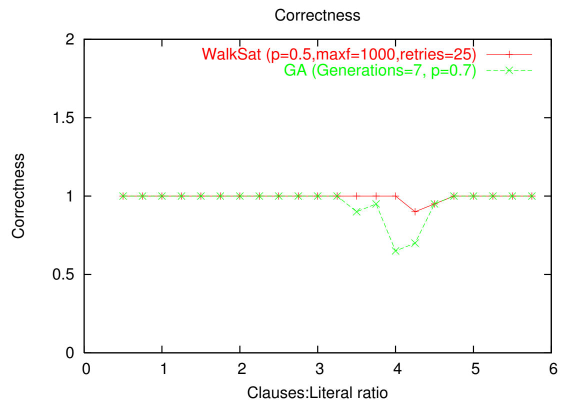

Figure 5: Correctness of WALKSAT and Genetic Algorithm based heuristics.

exists. We compare the result of WALKSAT/Genetic Algorithm to that of DPLL. The result is declared correct if the heuristic agrees with DPLL on the solvability of the problem. Note that the heuristics need not return the same satisfiable assignment as DPLL. Moreover, the case that DPLL is not able to find a solution while heuristics find one cannot arise because DPLL considers all possible candidate solutions. The figure illustrates that while incomplete search heuristics are mostly correct for *easy* problems they may return incorrect results for *hard* problems.

Table 2 illustrates that the correctness of WALKSAT is a strong function of its paramter values. Infact, the correctness of WALKSAT is greatly improved by increasing the number of flips and the flip probabilities. Clearly this represents the trade-off between time and correctness. This can be attributed to the fact that increasing the number of flips increases the number of local solutions explored by WALKSAT while increasing the flip probability increases the probability of deterministic choice thus preventing WALKSAT to "walk" away from the exact solution.

Effect of Problem Size

In this section, we show that increase in problem size has a tremendous impact on various SAT solving algorithms. In particular this impact may be exhibited by various algorithms in two different ways. For complete search algorithms such as DPLL, increase in problem size results

Flips **Probability**	**1000**	**5000**	**9000**
0.2	0.7	1.0	1.0
0.5	0.9	1.0	1.0
0.7	1.0	1.0	1.0

Table 2: Effect of parameters on the correctness of WALKSAT at clauses to literals ratio of 4.25

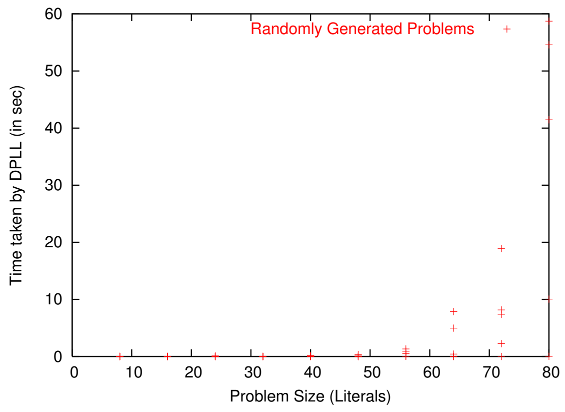

Figure 6: Effect of increasing problem size on the runtime of DPLL

in non-linear increase in running times. Heuristics such as WALKSAT which execute in bounded number of iterations may show this non-linearity by decrease in correctness.

Figures 6 and 7 show the effect of increased problem size on the run times of DPLL and WALKSAT at clauses-to-literals ratio of 4.3. While WALKSAT shows more of a linear worst case increase in runtime, DPLL shows shows a non-linear increase. The linear increase in WALKSAT is owing to the linear increase in problem size while that of DPLL is dues to an exponential increase in the size of search space. On the other hand, for a fixed parameter value WALKSAT begins to show decrease in correctness for larger problems, as shown in Figure 8.

Comparison of Heuristics

We shall now address the question on *what heuristic should be used to solve a given set of satisfiability problems?* Ofcourse, our arguments are subject to our implementation of

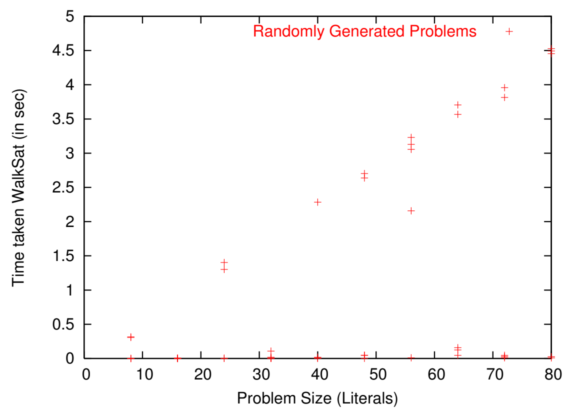

Figure 7: Effect of increasing problem size on the runtime of WALKSAT

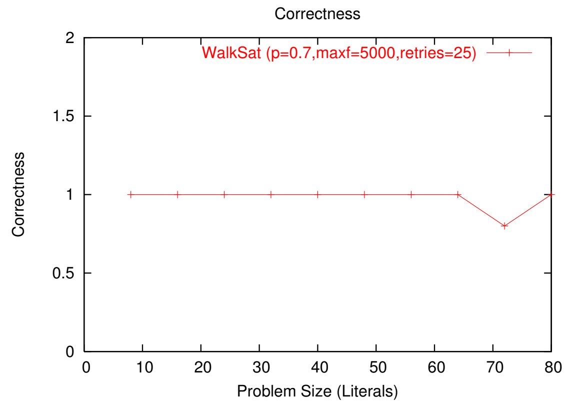

Figure 8: Effect of increasing problem size on the correctness of WALKSAT

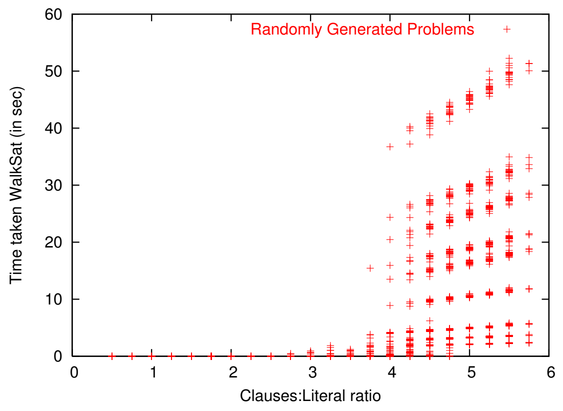

Figure 9: Distribution of run-times for WALKSAT on randomly generated 60 symbol problems.

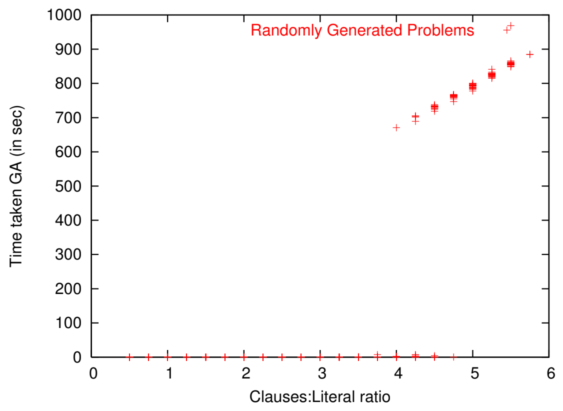

Figure 10: Distribution of run-times for Genetic Algorithm on randomly generated 60 symbol problems.

various heuristics. While we do not argue that our implementations for various heuristics is the most efficient implementation, we took great care to optimize various data structures used by the algorithms. Furthermore, all the heuristics were programmed in C, and compiled using gcc 3.3.3. For our experiments, we used Linux 2.6.7 running on a Dell OptiPlex GX400 with a 1.7 GHz Pentium 4 processor, one GB of memory. While running the experiments, no other applications were active on the system. We determined that the memory requirements never exceeded the amount of physical memory available on the system.

Figures 4, 9, and 10 show the distribution of runtimes on randomly generated 60 symbol problems for DPLL, WALKSAT, and Genetic Algorithm respectively. A striking difference to note is that while the runtimes for DPLL reduces after the hard problem region (i.e., after clauses-to-literals ratio of 4.3) the same is not true for partial search heuristics such as WALKSAT and Gentic Algorithms. The reason for this difference is owing to the number of unsatisfiable problems. While DPLL finds a contradition and terminates declaring no solution exists, WALKSAT and Genetic Algorithms continue running for maximum number of tries before they arrive at the same conclusion. This shows that while partial search heuristics may be good to solve problems which are guaranteed to have a solution, they are not good for highly constrained problems. Another interesting point to notice is that even though the number of retries remain the same as the clauses-to-literals ratio increases the run time also increases towards the right of the hard problem region. This increase is owing to additional work that needs to be done to evaluate more clauses for every symbol assignment.

Figure 9 exhibits a range of different clusters of run-times for various values of maximum number of flips.

In summary, the choice of algorithm to use is highly dependent on the clauses-to-literals ratio. In particular for low clauses-to-literals ratio, WALKSAT works very well as it finds a solution quickly without much error. For highly constrained problems DPLL works well since it generates a contradiction quickly. However, at the high *entropy* region , the choice of algorithm is based on the trade-off between runtime and the correctness of the output.

6 Conclusions and Future Work

In this report, we have presented an overview and performance analysis of two popular approaches to propositional satisfiability problems – DPLL and WALKSAT. Both have their pro and cons, naturally arising out of the trade off between a complete and incomplete nature of searches they perform. Our experimental results confirm with some of the conjectures posed in literature. In addition, we have also proposed new heuristics whose performance was found to be competitive with existing heuristics.

Given the dynamic ranges of parameter settings which were found to be optimal for the aforementioned two categories of problems, statistical measures and techniques which allow the algorithm to dynamically adapt the parameter settings are particularly appealing (McAllester, Kautz, & Selman 1997). Such algorithms would also enable a much more uniform platform for comparing the performances of these algorithms.

During the course of our experiments, we have observed that while a heuristic may be motivated by shortcomings present in existing approaches, incorporating the heuristic may incur a performance overhead. In some cases, this overhead has not been adequately compensated by improvement in performance. In particular, we found that the simplest heuristic to apply – the greedy heuristic of WALKSAT performed better than other heuristics. We believe that a 'lean-and-mean' approach for designing heuristics can help improve the performances of search algorithms.

We explored the possibility of bootstrapping the search by using techniques which generate a good initial assignment. In our case, we used a Genetic Algorithm to find the purported good 'seed' with which to commence the search, but there was no observed improvement, owing to the fact that we did not incorporate any optimized heuristics for Genetic Algorithm itself. However, we believe that if the bootstrapping is done intelligently, it can boost the speed of existing algorithms greatly. This could be a particularly fruitful direction for future research in search techniques for propositional satisfiability.

The experiments discussed in this report were performed on randomly generated $3 - CNF$ problem instances. It would be interesting to observe the performance of various heuristics for other kinds of CNF problems. Another potential test set for future work employing the heuristics mentioned here could be planning problems which have been compiled into SAT problems. In our experiments, we have not compared the performance with existing implementations since we would have wanted to discuss the results not just by comparing running times but also by a qualitative explanation of the tweaks implemented therein. We believe that would lead to a better understanding of the method, however, time and space constraints have necessitated the deferment of such a study to future.

A Division of Work

- DPLL and various heuristics, bootstrapping algorithm for GA - `Gaurav`
- WALKSAT and various heuristics - `Ravi`
- SAT Generators, Scripts to run programs in batch mode - `Gaurav, Ravi`
- Literature Survey - `Gaurav,Ravi`
- Project Report - `Gaurav,Ravi`

References

Cook, S., and Mitchell, D. 1997. Finding hard instances of the satisfiability problem: A survey. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 5.

Davis, M., and Putnam, H. 1960. A computing procedure for quantification theory. *J. Assoc. Comput. Mach.* 7:201–215.

Gu, J. 1992. Efficient local search for very large-scale satisfiability problems. In *Sigart Bulletin*, volume 3, 8–12.

Hao, J.-K.; Lardeux, F.; and Saubion, F. A hybrid genetic algorithm for the satisfiability problem. In *Universite d' Angers*.

Lardeux, F.; Saubion, F.; and Hao, J. 2004. The gasat solver. In *SAT 2004 Competition: Solver Descrptions*.

McAllester, D.; Kautz, H.; and Selman, B. 1997. Evidence for invariants in local search. In *Proceedings of AAAI-97*.

Russell, S., and Norvig, P. 2003. *Artificial Intelligence - A Mordern Approach*. Prentice Hall.

Selman, B.; Kautz, H.; and Cohen, B. 1996. Local search strategies for satisfiability testing. *Second DIMACS Challenge on Cliques, Coloring and Satisfiability*.

Selman, B.; Kautz, H.; and Cohen, B. 1997. Ten challenges redux: Recent progress in propositional reasoning and search. In *Proceedings of IJCAI-97*.

Selman, B.; Levesque, H.; and Mitchell, D. 1992. A new method for solving hard satisfiability problems. In *Proceedings of AAAI-92*.