An Analysis and Comparison of Satisfiability Solving Techniques

Ankur Jain, Harsha V. Madhyastha, Craig M. Prince
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195
{ankur, harsha, cmprince}@cs.washington.edu

Abstract

In this paper, we present an analysis of different algorithms that have been used for SAT solving. We implement and analyze the working of two standard algorithms used for solving SAT instances - WalkSAT and DPLL. Identifying key optimizations in both these algorithms, we demonstrate with the help of extensive experiments the improvement in performance brought about by these optimizations. Based on the insight gained from our explorations with WalkSAT and DPLL, we implemented and analyzed a new approach to SAT solving, which we call HybridSAT. HybridSAT draws on the strengths of both WalkSAT and DPLL, resulting in a remarkable improvement in performance over DPLL, while still retaining completeness.

Introduction

Satisfiability (SAT) is one of the canonical NP-complete problems in computer science. Given a boolean formula with conjunction, disjunction and negation, the goal is to find an assignment to the boolean variables that makes the formula true. This has a wide range of applications in all fields of computer science and especially artificial intelligence.

The common representation of a boolean formula is conjunctive normal form (CNF). In this form, formulae are given as several clauses containing only disjunctions (boolean OR), which are all linked together by conjunction (boolean AND). It has been shown that any boolean formula can be expressed in CNF. In addition, clauses need no more than 3 symbols each to be able to express any boolean formula. Because of this, for the remainder of this paper we will focus on solving 3-CNF formula.

SAT as Search

Solving a boolean formulae is essentially a search problem. As such, we can apply many of the techniques and algorithms used for general search to the problem of satisfiability. Two such classes of search algorithms commonly used for satisfiability are backtracking search and local search.

- **Backtracking Search:** Backtracking searches usually are designed to do a complete exploration of the search space to uncover the solution to a problem. They do this by "remembering" where they have searched before (either explicitly or because they employ a systematic approach). Backtracking searchs are those such as depth-first-search, breadth-first-search, etc.
- **Local Search:** Local search involves using local or immediate information to inform the direction of the search. These approaches are usually greedy and can often quickly lead to the correct result. Hill climbing is an example of a local search algorithm.

In our paper we explore both a backtracking search (DPLL) and a local search (WalkSAT) so that we can compare the two methods. In addition, we have developed a hybrid approach of the two algorithms – taking the "best" aspects of each – in an attempt to create an algorithm that is better than either.

Methodology

The goal of this paper is to gain understanding into the benefits and tradeoffs of various satisfiability algorithms. Not only did we want to gain an intuitive understanding, but also a quantitative understanding of the tradeoffs involved with each. We have chosen to experiment with the DPLL backtracking algorithm and the WalkSAT local search algorithm.

We knew *a priori* that DPLL was complete, but could also be slow. We also knew that WalkSAT would work very quickly most of the time, but was not guaranteed to find a solution if one existed. What we didn't know was exactly how the two compared and what the effect of various optimizations would be on the algorithms. For this we designed several benchmark tests in order to quantify these values.

We began by simply implementing the two algorithms without adding any enhancements. We used these as a baseline for evaluation. For the case of WalkSAT we were then able to adjust the various parameters of the algorithm in order to determine the optimal values for them. After developing the basic implementation we added an enhancement to each algorithm and again ran our tests to determine how much improvement there was. Our main metric for performance was running-time; however, in the case of WalkSAT a second metric was the "accuracy" of the algorithm; namely, given satisfiable clauses, how likely was the algorithm to find the assignment.

Once our evaluation of the individual algorithms was complete we implemented a hybrid algorithm that we felt combined the benefits of both WalkSAT and DPLL. We then evaluated this hybrid – comparing it to the other algorithms.

Outline

We begin with an overview of related work in the field of satisfiability. The next section describes our implementation and experimental exploration of the WalkSAT algorithm. The next section explains our implementation and results for the DPLL algorithm. After this we explore our hybrid approach combining both DPLL and WalkSAT, again giving experimental results of its performance. Finally, we conclude with a summary of our results.

Related Work

A lot of research has been conducted in the field of satisfiability. In the 1960s work was done by Davis, Logemann, and Loveland to develop what we know today as the DPLL algorithm (Davis, Logemann, & Loveland 1962). Since this time, there has been extensive work on attempting to improve upon this algorithm by providing heuristics and optimizations for improving the performance of this backtracking search algorithm. Some of these include using heuristics for guiding the search (De 2002) as well as more advanced techniques like conflict resolution and restarting (Moskewicz *et al.* 2001).

On the other hand there has been a great deal of work in improving the local search methods of finding satisfiability. WalkSAT is perhaps the most famous local search method (Selman, Kautz, & Cohen 1993); however, before WalkSAT there was a related algorithm called GSAT (Selman, Levesque, & Mitchell 1992). This differs from WalkSAT in that it does not employ randomness in its variable flipping, but instead takes a pure greedy approach, running multiple times with random starting configurations each time.

Much of the research around satisfiability is sparked by the versatility of SAT solvers in solving real problems. SAT can be used to solve planning (Kautz & Selman 1992) in addition, SAT has been used in circuit verification (Goldberg, Prasad, & Brayton 2000). These real-world applications only further emphasize the need for good SAT algorithms.

WalkSAT

Introduction to WalkSAT

The WalkSAT algorithm is considered a local search algorithm (more specifically the search is randomized hill-climbing) that does not keep any search history while looking for a satisfying assignment to variables. As a result, this algorithm cannot know when it has completely explored the search space – meaning that WalkSAT is not complete. We implemented the basic WalkSAT algorithm as described in (Russell & Norvig 2003). The pseudocode for this algorithm is given as follows:

```
$WALKSAT(clauses, p, maxFlips)$
{
    inputs: $clauses$, a set of clauses in propositional logic
        $p$, the probability of choosing a random walk step
        $maxFlips$, number of flips allowed before giving up
    $model$ = a random assignment of true or false to the
        symbols in $clauses$
    for $i$=1 to $maxFlips$ do
        if $model$ satisfies $clauses$ then return $model$
        $clause$ = a randomly selected clause from $clauses$
        that is false
        with probability $p$ flip a random symbol from the
        clause
        else flip the symbol in $clause$ to maximize the num.
        of satisfied clauses
    return failure
}
```

There are two parameters that can be adjusted which affect the performance of WalkSAT. The first is the maxFlips parameters. This controls how many steps the algorithm runs for. If the algorithm hasn't found a solution after running for this number of iterations, then it will give up. The second parameter, p, is the probability with which we choose to flip some symbol at random from an unsatisfied clause as opposed to using the heuristic of maximizing the number of satisfied clauses.

This algorithm has some very nice properties. First of all, because of the randomness, this mitigates the chance that the algorithm will get stuck in a local maximum. In addition, because the algorithm runs for a finite number of flips we know that the algorithm will always terminate. This means that we can bound the running time. Also, because the search heuristic is good, it means that the algorithm usually converges quickly to an answer.

While the WalkSAT algorithm has some nice properties, there are some important limitations of the algorithm. Most importantly is the fact that the algorithm is not complete. This means that when the algorithm does return an assignment we can be certain that the formula is satisfiable; however, if the algorithm fails we don't know whether this is a result of the formula being unsatisfiable or just the fact that we didn't run for long enough.

WalkSAT Experimental Results

We wanted to quantify the performance of WalkSAT and so we conducted a series of tests to benchmark the effectiveness and speed of this algorithm. The first set of experiments that we conducted were designed to determine the best parameters to use for both the maximum number of flips and the random flip probability.

Figure 1 shows the accuracy (how many of the satisfiable formulae it was able to solve) for various different maxFlips settings. We can see that until we drop below the 2000 mark we correctly identify nearly all the formulae. After this the accuracy begins to drop off linearly from 2000 to 50.

Similarly, Figure 2 illustrates an interesting phenomenon. Namely we can see that the time it takes to get a valid solution with WalkSAT is usually small and so if we take longer

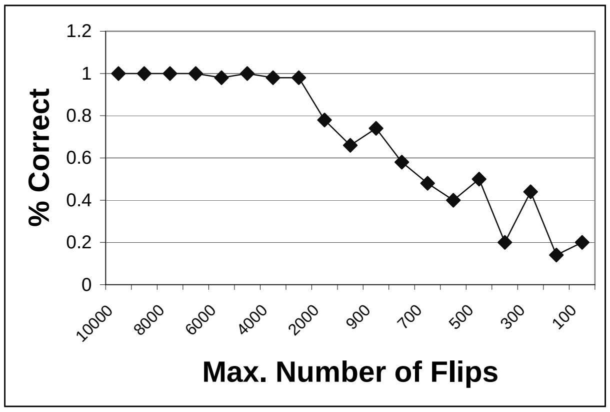

Figure 1: Percentage of the satisfiable 3-CNF formulae that WalkSAT produced a satisfying assignment for over varying maximum number of flips. All had a random flip probability of 0.5, and a clause-variable ratio of 4.6

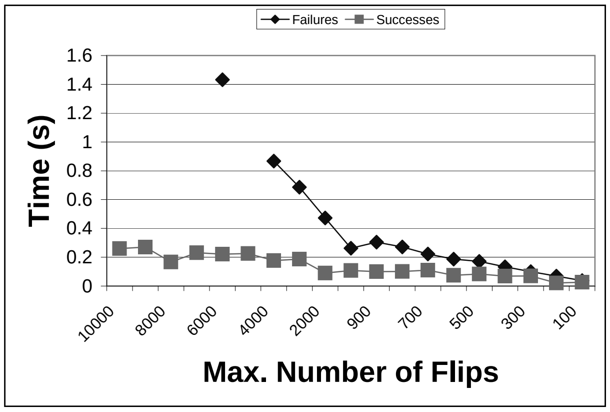

Figure 2: Average time taken to solve 10 satisfiable 3-CNF formula (with 5 runs for each) when varying max. number of flips. The two lines represent the time taken when a solution is eventually found and when a solution is not found

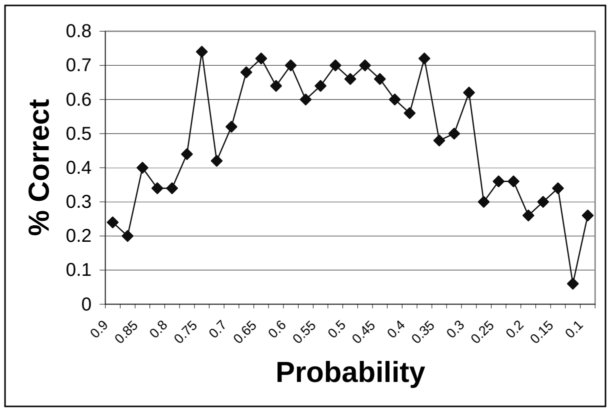

Figure 3: Percentage of the satisfiable 3-CNF formulae that WalkSAT produced a satisfying assignment for over varying the probability of a random flip. All had a max. number of flips of 800, and a clause-variable ratio of 4.6

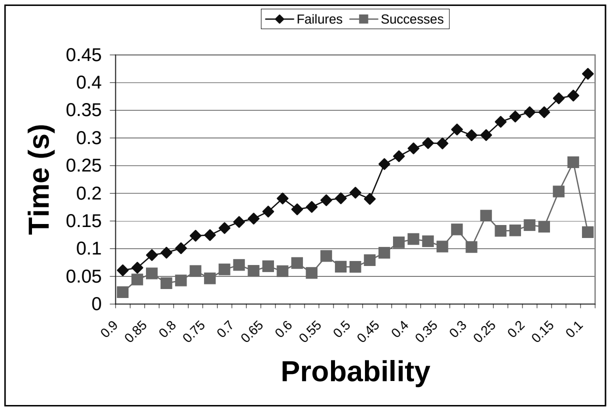

Figure 4: Average time taken to solve 10 satisfiable 3-CNF formula (with 5 runs for each) when varying the probability of a random flip. The two lines represent the time taken when a solution is eventually found and when a solution is not found

then we increase our confidence that a formula is not satisfiable. Note also that as we increase the maximum number of flips the gap between the lines grows – meaning that we can gain better confidence by increasing the number of flips.

Figures 3 and 4 are the same as the previous graph except plotted over varying values of probability for making a random flip. Here the story is somewhat different. There seems to be a range between 40 and 60 percent where WalkSAT remains effective. Also like above, in the time chart we again see the gap between the failure and success cases of WalkSAT. Notice that the overall time increases as we decrease the probability. This is an artifact of the algorithm. Since it is much faster to choose a symbol at random than to choose the best symbol, when we take the random choice we run faster.

From these two experiments we can concluded that in general a probability of 0.5 works well and a high number of flips (over 2000) will give high accuracy and good performance. With this knowledge we then proceeded to directly test how quickly WalkSAT could solve satisfiable problems. Given a set of 160 satisfiable formulae we ran WalkSAT and timed how long it took (with a sufficiently high maxFlips to ensure that we found the solution). Figure 5 summarizes the results. As expected, between a ratio of 3.75 and 4.75 is where most of the difficult problems lie. We will use this same set of formulae as a benchmark to compare the various different algorithms presented later.

At this juncture, we must point out the mechanism we employed to generate satisfiable SAT instances. An obvious way of doing this is to first pick a random assignment of the variables and while generating clauses ensure that each clause is satisfied under this assignment. However, as outlined in (Achlioptas *et al.* 2000), instances generated using

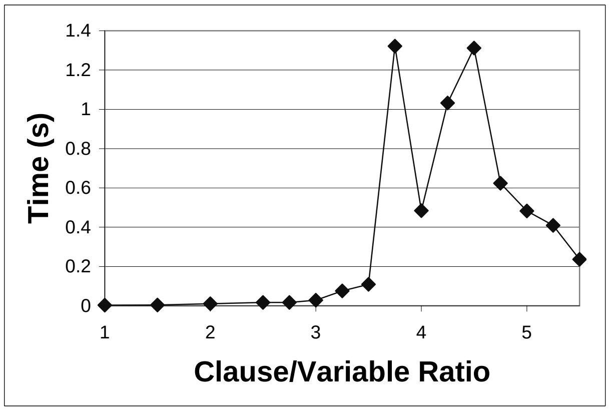

Figure 5: Average time taken to solve 10 satisfiable 3-CNF formulae with 60 variables for varying clause-variable ratios

such a procedure tend to have some kind of bias and are usually easier to solve. So, instead we generated satisfiable instances by generating random instances of SAT and then verifying that they are satisfiable using the publicly available solver *zChaff* (Moskewicz *et al.* 2001).

Restart Optimization

During our experiments we noticed that many times the initial random choice of assignments made could have a large impact on the success or failure of the algorithm. With this in mind we hypothesized that performance could be improved by restarting the WalkSAT at various times. By looking at related work (Moskewicz *et al.* 2001), we found three types of restarting optimizations that we could perform:

- **Retry:** If the algorithm fails, run it again up to n more times
- **Fixed Time Restart:** If the algorithm runs for a certain amount of time without finding a solution, then restart (this is similar to retrying).
- **Randomized Restart:** With some probability after each iteration, there is a chance of restarting. As the algorithm runs longer there is a higher probability that the search will be restarted.

Not only do we have intuition into why restarting should be successful, but also from Figures 2 and 4 we can see that there is a significant gap between the time on average it takes to fail and the time it takes to return a satisfying assignment. This means that as we spend more time on the answer then we can be confident that either there is no answer, or that we are stuck on a local maximum – and so restarting would be helpful.

We implemented the basic retry mechanism to evaluate how it affects the performance of the WalkSAT algorithm. Figure 6 shows the accuracy of the algorithm as a function of the number of retries performed. This graph was constructed such that the total number of flips remained constant through all runs of the algorithm. As such, for the case with 0 restarts we used a maxFlips of 800, for 1 restart we used a maxFlips

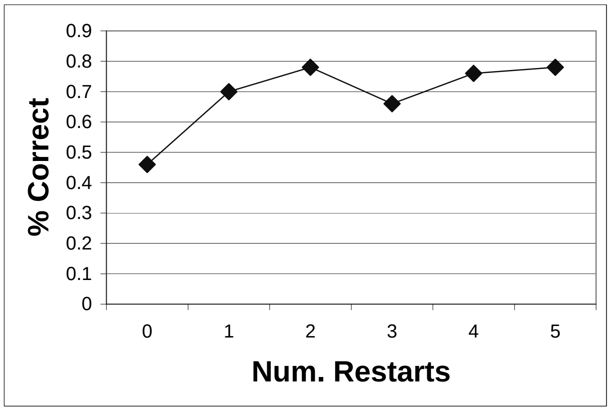

Figure 6: Percentage of the satisfiable 3-CNF formulae that WalkSAT produced a satisfying assignment for over varying numbers of restarts. All had a total number of flips of 800, a random flip probability of 0.5, and a clause-variable ratio of 4.6

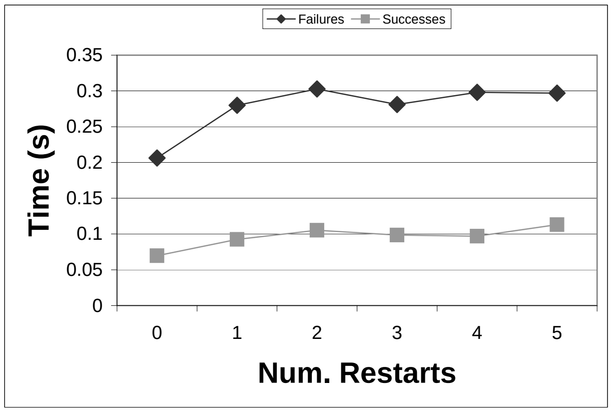

Figure 7: Average time taken to solve 10 satisfiable 3-CNF formula (with 5 runs for each). The two lines represent the time taken when a solution is eventually found and when a solution is not found

of 400 (so that the total remained 800), etc. In this way we can effectively evaluate the restart mechanism without bias. The results are somewhat surprising, we see that there is a large improvement by using just 1 restart; however, there is no improvement as the number of restarts is increased.

In addition, we also explored the time it took to run the algorithm with varying numbers of restarts. These results are reported in Figure 7. We can see that there is little effect on the time it takes to find a satisfying assignment and only a moderate effect on the time it takes to fail. This makes sense since the purpose of restarting is only to improve accuracy and not to improve the speed of the algorithm.

Summary

Overall the performance of WalkSAT is very good when it is able to find the solution. However, because the algorithm is not complete WalkSAT has limited use when searching for unsatisfiability. We attempted to improve the performance of WalkSAT by adding restarts; we found that having restarts gives a slight increase in accuracy and the effect on running time is negligible.

DPLL

Introduction to DPLL

The second satisfiability solver that we implemented was based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm. This algorithm essentially performs a depth-first search of the assignment tree. The only distinguishing feature of this algorithm from a naive depth-first search is the use of intelligent heuristics to guide the order in which the tree is traversed. At every step, instead of picking a random unassigned symbol and a random assignment for it to branch on, it employs a couple of heuristics for choosing the literal to branch on.

There are two basic heuristics used for choosing the literal to branch on.

- The first heuristic is called the *"pure symbol"* heuristic. The basic idea underlying this heuristic is that if a symbol occurs as the same literal in all the clauses, then it might as well be assigned the value that makes that literal true. This is because this assignment is guaranteed not to conflict with any other assignments as this assignment does not introduce a false literal in any clause. Also, only the currently unsatisfied clauses need be considered for finding such a symbol as it does no harm if a literal in an already satisfied clause is falsified.
- The second one is called the *"unit clause"* heuristic. In this heuristic, the algorithm checks whether there exists any clause which has only one literal, the symbol of which has not yet been assigned any value. If there does exist such a clause, then clearly satisfying the only literal in that clause is the only means of satisfying the clause. Note that while finding a unit clause, it is not necessary that we just look out for uni-literal clauses. In a multi-literal clause, if all but one clause have already been falsified, then in this case too, the one unassigned literal can be treated as a unit clause.

Other than these heuristics to decide on the symbol and its assignment to branch on, the DPLL algorithm also applies a couple of pretty obvious tests to prune the search along a particular branch if it becomes known that a solution cannot be obtained by going further down the path. The two tests that are employed are:

- A check is made whether all the clauses are already satisfied. If yes, it implies that a satisfying assignment has already been found and hence, there is no need to search any further. The current assignment of the symbols is a satisfying assignment.
- A check is made whether there exists any clause wherein all the literals have already been assigned, and the resulting value for the clause is false. If yes, then it is futile to continue searching along this path as any assignment of the currently unassigned symbols cannot lead to this clause being satisfied. So, backtracking is initiated instead.

In short, the essence of this algorithm can be summarized as below.

```
$dpll(clauses, symbols, model)$
{
    **if** (all clauses are satisfi ed)
        **return** TRUE;
    **if** (some clause is false)
        **return** FALSE;
    $(P, v)$ = FIND_PURE_SYMBOL(clauses, symbols, model)
    **if** (P not equal to NIL)
        **return** dpll(clauses, symbol, EXTEND(model, $P$, $v$));
    $(P, v)$ = FIND_UNIT_CLAUSE(clauses, symbols, model)
    **if** (P not equal to NIL)
        **return** dpll(clauses, symbol, EXTEND(model, $P$, $v$));
    $P$ = PICK_UNASSIGNED_SYMBOL(symbols, model)
    Heuristically pick some value $v$ for $P$
    if (dpll(clauses, symbols, EXTEND(model, $P$, $v$)) == TRUE)
        **return** TRUE
    **return** dpll(clauses, symbols, EXTEND(model, $P$, $\tilde{v}$));
}
```

In the pseudocode of the DPLL algorithm presented above, there is some room for choice only at the following three steps:

- Picking the pure symbol when more than one exist
- Picking the unit clause when more than one exist
- Picking an unassigned symbol and an assignment for it when neither a pure symbol nor a unit clause exists

There is no advantage in picking the *"right"* pure symbol or unit clause when more than one exist. This is because in each step of the recursion, DPLL first attempts to assign pure symbols and unit clauses before branching on any other variable. So, all the pure symbols and unit clauses would be assigned before assignment of any other variable is attempted. This implies that the only room for optimization is in choosing the right variable to branch on when neither a pure symbol nor a unit clause exists. The simple heuristic we tried out for this was to choose the variable and its assignment which would cause the maximum number of unsatisfied clauses to become satisfied. We henceforth refer to this variant of DPLL as optimized DPLL and the version where we just pick a random unassigned symbol as unoptimized DPLL.

DPLL Experimental Results

We first studied the performance of optimized DPLL. For this, we generated satisfiable SAT instances with different number of variables and determined that our solver takes a non-trivial amount of time, *i.e.*, of the order of seconds and not of the order of milliseconds, to solve the problem when the number of variables is greater than 60. So, we considered the cases when number of variables is equal to 70 and 75. In either case, we considered a set of clause to variable

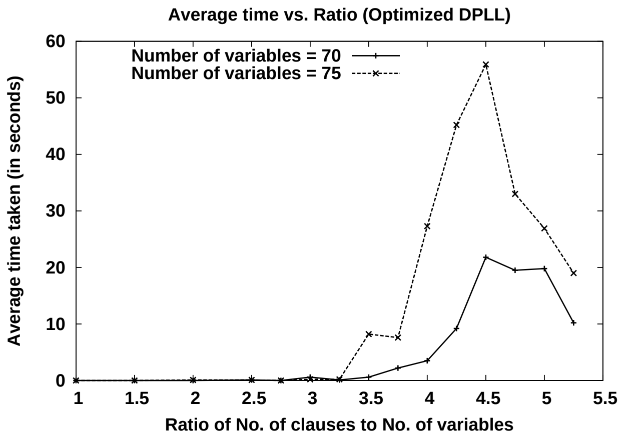

Figure 8: Variation of average time taken with number of clauses to number of variables ratio

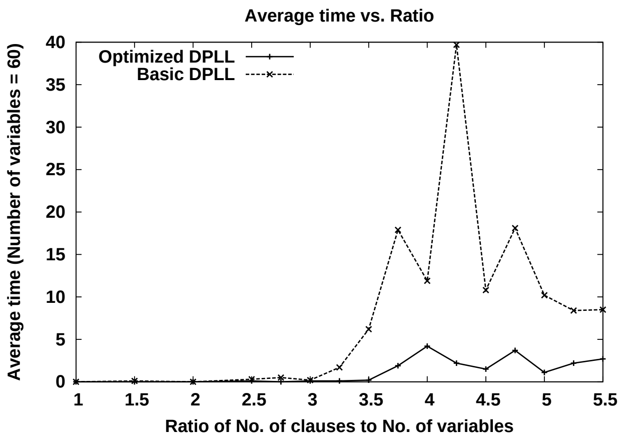

Figure 9: Comparison of average time taken by optimized DPLL and unoptimized DPLL

ratios in the range 1 to 5.5[1], and for each ratio, generated 10 satisfiable problem instances.

Figure 8 plots the variation of average time taken to solve the satisfiable instances that we generated. It is a well-known result that solving random satisfiable instances is hardest when the ratio of number of clauses to number of variables is around 4.33. We see that this fact is validated in Figure 8 where we see that the average time taken peaks at ratio of number of clauses to number of variables equal to 4.5.

We next studied what advantage is offered by the heuristic we used to build optimized DPLL. Our explorations with unoptimized DPLL showed that the time it takes to solve is considerably longer than that for optimized DPLL for the range of number of variables we considered above. So, in Figure 9, we instead show the comparison of time taken

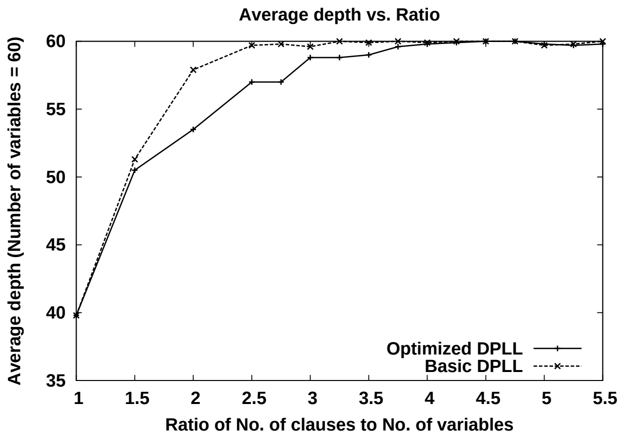

Figure 10: Comparison of maximum depth traversed during search by optimized DPLL and unoptimized DPLL

by optimized and unoptimized DPLL for different clause to variables ratios when the number of variables is fixed at 60. The graph clearly highlights the improvement in performance gained due to the heuristic employed in optimized DPLL to choose the literal to branch on.

The potential causes for this improved performance could be:

- The search has to reach a lesser depth than what it would have originally, as some variables can be arbitrarily assigned
- The search gets directly along the right path

In other words, the savings could be either due to the fact that the search goes to a shallower depth or due to the fact that the search was focused in the right part of the tree. To determine which of these cases was true, we studied the maximum depth the search traverses during its execution, both in the case of optimized DPLL as well as for unoptimized DPLL. Figure 10 shows that the gain in depth is not much and in fact, both searches traverse the search tree till the maximum depth of 60 for most ratios. So, this shows the gain is more in terms of directing the search in the right region of the tree rather than decreasing the depth until which the search has to be performed.

HybridSAT solver

The key insight we gained from our experiments with the DPLL solver was that the performance can be improved significantly by choosing the "right" variable to branch on, when neither a pure symbol nor a unit clause exists. Moreover, the optimized DPLL highlighted the fact that choosing the "right" variable led to exploring the "right" region of the tree, which was the reason for better performance.

Although the heuristic we use for the optimized DPLL performs much better than the trivial approach of picking a variable randomly, it is still very simple. Choosing the literal that maximizes the number of unsatisfied clauses getting satisfied is, at best, a myopic greedy strategy. It would

[1]Finding satisfi able problem instances for clause to variable ratios greater than 5.5 was extremely hard.

be more fruitful to choose the literal that would achieve the same even after a much larger number of steps. We do this by running WalkSAT for a few steps for each of the unassigned symbols, and choosing that symbol that satisfies the maximum number of unsatisfied clauses. This idea forms the basis of our new SAT solver which is essentially a hybrid of DPLL and WalkSAT; and hence the name!

There is a subtle point though that we need to take care of. Both in a “min-conflicts” and a “random walk” step, WalkSAT picks up a symbol to flip randomly. But some of these symbols might have already been assigned by DPLL during its search - and we should not meddle up those assignments. Therefore, WalkSAT needs to be modified so that it is restricted to assign only those symbols which were left unassigned by DPLL; the symbols which have already been assigned by DPLL should not be changed. This appears as two changes in the WalkSAT code. Firstly, the clause that is randomly selected from the set of clauses which are false in the model should meet the added requirement that it has at least one literal that was not assigned by DPLL. Secondly, the symbol that is flipped should not have been assigned by DPLL.

HybridSAT Experimental Results

HybridSAT has the same two tunable parameters that WalkSAT has, that affect its performance. Our first set of experiments are targeted to find the operating point for the solver. For this, we chose 10 satisfiable formulae with 60 variables and clauses-to-variable ratio of 4.25; and ran HybridSAT for different values of maxFlips and the probability, p. Figure 11 plots the the variation of the average time taken to solve the instances versus maxFlips for different values of p. The more the number of maxFlips, the more area gets searched and thus, the better is the symbol that is chosen. On the other hand, the more the maxFlips, greater is the constant amount of time the WalkSAT component of the solver spends in picking up a symbol. We see this tradeoff in the graph with most curves converging to a minimum close to 400. Figure 12 plots the same numbers as Figure 11 but in a different visualization - it shows the variation of the average time taken to solve these instances with p for different numbers of maxFlips. Although not very pronounced, the common trend in all the curves Is that they decrease till some value of p between 0.6 and 0.8; and increase after that.

Based on these two figure, we choose maxFlips as 400 and p as 0.7 as the operating point for HybridSAT.

We move on to comparing the performance of HybridSAT with DPLL. For this we ran HybridSAT on the same set of formulae as in Figure 8. Figure 13 plots the variation of the average time taken to solve the satisfiable instances by both the optimized DPLL and HybridSAT. For all clauses-to-variables ratios of both the 70 and the 75 variable instances, we notice that the HybridSAT outperforms optimized DPLL.

Conclusions

We implemented and analyzed the performance of WalkSAT and DPLL. In our analysis of WalkSAT, we showed how its

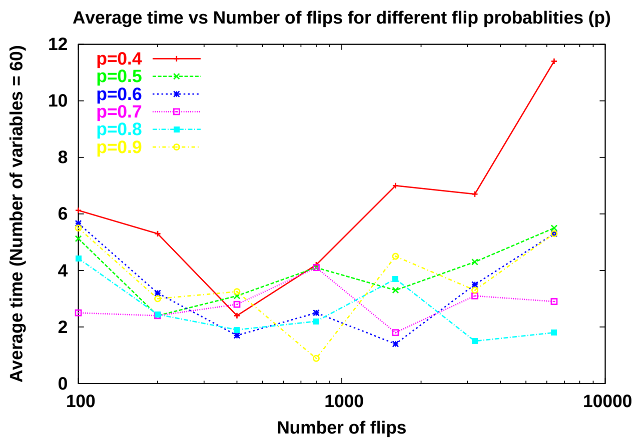

Figure 11: Variation of average time taken by HybridSAT with maxFlips for different values of probability to do the random-walk step

performance depends on each of the parameters the algorithm requires. We also showed that introduction of randomized restarts helps improve the accuracy of WalkSAT without resulting in a significant increase in running time. We identified a key optimization that can be added on to the basic DPLL algorithm to reduce the running time significantly. Drawing on these results, we implemented a new SAT solving technique, called HybridSAT. HybridSAT combines the strengths of WalkSAT and DPLL to reduce running time without sacrificing in completeness. All our results were substantiated through extensive experiments.

References

Achlioptas, D.; Gomes, C. P.; Kautz, H. A.; and Selman, B. 2000. Generating satisfiable problem instances. In *Proceedings of AAAI'00*.

Davis, M.; Logemann, G.; and Loveland, D. 1962. A machine program for theorem-proving. *Commun. ACM* 5(7):394–397.

De, B. L. 2002. Heuristic backtracking algorithms for sat.

Goldberg, E.; Prasad, M.; and Brayton, R. 2000. Using sat for combinational equivalence checking.

Kautz, H. A., and Selman, B. 1992. Planning as satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, 359–363.

Moskewicz, M.; Madigan, C.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an efficient sat solver. In *Proceedings of Design Automation Conference 2001*.

Russell, S., and Norvig, P. 2003. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition.

Selman, B.; Kautz, H. A.; and Cohen, B. 1993. Local search strategies for satisfiability testing. In Trick, M., and Johnson, D. S., eds., *Proceedings of the Second DIMACS Challange on Cliques, Coloring, and Satisfiability*.

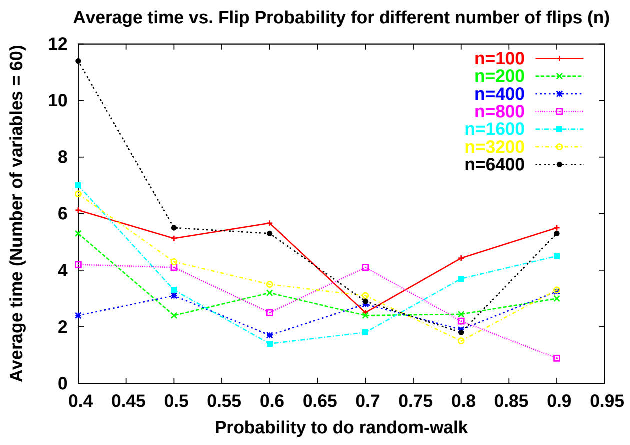

Figure 12: Variation of average time taken by HybridSAT with the probability to do random-walk step, for different values of maxFlips

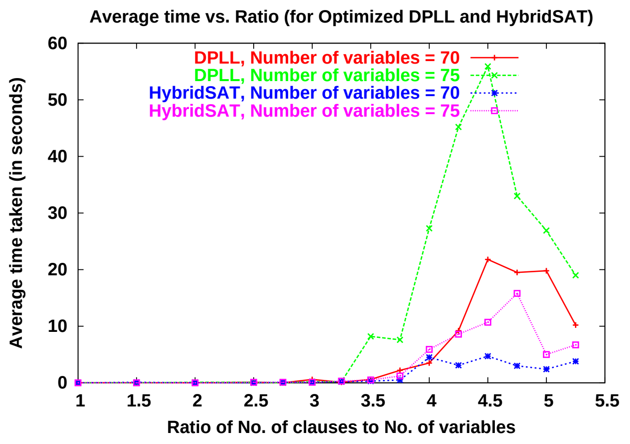

Figure 13: Variation of average time taken with number of clauses to number of variables ratio

Selman, B.; Levesque, H. J.; and Mitchell, D. 1992. A new method for solving hard satisfiability problems. In Rosenbloom, P., and Szolovits, P., eds., *Proceedings of the Tenth National Conference on Artificial Intelligence*, 440–446. Menlo Park, California: AAAI Press.

CONTRIBUTIONS

- **Ankur:** Wrote the HybridSAT code, wrote the Hybrid portion of the paper.
- **Harsha:** Wrote the DPLL code, wrote the SAT generation script, wrote the DPLL, Abstract, and Conclusions portion of the paper.
- **Craig:** Wrote the WalkSAT code, wrote the WalkSAT, Intro, and Related Work portions of the paper.

We used the zChaff code in order to check for satisfiable assignments for testing our code and for generating only satisfiable formulae for our experiments.