WalkSAT as an Informed Heuristic to DPLL in SAT Solving

Brian Ferris and **Jon Froehlich**
Department of Computer Science
University of Washington
Seattle, WA 98195
bdferris@cs
jfroehli@cs

Abstract

It is well known that stochastic search methods tend to outperform systematic search approaches in solving randomly generated SAT problems. Typically, stochastic local search algorithms like GSAT and WalkSAT can solve hard, randomly generated problems that are significantly larger than those handled by traditional complete search algorithms like DPLL. However, unlike DPLL, local search algorithms are not complete and, as a consequence, cannot prove unsatisfiability. Therefore it is desirable to find a hybrid of these two approaches that leverages the strength of both. First, we present a set of initial experiments generated to find exploitable areas in DPLL and WalkSAT. Second, we propose that WalkSAT is a candidate for generating dynamic heuristics for DPLL. Finally, we present a few novel hybrid implementations of WalkSAT and DPLL.

Background

Propositional logic defines a simple logic for representing and reasoning about knowledge mathematically in computation-based systems. It provides a syntax and describes its semantics, which establishes the allowable propositional sentences and the meaning of those sentences (i.e. how the truth of a sentence can be determined). Sentences are composed of symbols linked together using logical connectives (like AND and OR). The symbols themselves represent a proposition that can either be TRUE or FALSE. The simplest sentence, then, would consist of a single symbol (e.g. Sentence = A). A slightly more complicated example would be, Sentence = (A OR B).

Conjunctive normal form (CNF) defines a special format for propositional logic. Every sentence of propositional logic can be equivalently expressed in CNF. It is therefore used as a way to standardize logical formulas. A sentence in CNF is expressed as a conjunction of disjunctions of literals. Each disjunction of literals is called a clause. Formally, a CNF sentence: $(l_{1,1} \vee \ldots \vee 1_{1,x}) \wedge \ldots \wedge (1_{n,1} \vee \ldots \vee 1_{n,y})$. This is the format used by satisfiability solvers.

We say a sentence is satisfied if its symbols can be set such that the logic expressed in the sentence results to TRUE. For example, a satisfiable solution to the following CNF sentence (A OR B) AND (C OR B) would be A=TRUE, C=TRUE. Satisfiability, then, is concerned with determining if a given sentence is satisfiable and identifying a satisfying assignment for it. Many problems in computer science are really satisfiability problems (Russell & Norvig 2003). With appropriate transformations, for example, search problems can be solved by checking satisfiability. Other problems such as circuit verification, the graph coloring problem, planning problems, and scheduling problems can also be encoded into SAT.

Unfortunately SAT is NP-Complete and therefore intractable in the worst case. Despite this difficulty, much research has been dedicated to solving hard SAT problems efficiently[1]. Two distinct approaches have been developed: stochastic local search algorithms like WalkSAT (Selman, Kautz, & Cohen 1996) and systematic search algorithms like DPLL (Davis, Logemann, & Loveland 1962). As in other application domains, stochastic search is generally much faster than its deterministic counterpart at finding a satisfiable solution. However, local search algorithms like WalkSAT are hindered by incompleteness. That is, if WalkSAT does not find a satisfiable solution for a given problem, no conclusion can be drawn about the satisfiability of that problem. DPLL, on the other hand, is a systematic, complete algorithm and can therefore prove unsatisfiability. Simply put, the algorithmic trade-off tends to be speed for completeness.

For this reason, successfully combining stochastic and systematic search techniques is a luring proposition. In the paper, *Ten Challenges Redux: Recent Progress in Propositional Reasoning and Search*, Henry Kautz and Bart Selman (1997) present ten challenges for research on satisfiability testing. In Challenge Seven, "Randomized Systematic Search," the authors encourage researchers to investigate hybrid approaches to SAT solving.

Two specific approaches are cited in Ten Challenges as examples of SAT solvers that integrate both local search and DPLL search methods into one SAT algorithm (Habet *et al.* 2002; Mazure, Sas, & Grgoire 1998). The first example, introduces WalkSatz, a combination of WalkSAT and Satz (Li & Anbulagan 1997), a preexisting DPLL solver with improved branching rules. WalkSatz begins by running Satz.

[1]Efficiently is used loosely here. Because SAT has been shown to be NP-Complete, it's unlikely that a polynomial time SAT solver exists (unless P=NP). Instead, an efficient solver might perform well on average, or with high probability, or on a class of interesting inputs (Cook & Mitchell 1997).

At each node in Satz's decision tree, a complete implication graph is constructed and minimized. A slightly modified version of WalkSAT is applied to this resulting graph. WalkSAT terminates if either a satisfiable solution is found or a maximal fixed constant is reached. In the latter case, WalkSatz continues to the next node in the decision tree and the process is repeated, otherwise WalkSatz returns true.

Conceptually, WalkSatz works by taking advantage of a well known technique in local search algorithms that exploit variable dependencies. In this case, analysis of variable dependencies is extended to implications and equivalencies between literals. The implications themselves are naturally performed by Satz during branching such that the implication graph can be constructed in linear time. The key to WalkSatzs success, and the underlying aspiration of all hybrid approaches, is to intelligently combine completeness with randomization such that the algorithm contains the best of both worlds. As Habet states, whenever Satz works well, WalkSatz takes advantage in robustness, and when Satz works less well, there is a small improvement in robustness at the expense of computation time. On random SAT problems containing few variable dependencies, Walksatz and Walksat essentially have the same behavior, (Mazure, Sas, & Grgoire 1998).

Mazure et al. take a much different approach to building a hybrid solver. While both begin by starting a version of the DPLL algorithm and making consecutive calls to a local search algorithm (in this case TSAT), Mazures algorithm, called DP+TSAT, uses the unsatisfiable local search result to build heuristic data for DPLL. More precisely, an instrumented version of TSAT (Mazure, Sas, & Grgoire 1997) is run at each step in the decision tree to calculate the next literal to be assigned by DPLL. The literal selected by TSAT is based on two trace records. The first trace counts the number of times each clause is falsified where a step of time is one flip. The second trace is recorded for each literal occurring in the SAT instance, keeping track of the number of times the literal appears in falsified clauses. Each time DPLL needs to select a new variable, a call to TSAT can be performed with respect to the remaining part of the SAT instance.

Essentially, the trace of TSAT is used as a heuristic for selecting the next literal to be assigned by DPLL. If the SAT instance is unsatisfiable, the most often false clauses are likely to belong to an unsatisfiable subset of the SAT instance. Similarly, the literals that receive the highest counts should also be part of this unsatisfiable subset. DP+TSAT was empirically shown to be effective. For satisfiable problems, DP+TSAT performed comparable to a stochastic local search algorithm (as DP+TSAT begins with a call to TSAT). For certain unsatisfiable problems, DP+TSAT was also shown to perform very well. For example, it solved a real-world circuit fault analysis problem in 32 seconds while other solvers such as C-SAT and DP+FFIS failed to prove this problem unsatisfiable within 73 hours and 17 hours CPU time respectively (Mazure, Sas, & Grgoire 1997).

It should be noted that invoking randomization in systematic search does not necessarily require integrating stochastic local search. Modern DPLL implementations have improved dramatically by adding randomization and restart strategies. Chaff (Moskewicz *et al.* 2001), for example, uses clause learning, variable selection heuristics, and randomized restarts to search different spaces on different restarts of the algorithm. For our purposes, however, WalkSatz and, particularly, DP+TSAT are more relevant to our approach here.

WalkSAT and DPLL

As mentioned above, WalkSAT and DPLL are in two different classes of algorithmic SAT solvers. WalkSAT is a type of stochastic local-search algorithm while DPLL is a complete backtracking search algorithm. In this section, we will explain these algorithms in more detail including an exposition of their comparative strengths and weaknesses and a description of our investigative experiments. We will start with some background information on the hard random problems in 3-SAT as this is an absolutely critical area for both WalkSAT and DPLL.

Hard Random Problems

According to Russel and Norvig, a problem is underconstrained if its solutions are very densely distributed in the space of assignments, and any initial assignment is guaranteed to have a solution nearby. For example, suppose we had the following 3-CNF sentence with five symbols and five clauses:

$$\begin{gathered}(\neg D \vee \neg B \vee C)\wedge \\ (B \vee \neg A \vee \neg C)\wedge \\ (\neg C \vee \neg B \vee E)\wedge \\ (E \vee \neg D \vee B)\wedge \\ (B \vee E \vee \neg C)\end{gathered}$$

There are 16 possible assignments that result in a satisfiable solution out of a total of 32. This means that, on average, it would take two random guesses to find the model [Russel and Norvig].

It turns out that we can easily identify underconstrained problems in random 3-SAT by looking at the clause to symbol ratio (C/S). In the example above there were five clauses and five symbols, resulting in a C/S = 1. When C/S is very low (<3.5), problems will generally be satisfiable and solved in a tractable number of steps.

A problem is overconstrained if it has a large (>5.0) C/S ratio. Like underconstrained problems, overconstrained problems are likely to be resolved in a tractable number of steps; however, unlike underconstrained problems, the solution here is typically unsatisfiable. In both of these extreme cases, it's likely that the computational cost will scale like a low-order polynomial function of n (O'Donnel 2002). It is possible, though unlikely, to have unsatisfiable problems in the underconstrained area just as it is possible to have satisfiable problems in the overconstrained area.

For randomly generated SAT, the hard problems tend to fall between 3.5 and 5.0. More precisely, they have a C/S ratio of 4.3. As can be inferred from the Graph 2, CNF sentences near this critical point are typically very difficult to solve.

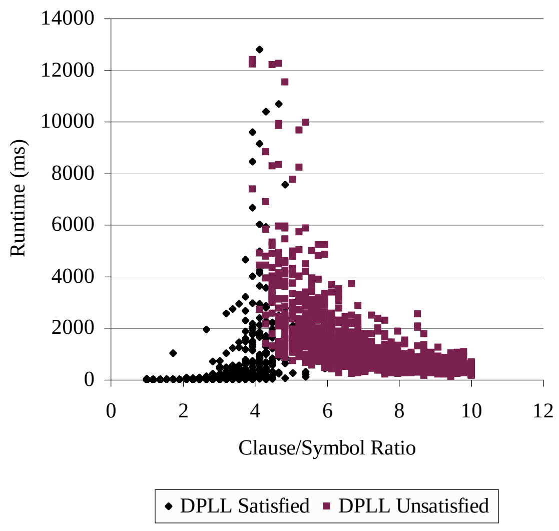

Figure 1: Clause/Symbol Ratio SAT/UNSAT Breakdown in DPLL

WalkSAT

Local-search algorithms are known for their speed and simplicity. WalkSAT begins by randomly assigning every symbol in its input CNF sentence to either TRUE or FALSE. From there, it takes incremental steps in the space of complete assignments, flipping the value of one symbol at a time (Russell & Norvig 2003). The manner in which WalkSAT selects the next symbol to flip and its subsequent truth value assignment is the crux of the algorithm.

Like all local-search algorithms, WalkSAT suffers from getting trapped in local-minima bounds. A popular strategy to reducing this problem is to insert some randomness into the search algorithm. Rather than always selecting the best move (the greedy selection), WalkSAT alternates between greedy moves and noisy moves - those moves which are randomly selected from the variables that appear in unsatisfi ed clauses (Selman, Kautz, & Cohen 1994). The WalkSAT algorithm is based on the insight that such noisy moves could be made the basis for local search. The WalkSAT algorithm follows:

1. Randomly assign TRUE/FALSE to the symbols in the input CNF sentence
2. On every iteration (up to MAX-FLIPS), pick an unsatisfied clause C and pick a symbol in that clause to flip.
3. Choose randomly with probability p which of the following flipping strategies to use
   (a) Flip the value of a randomly selected symbol from C
   (b) Flip whichever symbol in C maximizes the number of satisfied clauses in our input CNF sentence
4. If all clauses are satisfied, return true. If not, repeat at step 2.

The default implementation of WalkSAT contains three variables that can be tuned according to the problem space: MAX-RESTARTS, MAX-FLIPS, and flip probability p. In the case of random SAT, we have found that, in general, the best fi xed value for p is approximately 0.5. We ran WalkSAT on 1,068 problems with evenly distributed clause-symbol ratios [from C/S=1 to C/S=10]. For each problem, WalkSAT was executed ten times with ten different values of p:

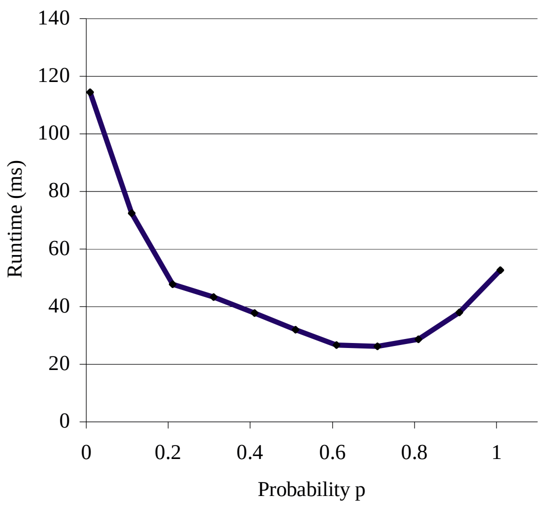

Figure 2: Random Flip Probability For Satisfi able Problems

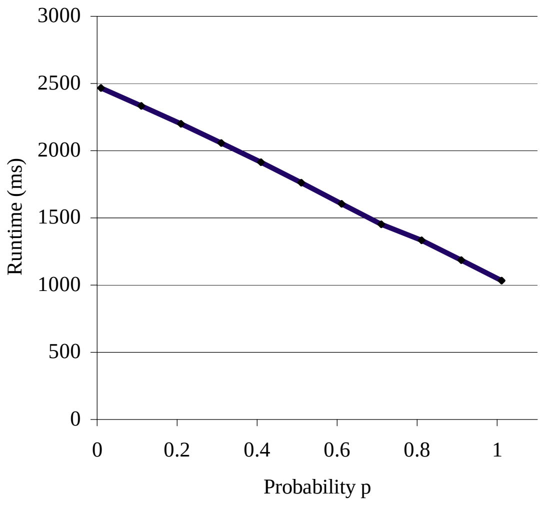

Figure 3: Random Flip Probability For Unsatisfi able Problems

Note that extreme values for p result in worst-case performance. For example, a greedy implementation of WalkSAT (when p=0.01) is 4.4 times slower than the fastest average runtime (when p=0.6) for satisfi able problems. Somewhat surprisingly, for hard problems (those that WalkSAT was unable to fi nd a solution to) setting p=1.0 results in the best-case performance. We believe that this is because of the computational overhead involved in calculating which symbol to flip in Step 3b of the algorithm. When p=1.0, WalkSAT simply fails faster for hard problems because it does less work.

A main criticism of local-search algorithms is that they are incomplete. Indeed, when WalkSAT is run on hard problems and is unable to fi nd a satisfi able solution, we are un-

WalkSAT Incorrect Solution Rate			
Random Problems	Number of Tests	Clause/Symbol Ratio Range	WalkSAT Percentage Incorrect
Normally Distributed	2018	[1,10]	0.64%
Hard Problems	514	[3.5,4.5]	17%

Figure 4: WalkSAT Incorrect Solution Rate

able to conclusively say a problem is unsatisfiable; to do that, we need a complete algorithm (like DPLL). To better understand how often WalkSAT fails to find a satisfiable solution when one exists, we conducted a series of simple tests. The first test randomly generated 2,018 problems with clause-symbol ratios evenly distributed over the range C/S = [1,10]. Both WalkSAT and DPLL were run on the test set and their results were compared (in terms of sat and unsat). Because DPLL is complete, a disagreement in solutions implies that WalkSAT incorrectly identified a problem as unsatisfiable. The second test generated 514 random hard problems with a clause-symbol ratio evenly distributed over the known difficult C/S range of [3.5, 4.5]. Again, their solutions were compared to determine if WalkSAT returned failure on a satisfiable problem. The results of these two experiments are shown in Figure 4).

For an evenly distributed clause-symbol ratio problem set, WalkSAT is unable to find a satisfiable solution when one exists less than 1% of the time. This is, in most circumstances, a very acceptable percentage given the tradeoff of speed vs. completeness. However, to contextualize this optimism it should be noted that for hard problems (Figure 3 above), WalkSAT returned failure on satisfiable problems 17% of the time. Though this was expected, it is a less encouraging result. It should be noted that proving unsatisfiability is an important problem in comptuer science. Proving unsatisfiability is the final objective in several practical CS applications, including automated theorem proving in AI, circuit verification and circuit delay computation in EDA (Lynce & Silva 2003). Clearly a complete solution is needed in these cases.

DPLL

One of the oldest and surprisingly still relevant complete algorithms for SAT solving is called DPLL (named after the authors who invented it). The algorithm is essentially a recursive, depth-first enumeration of all possible assignment models in the problem space. DPLL uses a few tricks to reduce the search space but, overall, its efficiency is dependent primarily on branching and symbol ordering. The algorithm works as follows:

```
DPLL( clauses, symbols, model )
if every clause is true then
    return true
end if
if some clause is false then
    return false
end if
UnitLiteral() and PureLiteral() propagation
S ← Next( symbols )
R ← Rest( symbols )
for v in DomainValue(S) do
    if DPLL(clauses, symbols, model) then
        return true
    end if
end for
```

DPLL is obviously a much more complicated algorithm than WalkSAT; however, it's not necessary for the purposes of our discussion that specifics of the algorithm be understood. Instead, note the following three key characteristics. First, that DPLL is capable of detecting whether an input CNF sentence is TRUE or FALSE with a partially completed model. That is, DPLL can resolve clauses that do not have truth assignments for all of its literals. For example, the sentence $(D \vee \neg C) \wedge (E \vee \neg C)$ can be resolved by setting C to FALSE without inspecting D or E. This allows DPLL to avoid examination of certain subtrees in the search space. Second, that DPLL uses the two methods PureLiteral() and UnitLiteral() to further restrain its search space by attempting to select the most critical symbols and clauses earlier on in its search space. And finally, that truth assignments exist as branches in the decision tree. Selecting the correct truth assignment higher up in the decision tree is critical to finding a solution efficiently, as it reduces needless backtracking.

Because DPLL is complete, in the worst case, it must enumerate all possible assignment models in the search space. For hard problems, this tends to dramatically increase the runtime of the algorithm (as can be seen in Figure 5). In this figure, WalkSAT runtimes are plotted in black while DPLL runtimes are plotted in red. Note the consistency in runtime spikes between both algorithms on those problems with a C/S of 4.3. Further, notice that WalkSATs runtimes become increasingly longer after the 4.3 ratio. This is because upon reaching an unsatisfiable solution, WalkSAT is unable to conclude that the input problem itself is unsatisfiable. On the contrary, because of its completeness, DPLL runtimes steadily decrease after the 4.3 spike because it quickly determines that the 3-CNF sentence is unsatisfiable.

Heuristics

As mentioned before, our exploration of SAT solver optimizations is primarily inspired by one of the ten challenges outlined by Kautz, et al (Selman, Kautz, & McAllester 1997). Specifically, we wish to combine the speed of local search algorithms such as WalkSAT with the completeness of DPLL. There are a variety of ways to approach this challenge, but we focus directly on the idea of using cumulative statistics from partial executions of WalkSAT to inform the DPLL search algorithm. Such a hybrid approach retains the completeness of DPLL by using WalkSAT as a heuristic during branching decisions, but ultimately still allows complete exploration of the search space. We attempt a number of possible WalkSAT-based heuristics.

Backbone Set Heuristic

A key concept in satisfiability problems is that of the backbone set (Slaney & Walsh 2001). The backbone set for a given satisfiability problem is the set of all symbols whose

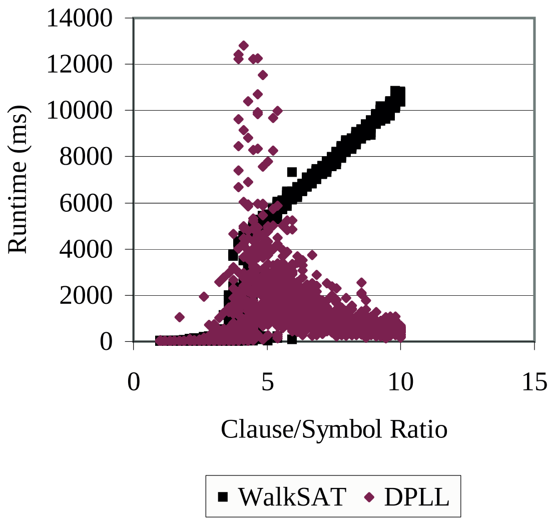

Figure 5: DPLL/WalkSAT Runtime verus Clause/Symbol Ratio

assignment does not change across all possible solutions to the problem. We wish to answer a number of questions about backbone sets. Can the backbone set be identifi ed by WalkSAT? Once identifi ed, does knowledge of the backbone set allow optimized execution of DPLL? In our exploration of these two questions, we fi rst attempt to characterize the properties of the backbone set over a random problem space.

We empirically derived the backbone set by modifying DPLL to follow all branches in order to produce all valid solutions for a given problem. We then compared the size of the produced backbone set with the ratio of clauses to variables over a range of randomly generated problems.

Figure 6 illustrates the clause/symbol ratio versus backbone set size for the range of randomly generated problems. The graph shows that the size of the backbone set approaches zero as the clause/symbol ratio approaches 4.3. This relation raises an important concern about our proposed heuristic. We know that generally that neither WalkSAT nor DPLL are very fast when approaching the 4.3 ratio, making this region a prime target for optimization. However, we have observed that the backbone set does not constitute a major subset of all symbols for problems in this region. Intuition suggests that there is little value in identifying what is not there, so attempts to identify the backbone set do not appear to be a worthwhile goal in the pursuit of runtime optimization.

Symbol-Ordering Heuristic

The initial dead-end of backbone sets explored, we stepped back to reevaluate what a good heuristic for DPLL might look like. DPLL in its simplest form identifi es pure and unit literals during the symbol selection phase at each iteration to rapidly reduce the set of clauses needing to be satisfi ed. This identifi cation process is a specifi c form of a symbol ordering heuristic. We choose to focus then on symbol-ordering heuristics as informed by WalkSAT for our next area of exploration.

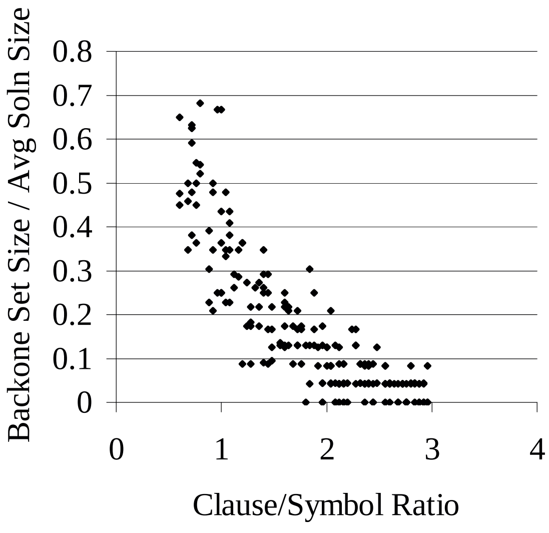

Figure 6: BackboneSet

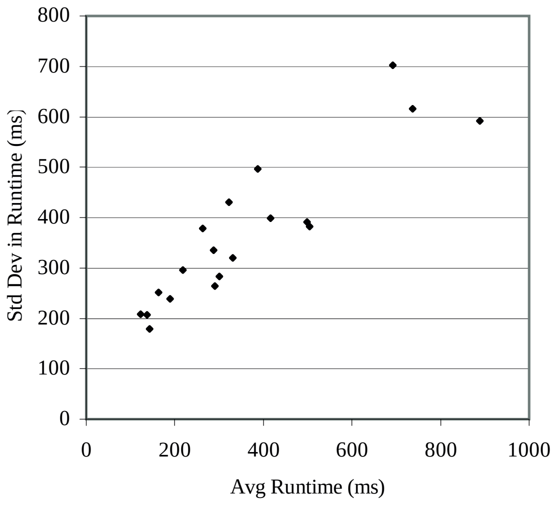

Figure 7: RandomOrder

We wish to fi rst establish that symbol order does have a major impact on the runtime of DPLL. Pure and unit literal optimization aside, we hypothesize that an ideal symbol-ordering exists for minimizing the runtime of DPLL. To test this hypothesis, we modify DPLL to accept an ordered list of symbols which is used to determine the order in which symbols are selected when recursively exploring the assignment space. We then explore a range of randomly generated SAT problems, running DPLL on each problem with a variety of randomly generated symbol orderings.

Figure 7 shows that the standard deviation in DPLL runtime for random symbol-ordering is on the same order of magnitude as average runtime, indicating that symbol-order can produce large swings in runtime, with those swings growing larger as problems grow more diffi cult.

We have shown that different symbol-orderings have pro-

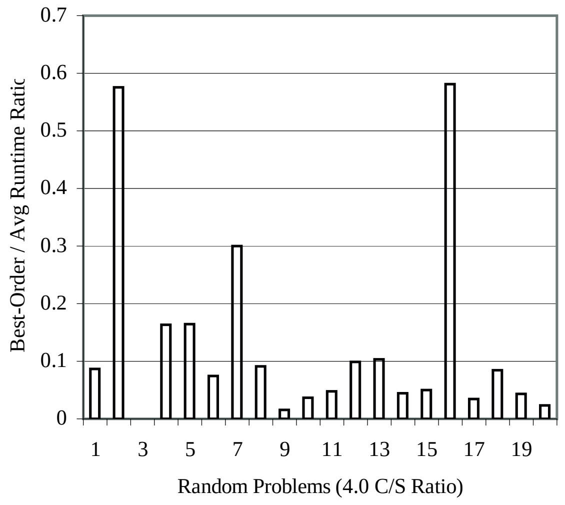

Figure 8: Best Order

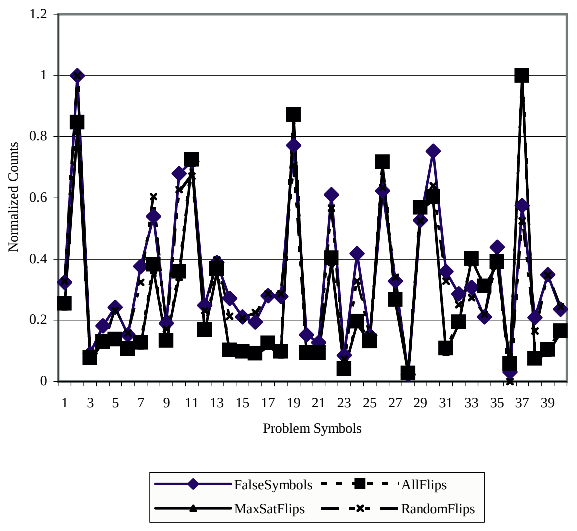

Figure 9: Counts

nounced effects on the runtime characteristics of DPLL. We also observe that there is no single symbol-ordering that produces an improved runtime, but instead often a collection of orderings with improved runtimes. We now wish to generalize some optimal ordering for a given problem. We hypothesize that, if a symbol often appears at the beginning of an ordering for a low runtime or at the end of an ordering for high runtime, the symbol should always be placed towards the beginning of the ordering.

We formalize this notion in the following way. Given some number of runs of DPLL on the same problem with a randomized symbol-ordering for each run, we determine a maximum and minimum runtime across all runs. We then normalize each runtime by subtracting half the difference between the max and min runtime, centering all runtimes around the origin. We also number symbol positions such that the center position is labeled zero, the fi rst position is a negative number and the last position is a large positive number. The normalization of both runtime and position cause their product to be positive for beginning positions that result in low runtime and ending positions that result in high runtime. Beginning positions that result in high runtime and vice versa will have negative products. We can take the average product of runtime and position for each symbol in a problem across all DPLL runs to get a score that, when largely positive, indicates that a symbol should generally be ordered fi rst for DPLL. Given this ordering heuristic, we can create an idealized symbol-ordering for comparison against the average runtime of DPLL with random symbol ordering.

Figure 8 shows the ratio of the best-order runtime to average random-order runtime versus clause/symbol ratio. We see that best-order to average random-order runtime ratio is generally less than one, indicating that our constructed ordering improves on the average case. However, the increase is not dramatic, suggesting that any optimization based on our ordering heuristic will not result in drastic improvements.

Thus far, we have only proven our hypothesis that symbol order does have a major impact on DPLL runtime and that some ideal ordering does exist. We now wish to use WalkSAT to identify this ordering. Our general goal is to run WalkSAT for some reduced period of time on a given SAT problem. If WalkSAT is lucky enough to fi nd a satisfi able solution in this time period, then our task is already complete. Otherwise, we wish to accrue statistics during that operation of WalkSAT that will help us make an informed decision about symbol-ordering for a subsequent run of WalkSAT.

There are a number of statistics that can be gathered during the execution of WalkSAT, but we choose to primarily track the number of times a symbol is flipped, both randomly or because it satisfi es a maximum number of clauses. Additionally, we track the number of times a symbol appears in a false clause during WalkSAT execution. These statistics are both obvious and easy metrics to monitor during solver execution.

Our initial examination of flip counts, both max-sat, random and total flips, as well as false clause counts shows that the three counters are roughly the same for a given symbol in a given problem. Figure 9 shows the normalized counts for each symbol in a given problem, demonstrating the strong correlation between the four counters for each symbol. Notice for the graph that all four counts have roughly the same magnitude for each symbol, as seen by the overlapping data series. Since the counts are effectively the same, we can simplify in accounting, allowing us to only consider total flip counts for each symbol when performing our accounting and analysis.

Unfortunately, fi gure 10 demonstrates there is no real correlation between flip counts and desired symbol ordering. We generated this graph by running fi fty instance of DPLL with random symbol-orders and fi fty instance of WalkSAT on twenty randomly generated problems with a 4.0 clause to symbol ratio. We used the instances of DPLL to generate a best-order symbol-ordering per the previously described method. The instances of WalkSAT were used to calculate

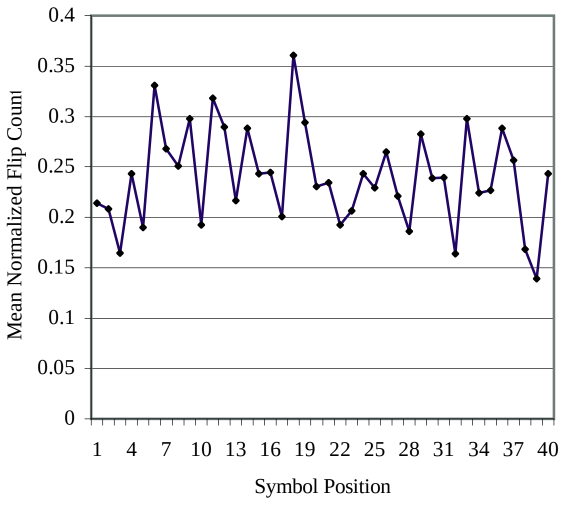

Figure 10: Counts vs Order

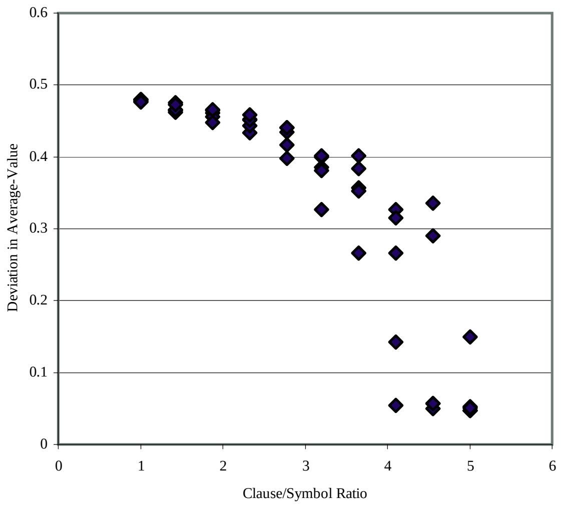

Figure 11: ValuesChangeLess

average flip counts for the symbol space. The graph shows the symbols in best-order from left to right, with each symbols normalized flip count graphed for each problem. The graph shows that there is no cross-problem correlation between best symbol-ordering and flip counts. Just to be certain we ran DPLL on symbol-orders of both increasing and decreasing flip count and neither method did better than average runtime.

Value-Ordering Heuristic

Though symbol-ordering demonstrably plays a role in DPLL runtime, it is not the only target for heuristic optimization. Just as the order in which symbols are selected for assignment is negotiable, so is the order in which domain values for these symbols are assigned. Expanding the assigned-true branch before the assigned-false branch for a given symbol is a potential influence on DPLL runtime.

While value-ordering seems the obvious counterpart to symbol-ordering when exploring DPLL branching, we were especially motivated to explore value-ordering based on a newly considered WalkSAT execution statistic. In addition to flip counts, we can track the portion of time a value spends with a given assignment during the course of solver execution. We record this as a percentage, where zero represents false and one represents true. Thus, a symbol which is mainly valued as false during WalkSAT execution will tend to zero, while symbols valued as true will tend to one. We hypothesize the symbols with high flip counts will tend towards 0.5, as they will alternate between true and false throughout solver execution.

We wish to characterize this statistic for various classes of SAT problems. Our intuition tells us that if a symbol is valued consistently across WalkSAT execution, then the valuation might provide some hint to value-ordering for that symbol in DPLL execution. We record the average value statistic for each symbol in a SAT problem across multiple runs of WalkSAT over a randomly generated problem set. For multiple executions of WalkSAT on the same problem, we record the standard deviation of each symbols average value. The standard deviation tells us roughly if each symbol is assigned consistently across multiple WalkSAT runs, as indicated by low standard deviation, or if the symbol is instead assigned sporadically. We can average all these standard deviations across multiple symbols to get an idea of the stability of assignments for a given problem.

Figure 11 shows the relationship between average assignment stability and clause/symbol ratio for a randomly generated problem set. We know assignment stability is most stable as it approaches zero, since this indicates the smallest mean deviation in symbol assignment over time. We notice that as we approach the 4.3 asymptotic region, assignments become more stable than problems in the rest of the region. This observation is important, since we've hypothesized that a stable assignment might inform value-ordering in DPLL and assignments are most stable in the diffi cult problem region.

Though we've shown stable average-value assignment for WalkSAT approaching the 4.3 asymptote, how does the average-value best inform DPLL? For a randomly generated problem set, we ran both WalkSAT and DPLL to respectively accumulate average-value statistic and to fi nd a known solution for each problem. We then compared the average-value of each symbol with its known solution value and calculated a percentage of matching assignments out of all assignments. The average percentage matching was 61.4 with a standard deviation of 4.21, indicating that the average-assignment does only slightly better than average at determining a symbols solution value.

A Framework for WalkSAT Heuristic Application

Though none of our potential WalkSAT statistic heuristics seem particularly promising, we still wish to follow through on our goal of tying together DPLL and WalkSAT. Recall the general form of DPLL we outline earlier. We extend DPLL

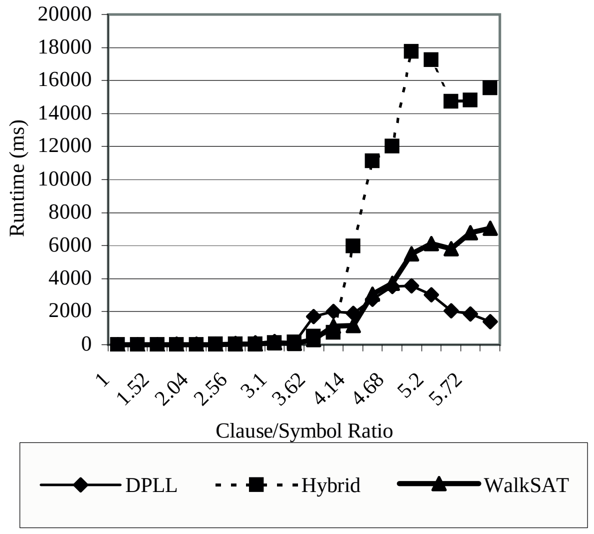

Figure 12: Hybrid

by adding the following code before the unit and pure literal propogation (line 8):

```
1: if some probability then
2:     reduced ← Reduce( clauses, model)
3:     if WalkSAT( reduced, symbols ) then
4:         return true
5:     end if
6:     Adjust Next(), DomainValues() from WalkSAT
7: end if
```

We essentially run WalkSAT on a reduced SAT problem and condition symbol-order and value-order branching based on cumulative WalkSAT statistics. As an added bonus, if WalkSAT happens to solve the problem, we can return immediately. Notice that the local search is not run at each node in the DPLL search tree. Instead, WalkSAT is run with some conditional probability factor, allowing us to avoid the performance over head of WalkSAT being run for every node in the DPLL search tree. Ideally this hybrid algorithm will have the completeness of DPLL and the optimized runtime of WalkSAT.

Application of Value-Ordering Heuristic

Of our available heuristics, value-ordering based on accrued average-value WalkSAT statistics seems the most promising. Utilizing the framework above, we run WalkSAT and adjust the order of DOMAIN-VALUES for each unassigned symbol in the DPLL problem space. In theory, DPLL will branch more often than not on correct symbol assignments, speeding up the search time to fi nd the average solution.

As fi gure 12 shows, the theory does not hold true in practice. Though the hybrid solver performs on-par with DPLL and WalkSAT in the easy symbol/clause region, the runtime is drastically worse as we pass through the 4.3 region. In short, the hybrid approach offers no real speed benefi ts over WalkSAT in the satisfi able region and degrades in the unsatisfi able region.

Conclusion

We have explored a number of different avenues involving applying WalkSAT to generate a heuristic for DPLL. Our initial focus was on the backbone set, but our results showed that the backbone set tends towards zero as problems approach the 4.3 clause/symbol ratio. Intuitively, this relation makes sense. For SAT problems with smaller clause/symbol ratios, the symbols are less-constrained, allowing an increased number of solutions and increasing the size of the potential backbone set. Regardless, identifying the backbone set for randomly-generated SAT problems does not appear to be worthwhile.

Further exploration focused on informing symbol and value ordering in DPLL using WalkSAT cumulative statistics. Our experimentation showed that symbol-order does play a key factor in DPLL runtime. We were not able to establish a linkage between flip counts in WalkSAT and symbol ordering in DPLL to consistently decrease the runtime of DPLL. Intuitively, we know that higher flip counts suggest that the given symbols are highly constrained. We hypothesized that branching on these symbols might allow DPLL to discover and prune inconsistent assignments earlier in the search tree. However, it seems that this rough-grained approach to symbol ordering is not enough to produce a reduction in runtime.

Our exploration of value ordering appeared more promising. Our results showed that symbols have increasingly consistent assignments under WalkSAT as problem clause/symbol ratios approach 4.3. We hoped this might tell us something about assignments of a possible solution, though our results indicate such statistics were only marginally better than average at predicting a problem solution. Intuitively, we can explain the increasingly consistent assignments as a function of the level of constraint in a problem. We hypothesize that for highly constrained problems, WalkSAT tends to get stuck in local maxima, flipping just a few values while the remaining symbols remain largely untouched.

Lacking strong evidence of a worthwhile heuristic, we implemented a hybrid DPLL-WalkSAT algorithm that, unfortunately, did not perform well. For the most part, our hybrid algorithm performed on par with WalkSAT for lightly constrained problems, as our hybrid algorithm effectively reduced to WalkSAT in that region. As problems became more constrained, our internal WalkSAT had less luck fi nding solutions straight off, turning over execution to our informed value-ordering heuristic. Unfortunately, this heuristic does not seem to produce valued results.

It is interesting to note the parallels between our approach and another combining DPLL and WalkSAT (Mazure, Sas, & Grgoire 1998). Their approach combined DPLL and WalkSAT in much the same way, except that they used WalkSAT to inform symbol ordering in DPLL by tracking the symbol with the highest flip count. Their increased performance seems contradictory to our results demonstrating that flip count does not reliably inform symbol ordering. However, closer examination of their approach reveals the use of Tabu stochastic search, which keeps a history of recently flipped symbols which are not immediately considered for subsequent flipping. Such an implementation avoids repeated flip-flops in local maxima, which might inflate and distort flip counts in regular WalkSAT. Unfortunately, we lacked the time to verify the approach of Mazure, Sas, & Grgoire.

Contributions

1. **Brian Ferris**: Brian wrote most of the DPLL-related code, as well as the hybrid implementation. Most the experimentation regarding heuristics using WalkSAT and DPLL were implemented and written up by him.
2. **Jon Froehlich**: Jon wrote most of the WalkSAT-related code, as well as all of the result aggregation framework. Jon did most of the direct comparison between WalkSAT and DPLL and the accompanying write up.

External Code

All code was written from scratch for this project. The text was consulted for the basic implementation of DPLL and WalkSAT.

Acknowledgements

We wish to acknowledge the contributions of Henry Kautz of the University of Washington in our efforts. A conversation with him regarding the application of DPLL and WalkSAT together lead to the idea of monitoring statistics of WalkSAT for informing the DPLL search process.

References

Cook, S. A., and Mitchell, D. G. 1997. Finding hard instances of the satisfi ability problem: A survey. In Du; Gu; and Pardalos., eds., *Satisfiability Problem: Theory and Applications*, volume 35 of *Dimacs Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society. 1–17.

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

Habet, D.; Li, C.; Devendeville, L.; and Vasquez, M. 2002. A hybrid approach for sat. In *In Proceedings of the Eigth International Conferences on Principles and Practice of Constraint Programming*, 172–184.

Li, C.-M., and Anbulagan. 1997. Heuristics based on unit propagation for satisfi ability problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, 366–371.

Lynce, I., and Silva, J. M. 2003. An overview of backtrack search satisfi ability algorithms. volume 37, 307–326.

Mazure, B.; Sas, L.; and Grgoire, E. 1997. Tabu search for sat. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, 281–285.

Mazure, B.; Sas, L.; and Grgoire, E. 1998. Boosting complete techniques thanks to local search methods. *Annals of Mathematics and Artificial Intelligence* 22:319–331.

Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an Effi cient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*.

O'Donnel, T. 2002. Determining satisfi ability of np-complete 3-sat logic problems solely from the overlap of clauses.

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. 1994. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*, 337–343.

Selman, B.; Kautz, H. A.; and Cohen, B. 1996. Local search strategies for satisfi ability testing. In Johnson, D., and Trick, M., eds., *Second DIMACS implementation challenge : cliques, coloring and satisfiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society. 521–532.

Selman, B.; Kautz, H. A.; and McAllester, D. A. 1997. Ten challenges in propositional reasoning and search. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, 50–54.

Slaney, J., and Walsh, T. 2001. Backbones in optimization and approximization. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 254–259.