1 Introduction

Propositional satisfiability, SAT, is the very essence of NP-hard problems, it was the first NP-hard problem discovered and all subsequent NP-hard problems have been proved so by showing they are at least as hard as SAT (Mitchell et al. 1992). Given a formula of propositional variables, SAT is the task of determining whether there exists a setting for the variables such that the formula containing those propositional variables evaluates true. A naive approach to solving this problem is simply to enter a generate-test loop, first generate a possible solution, test whether it is a solution, if not generate the next possible solution, and so on until all possible solutions have failed, or until a correct solution is found. However, for a formula with 300 variables, a computer that can generate and test $2^{306}$ solutions per millisecond, a very powerful computer indeed, and an unsatisfiable formula, so every possible solution must be tested, this task would take approximately 256 million years. Far longer than the hardware of a computer will function will any kind of reliability.

There are several more refined approaches to solving this problem, with far more reasonable run times. There are two classes of SAT solvers, complete and incomplete. Complete SAT solvers are guaranteed to find the correct answer, either unsatisfiability or a set of variable assignments that satisfies the formula, the naive approach from above is an example of a complete algorithm. Incomplete SAT solvers will either find a satisfying assignment or it will be unable to determine whether the formula is satisfiable. Although incomplete algorithms are not guaranteed to return a useful answer they have proved to perform quicker than their complete counterparts (Hirsch et al. 2001).

Two incomplete algorithms have been very effective in solving SAT problems. The first, GSAT, randomly selects a start state, then greedily chooses to invert a variable value based on how many clauses the resulting state would satisfy. GSAT continues flipping variables until it finds a solution or until it reaches the maximum allowed flips, MAX_FLIPS. If GSAT reaches MAX_FLIPS without finding a solution it restarts with a new randomly assigned state, GSAT continues restarting until it reaches MAX_RESTARTS (Selman et al. 1992). The second algorithm, WSAT, is a variation of GSAT; at every flip iteration WSAT randomly chooses to make a GSAT move, or to randomly select an unsatisfied clause and satisfy it, based on a probability of 0.5. The selected clause is satisfied by randomly selecting a variable within it and flipping its value. Like GSAT, WSAT has a limited number of flips to find a solution before it restarts with a random state. Although the algorithm allows other probabilities, WSAT performs best when it has 50% chance of performing a GSAT move, even outperforming GSAT (Selman et al. 1994). Limiting the search depth and randomly assigning variables at restart is collectively known as rapid random restart. It is restarting that prevents these algorithms from becoming stuck on a bad search path.

However, it seems that restarting in a random position results in lost knowledge from the previous search paths. The random restart state may in fact be a node from the previous path, resulting in simply reiterating over much of the same path. Similarly, the random restart states may cluster in a particular part of the search space, leaving many areas of assignment completely unsearched. There may be a particular variable or set of variables which continually remain unsatisfied at the end of a search, it might be advantageous to satisfy that set of variables first before restarting. This paper shows that a heuristic approach to selecting a restart state performs better than choosing a random restart state. The scope of this analysis is over restart algorithms, not the mechanism for creating a search path once a restart state has been selected.

Several heuristics were implemented and tested during the preparation for this paper, the ones presented here, while not the complete set, represent a good cross section of the those tested. Close Continue Restart allows a search along a promising path to continue for another full set of flips if the current state is within a given threshold of satisfied clauses out of total clauses. Permuted Restart creates several permutations based on the last state from a full set of flips, and it chooses the permutation with the most satisfied clauses as the new restart state. Solved Restart creates a sub-formula from the unsatisfied clauses and returns a satisfying assignment over the variables in the sub-formula as the new restart value. Each of these algorithms attempts to address one of the problems with Random Restart.
2 Heuristic Restart Algorithms

2.1 Close Continue Restart

Randomly choosing a restart state discards all information gathered by the search up to the point of restarting; however, because WSAT is restricted by a limited number of flips, this may be discarding a viable state; a state which is very close to being solved, given more flips this search may result in a solution. However, with Random Restart it is impossible to dynamically allow a good search to continue, the search space to another, alternating at every restart, obvious that Inverted simply switched from one half of already covered. However, during test trials it became apparent that the next search as far as possible from the space already covered. However, during test trials it became obvious that Inverted simply switched from one half of the search space to another, alternating at every restart, leaving a large track of search space completely neglected. To combat this problem, the Inverted Restart returns a random restart state with probability $p$, and an inverted restart state with a probability $1 - p$.

This allowed a search to move throughout the search space while still having the benefit of searching exact opposite positions of the space, yielding results on average slightly better than Random Restart. However, further modifications to Inverted Restart created an algorithm that significantly outperforms Random Restart, Permuted Restart. The current state at restart time is broken into $x$ equal sections, each section is then inverted, and $2^x$ permutations are created from the inverted and non-inverted sections, including the original state and the completely inverted state. The original state permutation is discarded and the remaining permutations are evaluated with respect to the formula, the permuted state with the largest number of satisfied clauses is chosen as the new restart state, and the remaining permutations are discarded. Initially all the unchosen permuted states were stored in a list and the next set of permutations was added to it, the next restart state was the best scoring state out of all of the current permutations plus all of the previously unchosen permutations. However, the list of unchosen permutations soon grew to an unmanageable size and run times suffered. Attempting to scale this implementation by using the random restart probability of $p$ did not produce better results, and in fact produced worse results.

2.2 Permuted Restart

Intuitively it would seem that Random Restart would cover all sections of the search space evenly, however, there is a possibility of clustering in one region while neglecting another. This Random Restart algorithm is subject to the benefits and drawbacks of the particular pseudo-random number generator it uses. The first attempt to address this problem was Inverted Restart, inventing the last state of the current search space was switched from one half of the search space to another, alternating at every restart, leaving a large track of search space completely neglected. To combat this problem, the Inverted Restart returns a random restart state with probability $p$, and an inverted restart state with a probability $1 - p$.

However, there is a possibility of clustering in one equal sections, each section is then inverted, and $2^x$ permutations are created from the inverted and non-inverted sections, including the original state and the completely inverted state. The original state permutation is discarded and the remaining permutations are evaluated with respect to the formula, the permuted state with the largest number of satisfied clauses is chosen as the new restart state, and the remaining permutations are discarded. Initially all the unchosen permuted states were stored in a list and the next set of permutations was added to it, the next restart state was the best scoring state out of all of the current permutations plus all of the previously unchosen permutations. However, the list of unchosen permutations soon grew to an unmanageable size and run times suffered. Attempting to scale this implementation by using the random restart probability of $p$ did not produce better results, and in fact produced worse results.

2.3 Solved Restart

Addressing another problem with Random Restart, it may be the case that a large portion of the formula is easily satisfied, however, there is a small set of clauses which are repeatedly unsatisfied at every restart point. Random Restart ignores the possibility that these clauses need special tuning in order to find satisfiability; Solved Restart is an attempt address this problem. When a restart point is reached, a subset of the formula is created using only those clauses that are unsatisfied in the current state. This sub-formula is easily satisfied, however, there is a small set of clauses which are repeatedly unsatisfied at every restart point. Random Restart ignores the possibility that these clauses need special tuning in order to find satisfiability; Solved Restart is an attempt address this problem. When a restart point is reached, a subset of the formula is created using only those clauses that are unsatisfied in the current state. This sub-formula is given to a standard WSAT algorithm which returns a solved sub-formula for the sub-formula, the variables in the original state are set to match those from the sub-state and all variables not in the sub-state are randomly assigned values. If WSAT cannot find a solution for the sub-formula, a random state is generated as the next restart state. However, in practice this rarely occurs, primarily because if a variable, $a$, occurs in the set of unsatisfied clauses its negation, $\neg a$, will not occur, because either the variable or its negation must be true. Solving a formula containing only pure literals is trivial, making a random restart unlikely. In an attempt to make this algorithm more robust, and the sub-formula more complex, a trial was conducted in which every clause containing a variable, even of opposite polarity, from the set of unsatisfied clauses was included in the sub-formula. WSAT still solved these more complex sub-formulas a majority of the time because in practice it was still a relatively small sub-formula over a small number of variables. However, in initial tests this trial Solved Restart performed worse.
than the simpler algorithm, and with much longer run times, so this extra complexity was discarded.

3 Randomly Generated Test Formulas

Trivial test cases lead to trivial or skewed results, in an effort to prevent this, the experiments run were over hard test cases. The heuristic restart algorithms were tested against randomly generated conjunctive normal formulas containing N clauses with three variables per clause, referred to as 3CNF, and conforming to the standards of difficulty detailed below. The CNF's were randomly generated to prevent infecting them with hidden structures often present in hand generated test cases. Further, analytical tests have shown that the hardest 3CNF formulas to solve are those that occur in the region of test space where a given formula has a probability of 0.5 of being satisfiable. This region occurs at a specific ratio of variables to clauses, namely 4.3 clauses to variables, all test cases conformed to this ratio (Mitchell et al. 1992). To further increase the hardness of the CNF's, as each variable is randomly chosen for a given clause, it is assigned negative polarity with a probability of 0.5. Initial testing showed that either increasing or decreasing the probability of negative polarity drastically decreased the difficulty of solving a randomly generated 3CNF. Further, no variable is allowed to appear in a clause more than once, and because these heuristic restarts will be tested using incomplete SAT solvers, the test CNF's are guaranteed to be satisfiable to prevent guessing if the solver failed because there was no solution. In all of the performance tests the same set of randomly generated CNF's was used to compare the performance of the different restart algorithms. A CNF was randomly generated and given to each of the SAT solvers to be solved, and a new CNF was generated for each increment of testing.

4 Test Specifications

The experiments below were conducted on Sun Ultra 10 300Mhz machines running Solaris 8. The WSAT algorithm and the restart algorithms were developed in Java 1.4.1 using the Eclipse development environment, and the tests were run by executing Java jars. The experiments were run in 50 increment iterations, an increment corresponding to a single 3CNF given to each algorithm. Initial tests compared the heuristic restart algorithms using both GSAT and WSAT to perform the search portion of the SAT solver, however, regardless of the restart algorithm WSAT outperformed GSAT. Therefore, in the following tests only WSAT was used for the search portion, with a MAX_FLIPS of 300 and a MAX_RESTART of 40. Unless otherwise stated Permuted Restart used an x value of 8, creating 255 possible permutations.

5 Test Results

Figure 1 summarizes the results of comparing Random Restart, Close Continue, Solved, and Permuted; it represents the fraction of 3CNF's solved over several iterations, for 3CNF's of 100, 200, and 300 variables. While all of the algorithms performed well at 100 variables, each solving at least 90% of the 3CNF's presented. Random Restart and Solved Restart's performance dropped dramatically when increased beyond 100 variables, eventually falling below 50% at 200 variables, and to 0% at 300 variables. Close Continue performed better than Random and Solved at all levels, but it still fell below 50% at 300 variables. Permuted outperformed all of the restart algorithms at all levels, solving over half of the test cases at 300 variables. Run times for these four algorithms were on the order of seconds, and time differences between the algorithms did not vary by
more than a power of 2. However, run times can be deceiving, revealing more about implementation inefficiencies than algorithm inefficiencies. To prevent this implementational bias Figure 2 displays the average number of restarts, instead of the average search time, per 3CNF's needed to for each algorithm to find a solution. From these test results it appears that choosing a restart state from a wide area of the graph offers the more benefit than solving recurring unsatisfied clauses, or allowing a more dynamic search depth. It is not surprising that Random Restart performed badly, however, it is surprising how badly Solved performed, only slightly better than Random. Perhaps recurring unsatisfied clauses are not common enough to merit a restart algorithm based on them, or perhaps solving them in isolation is too trivial to effect the outcome of the overall search algorithm. The success of Close Continue may point to the need for greater initial setting of MAX_FLIPS, however, it may also highlight the need to restart early for search paths that are doing badly. The remarkable performance of Permuted may illustrate the need for a wide spread of possible restart point from throughout the search space. However, it may merely be a side effect of selecting any $n$ possible states and choosing the best among them for a restart state.

6 Multi-Random

If Permuted performs better simply because it creates a large selection of restart states to choose from, then any algorithm that chooses a restart state from a large selection should perform equally well. Multi-Random Restart creates a large set of randomly generated restart states at every restart point and selects the state with the most satisfied clauses as the restart state, the unchosen states are discarded.

However, testing showed that Multi-Random solves only 21% of the test cases at 200 variables, only 6% better than Random, and it does not solve any 3CNF's at 300 variables, see Figure 3. While Permuted Restart solves over 80% of test cases at 200 variables, and over half of the 3CNF's at 300 variables. Clearly, it is not the amount of choices for a restart state that makes Permuted perform well, but the evenly distributed variety of choices covering several equally distanced parts of the search space that explains Permuted's success.

7 Number of Sections

If Permuted is successful because of its widely distributed possible restart choices, it may be that more sections produce better results. Clearly with enough sections, $N$ sections and $N$ variables, Permuted simply degrades to our naive complete SAT solver, this means finding the restart state would also find every possible solution to the formula. However, with too few sections, namely one, Permuted is simply Inverted Restart, which performed only slightly better than Random. Further, a larger number of sections means drastically longer run times because increasing the number of sections by one increases the number permuted states generated by a power of two. In addition, since Permuted performs a full permutation over all possible inversion of its sections, running the algorithm with a given number of sections yields a subset of possible states from the same algorithm with twice as many sections. More formally, let $P$ be the set of permuted states generated from current state $S$, with $x$ number of sections. Then let $Q$ be the set of permuted states generated from the same state $S$, with $2x$ number of sections, then $P \subseteq Q$. Some balance must be struck between efficiency and accuracy, at some point the amount of accuracy gained by increasing the number of sections will not outweigh the decrease in efficiency. Figure 4 illustrates Permuted with the section variable set to 4, 6, 8, and 10 running against 200, 300, and 400 variables. While 10 sections...
does not appear to have an advantage over 8 sections at 200 variables, in fact it did slightly worse, as the number of variables increase so does the difference in performance between 8 and 10 sections. Perhaps the number of sections should be a function over the number of variables. However, as Figure 5 illustrates this will mean longer run times for formulas with more variables, but longer run times are generally expected for formulas with more variables. As the number of MAX_FLIPS and MAX_RESTARTS must scale with the problem, it may be that number of sections will need to scale as well.

8 Permuted Failing Point

Random and Solved completely failed at 300 variables solving none of the 3CNF's, it would be advantageous to know the failing point for Permuted. Therefore, Permuted was run with 8 sections over 500, 600, and 700 variables, see Figure 6. Permuted still solved close to 20% of the test cases at 500 variables, better than Random at 200 variables, and 10% of test cases at 600 variables, finally failing at 700 variables with less than one percent of 3CNF's solved. While increasing MAX_FLIPS, MAX_RESTARTS, and the number of sections would certainly allow Permuted to perform better it will also significantly increase its run time. It should be noted that Permuted only attempted each 700 variable 3CNF for an average of 5 minutes, however, even increasing this time to 10 minutes would produce a testing iteration, 50 increments, of over 8 hours. While an 8 hour runtime is generally feasible, current computer resources and end of the semester time constraints make increasing these parameters unmanageable.

9 Conclusion

Heuristic Restart algorithms can add significant performance to SAT solvers over 3-SAT problems with a clause to variable ratio of 4.3. Close Continue Restart performed better than Random Restart with large and small numbers of variables, and with very little implementational overhead. However, it can not outperform Permuted Restart, which performs significantly better than either Random Restart or the other heuristic restart algorithms over a small number of variables and over a large number of variables. Solved Restart adds significant implementational overhead with very little benefit over Random Restart for small numbers of variables and no benefit for large number of variables. Permuted Restarts success cannot be attributed to its ability to select from a large set of restart states, but rather the even distribution over the search space of the set of possible restart states. Increasing the number of sections used by Permuted Restart provides improved performance for formulas with large numbers of variables, but little performance increase for formulas with small numbers of variables implying that number of sections should be determined by a function over the number of variables. With MAX_FLIPS at 300, MAX_RESTARTS at 40, and number of sections at 8, Permuted Restart fails to solve any 3CNF's at around 700 variables, however, it is most likely limited by its parameters rather than by the algorithm itself. Further analysis might attempt Permuted Restart at 700 variables with less restrictive parameters and a less restrictive time constraint. All of the heuristic restart algorithms presented here start with a randomly generated beginning state, however, future research might find a way to apply Permuted Restart to finding an optimal start state. Some research might explore the performance of a Permuted Restart like algorithm that does not require rigidly separated equal length sections, but sections that may overlap or change size.
References