Variations on ND

Andrew E. Shaffer

February 7, 2002

0.1 Introduction

In this paper, I describe NDS, a performance-enhancing modification to the ND meta-reasoning procedure for solving propositional logic satisfiability problems. Both ND and NDS are parameterized by a reasoning procedure P, a procedure D that reduces a problem instance Ω to a related "subproblem," and an integer n. Neither ND nor NDS contain their own propositional reasoning logic, but instead use the supplied P to generate models of $D(\Omega)$ and use them to guide the search for a model of Ω.

One immediate difference that emerged during tests of ND against SATO was that ND's performance graph doesn't have the shape typical of problems such as CPC or 3SAT, with a running time maximum inside the phase transition region and easier instances in the over- and under-constrained regions. This is because ND must first find every model of $D(\Omega)$ (the "subsolutions"), and as problems move into the underconstrained region, the number of subsolutions increases exponentially. Since this exponential behavior causes ND to take longer than SATO in the underconstrained region, as well as use large amounts of memory when there are many subsolutions, a natural next step in developing the ND method is to attempt to free it from this restriction.

Rather than solve for all subsolutions and then process them in groups as with ND, NDS processes subsolutions in groups as they are generated. After P has generated n solutions to $D(\Omega)$, NDS processes them immediately and returns a solution to Ω if one is found. If not, NDS continues generating solutions to $D(\Omega)$ in this manner until all subsolutions are exhausted, and if by that point a solution to Ω has not been found, it returns unsatisfiable. Only in the unsatisfiable cases (or cases where the only subsolution that extends to a solution of Ω is in the last block of subsolutions) does NDS need to generate every subsolution. Even if it does need to generate all subsolutions, there is the added benefit that once a block of subsolutions is used, it doesn't need to remain in memory.

I also experimented with using WalkSat, a local search procedure, for P in ND and NDS. Since WalkSat is a local search algorithm, it is not guaranteed to find every possible subsolution. As such, there are issues with correctly identifying satisfiable instances. Also, because there is no definite point at which to return unsatisfiable (such as when all the subsolutions are exhausted), it must be specified as additional parameter to P.

The procedures were tested on Open Crossword Puzzle Construction (Open CPC) problem instances using the decomposition method described in the previous paper.

0.2 The ND and NDS Algorithms

The ND procedure is as follows[1]:

[1]From the original ND paper, with minor changes.

Input: a problem instance Ω, a propositional reasoning procedure P, a decomposition method D, and a number $n \in (1, 100]$.
Output: a model of Ω or "no"

---

1. Compute $D(\Omega)$.
2. Generate $ss(D(\Omega))$, the set of solutions to $D(\Omega)$. Let m denote the size of $ss(D(\Omega))$ at this point.
3. **while** $ss(D(\Omega))$ is nonempty:
   (a) Remove models $s_0, \ldots, s_k$ from $ss(D(\Omega))$, where $k = min(|ss(D(\Omega))|, mn/100)$.
   (b) Let $\Omega^- = \Omega \cup mutex(s_0, \ldots, s_k)$.
   (c) If Ω^- is solvable using P, stop and return the model.
4. Stop and return "no."

Procedure ND(Ω,P,D,n)

The NDS procedure is similar to the ND procedure, but most notably it eliminates the second step.

Input: a problem instance Ω, a propositional reasoning procedure P, a decomposition method D, and a number $n \in (1, 100]$.
Output: a model of Ω or "no"

---

1. Compute $D(\Omega)$.
2. **while** P can generate more models of $ss(D(\Omega))$:
   (a) Generate the next n models of $ss(D(\Omega))$, or if there are fewer than n models remaining, generate all remaining models. Call them $s_0, \ldots, s_k$.
   (b) Let $\Omega^- = \Omega \cup mutex(s_0, \ldots, s_k)$.
   (c) If Ω^- is solvable using P, stop and return the model.
3. Stop and return "no."

Procedure NDS(Ω,P,D,n)

The *mutex* function used in ND and NDS returns a set of clauses requiring that all the atoms in at least one of the solutions $s_0, \ldots, s_k$ be true. (The clauses of

Ω will likely assert that only one subsolution holds.) In this way, the procedures attempt to "extend" a subsolution into a supersolution by finding assignments for the unassigned atoms, those that are in the superproblem but not in the subproblem. In addition, mutex asserts that one of $s_0(i), s_1(i), \ldots, s_k(i)$ is true for each $i \in [1, |s|]$, where $s(i)$ is the ith true atom in s. This second set of clauses may not always be possible or appropriate for all problem domains, such as when each subsolution can have a different number of true atoms. For Open CPC, it asserts that in the original puzzle, each square that exists in the subpuzzle must contain a letter from a subsolution. IE: if all the subsolutions have either an 'a' or 'b' in the upper-left hand square, then $('a', 1, 1) \vee ('b', 1, 1)$ would be asserted. Also, note that n is interpreted differently by ND and NDS. In ND, n indicates what percentage of the models of $D(\Omega)$ to use per iteration. NDS, as it doesn't know how many models there eventually will be, interprets n to be the number of subsolutions to generate and use per iteration. Also, note that it is necessary for the atoms in the subproblem to have the same "name" as the corresponding atoms in the superproblem. In implementing the procedure, for convenience, one can rename the atoms in the subsolutions to appropriate atoms in the superproblem.

0.3 Finding an optimum function for NDS

Before comparing ND or NDS to other procedures, it is necessary to find a value for the parameter n. For Open CPC, n is determined as a function of word set size. The function was found by testing a range of n for many problem instances over the range of word set sizes, and fitting a curve to the highest performing n for a given word set size. Figures 1 and 2 show the results of these experiments with the minima and the fit curve. Although a function for ND was already available, it wasn't determined with great precision. Also, it has yet to be shown that the function isn't hardware independent to some degree, and previous experiments were performed on a different hardware configuration.

0.4 Experiments and Observations

[2] Using the fitted functions to determine n, the procedures were tested on Open CPC instances with $|\Sigma| = 10$ (the size of the word set alphabet) and word set sizes varying from 100 to 160, in order to capture the phase transition. SATO 3.2 was used with DATA=3 in all cases, both for generating the subsolutions and solving Ω^-. The results of these experiments are in figures 3 and 4.

As was intended, NDS avoids the exponential time curve shape of ND. It performs similarly to ND in the overconstrained region, but at around 50% satisfiability, NDS' running time curve bends back down like SATO's. NDS' "Generation Time" curve decreases as word set size increases, likely because it is easier to find models of

[2]The results for NDS are incorrect, I neglected to take into account the encoding time for NDS but left it intact for ND and SATO. This shouldn't be a significant factor in the results, and should leave the fitted equations intact. Preliminary experiments show that the effect should be bounded by about half a second. This needs to be fixed.

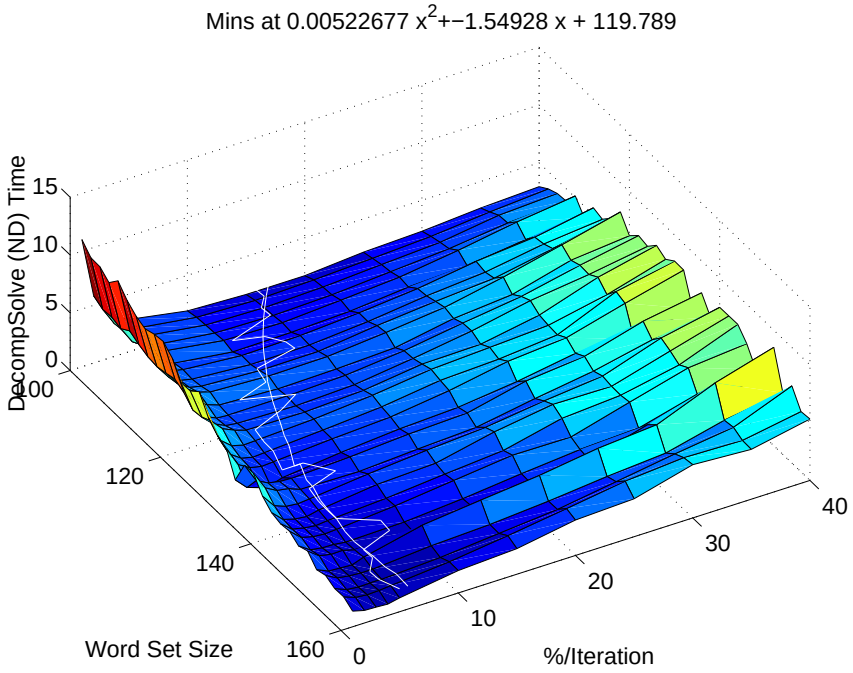

Figure 1: DecompSolve (ND) Solving Time

$D(\Omega)$ in the underconstrained region and because it is more likely that any given subsolution will extend to a supersolution. ND's overall time and "Generation Time" curves increase exponentially as was mentioned earlier. The satisfiabilities of ND and NDS, as depicted in in figure 4, are equal to each other and almost exactly equal to SATO's.

There is also an interesting point to make about the "Subsolution Processing Time" curves from figure 3, (the total time that is spent on solving Ω^- each iteration), and the subsolution information in figure 5. Even though NDS needs to processes fewer subsolutions on average, it spends more time doing so. (That NDS processes fewer subsolutions is due to how many subsolutions ND and NDS process at a time, rather than any ability NDS might have at picking "better" subsolutions.) One explanation is that prior to processing by ND, $ss(D(\Omega))$ is sorted lexicographically. As a result, the first few atoms of any $s_0, \ldots, s_k$ set are likely to be the same, and will be inserted as unit clauses by *mutex*, resulting in less branching and better performance. NDS, as it doesn't have access to every subsolution, cannot accomplish this level of "cohesion," or similarity between mutexed solutions. Even so, by the nature of backtrack search, there will still be similarities in adjacent subsolutions, but because of the way that *mutex* is implemented, similarity in the middle of a group of clauses may be less useful than similarity at the beginning. Finally, an informal experiment performed during debugging showed that subsolutions could be processed more quickly when taken from a sorted list than from an unsorted list. Therefore, ND has an advantage over NDS in this area, but the trade off is that it must spend much more time generating the subsolutions to begin with.

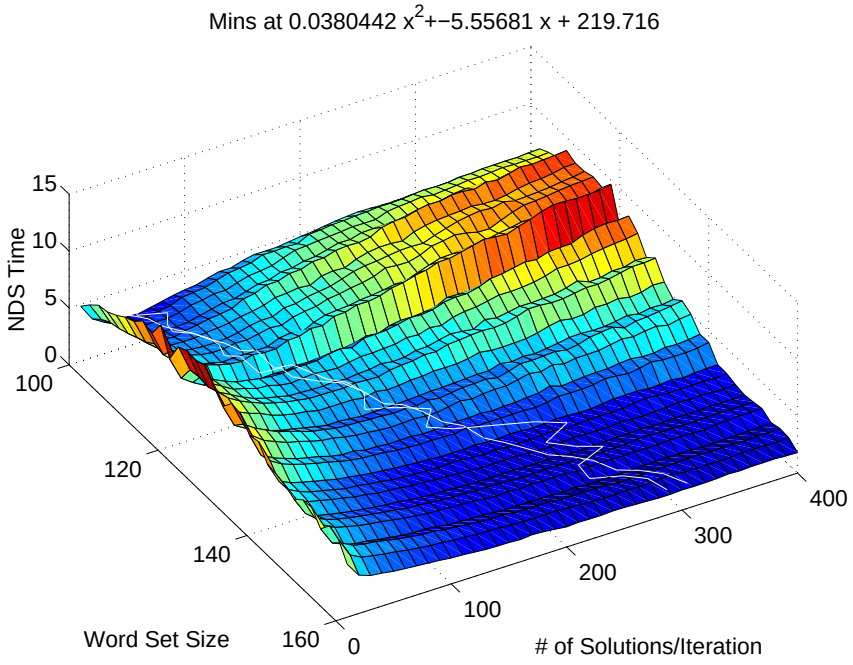

Figure 2: NDS Solving Time

0.5 WalkSat

I also measured a version of NDS that uses WalkSat to generate subsolutions. In addition to the parameters required by ND and NDS, it also requires a second parameter (really a parameter to P) that indicates how many subsolutions to try before deciding that the problem is unsatisfiable. Preliminary experiments[3] showed that NDS/WalkSat underperformed SATO or NDS/SATO, and that much of this time was spent processing the subsolutions. This is potentially due to the "cohesion" issue mentioned previously, as a local search is less likely than a backtracking search to have a lot of similiarity between consecutively found solutions.

Because ND makes cohesion less relevant, I then tried using WalkSat as P in ND. To determine how many subsolutions to generate in step 2 of ND, I used the average number of subsolutions that ND processed (as gathered in previous experiments), as shown in figure 5. I then determined n, in this case the percentage of $ss(\Omega)$ to use per iteration, by curve-fitting optimum n values. The performance results showed improvement, but many instances were not being correctly identified as satisfiable. I then tried increasing the number of solutions WalkSat would find, ranging from 100% to 200% of the original number. The results are in figures 6 and 7.

Although correctness was improved by increasing the number of subsolutions used, it came at the expense of time. Note that if using 100% of ND's subsolutions

[3]The preliminary NDS/WalkSat experiments were run before I had the idea to use the statistics for ND/SATO as a guideline of when to stop, and as such they aren't particularly reliable. I would like to repeat the tests when time permits, but for this paper I felt that ND would be more interesting. ND is also simpler, and as such is a better starting place. I have little experience with using local search methods and so this section should be interpreted as only a first step.

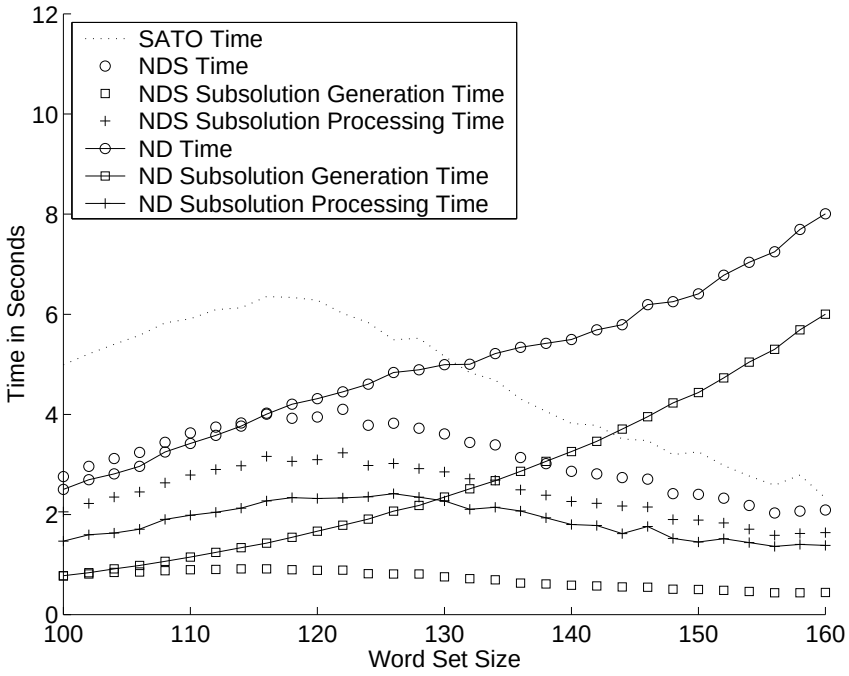

Figure 3: Time Graph

results in an Δ satisfiability discrepancy, 200% should result in $O(\Delta^2)$ discrepancy. ($\Delta < 1$.) Call the true percent satisfiable for a fixed a word set size S. Then ND/WalkSat will find $S - \Delta$ satisfiable when using 100% of ND's subsolutions. If one doubles the number of subsolutions used to 200%, it is equivalent to repeating the procedure again with 100% of ND's subsolutions on all the instances that are labeled unsatisfiable. On this second attempt, $S - \Delta$ of the instances incorrectly labeled unsatisfiable in the first trial will be labeled satisfiable. So if Δ instances are incorrectly labeled unsatisfiable the first time, and $S - \Delta$ of those are labeled satisfiable the second time, then satisfiability for 200% should equal $S-\Delta+\Delta(S-\Delta)$, and the error would then be $O(\Delta^2)$. A quick look at the satisfiability graphs shows that this seems to be the case. Also, although it seems that Δ is fixed for a given word set size and number of subsolutions, it may be possible to decrease it by removing duplicate subsolutions from those generated by WalkSat and replacing them with new ones. Although this is technically increasing the number of subsolutions that are generated, the previous way of increasing the number of subsolutions introduces more duplicates, whereas this method is guaranteed to add additional unique subsolutions. Thus, each subsolution will likely be more useful.

0.6 Future Work

- The idea of "cohesion" is still somewhat vague, and in order to determine whether or not it affects solving time, it would be useful to have a formal definition and a manner in which to measure it.

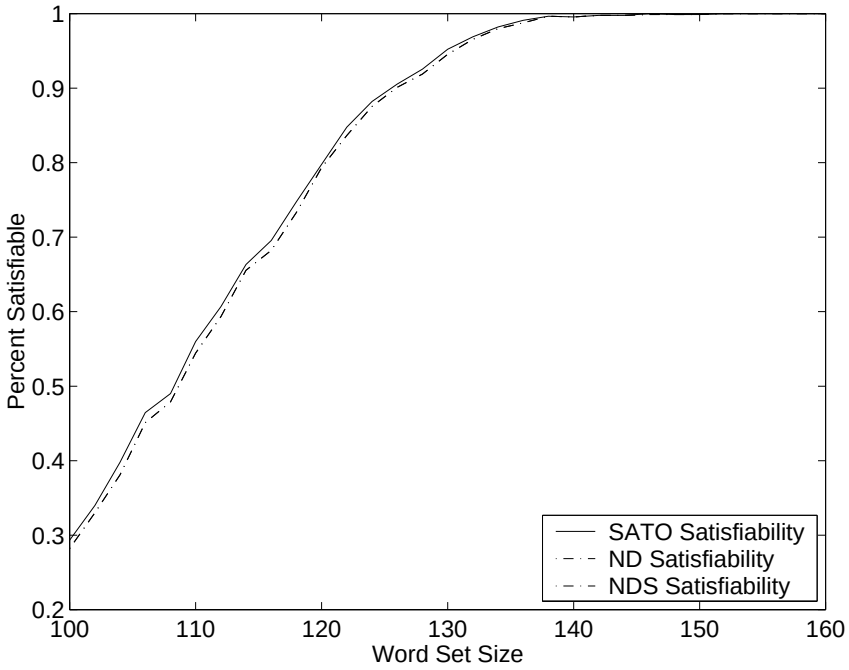

Figure 4: SAT Graph

- An observation of figure 3 shows that ND and NDS each allot their computing time differently. ND spends more time generating subsolutions and as a result needs to spend less time processing them. NDS spends less time generating subsolutions, but the resulting groups of subsolutions take more time to process. I would like to try incorporating the advantages of each extreme. The procedure would generate blocks of subsolutions, larger than blocks generated by NDS, but smaller than the set of all possible subsolutions. Those subsolutions could then be sorted as in ND, and then smaller blocks could be processed out of that larger block, taken off of the front. The processed subsolutions could then be replaced in the large block by a new set of subsolutions and the large block resorted or the large block could be completely exhausted and then replaced with another. This both provides a larger degree of cohesion than NDS and also eliminates step 2 of ND. Also, instead of simply sorting the subsolutions, for better cohesion the algorithm could try to find a "most cohesive subset", although finding a cohesive subset would likely be computationally difficult.

- SATO can be configured to use different methods when solving. These experiments were carried out using DATA=3, because DATA=2 has in the past not generated every subsolution. However, in quick experiments, DATA=2 seems to be significantly faster at solving Ω^- instances. It would be interesting to see how using DATA=2 would change the performance differences between the various procedures, as less time would likely be spent on processing the subsolutions, perhaps giving an advantage to both SATO and NDS.

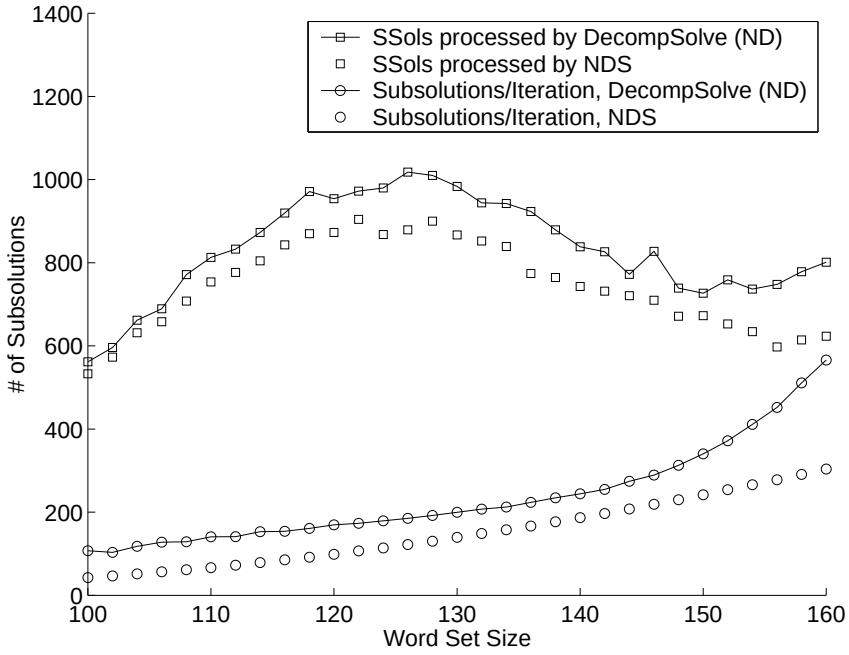

Figure 5: Subsolution Graphs

- Brief, initial tests on WalkSat and NDS/SATO showed them performing much worse than SATO or ND when the second set of *mutex* assertions were not included. This indicates that the second set of *mutex* assertions are important to keeping running times low. Unfortunately, they aren't always possible for all problem domains. As such, experiments on problem domains in which it isn't possible to generate these clauses would be interesting. In addition, experiments without the first set of *mutex* assertions, using only the second set, may give some insight. The benefit to NDS/WalkSat was greater than the benefit to NDS/SATO, and so I suspect that the level of cohesion in the subsolutions may be important in determining how helpful the second set of assertions is. Also, the second set of assertions is dependent on the set of subsolutions rather than the problem domain, and as such there may be a way to augment *mutex* to automatically determine whether or not to add them for a given set of subsolutions.

- One of the major difficulties in ND and NDS is determining a function for the parameter n. It would be useful if the procedures could somehow learn an optimal function for n without user interaction.

0.6.1 Future Investigations Using WalkSat

- It was mentioned previously that WalkSat may end up returning and hence processing the same subsolution more than once. It is also hypothetically possible that WalkSat returns subsolutions with nonuniform distribution, which causes problems if extendable subsolutions aren't returned with high proba-

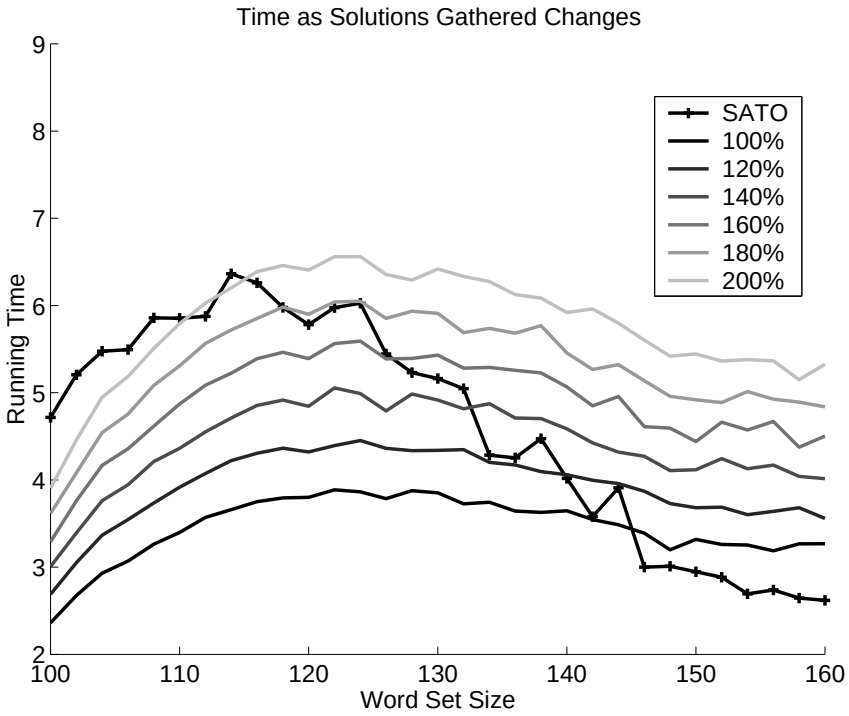

Figure 6: Solving Time for ND/WalkSat

bility relative to other subsolutions. This may be especially problematic when the problems are in the critically-constrained region and there are few subsolutions that extend to a supersolution, so that is it important a wide variety of subsolutions are found.

- In order to determine a point at which to return unsatisfiable, a procedure could use both WalkSat and SATO for P and use the subsolutions generated by both of them. Once SATO runs out of subsolutions, it is safe to return unsatisfiable, but during the run WalkSat may be able to generate subsolutions more quickly.

- The subsolution concept could also be integrated with local search algorithms. Instead of inserting clauses that assert the subsolutions, one could use subsolutions as the initial assignments for WalkSat, and randomly assign the rest of the atoms. They would have to be used one at a time, but since they aren't being strictly asserted, WalkSat may be able to "fix" any errors in the subsolution's atoms as well as in the unassigned atoms. It may be sufficient to find a "good" subsolution. If this was combined with using SATO for P, the procedure would have a good guess of when to stop, even if it still wasn't sure whether an unsatisfiable result was correct or not.

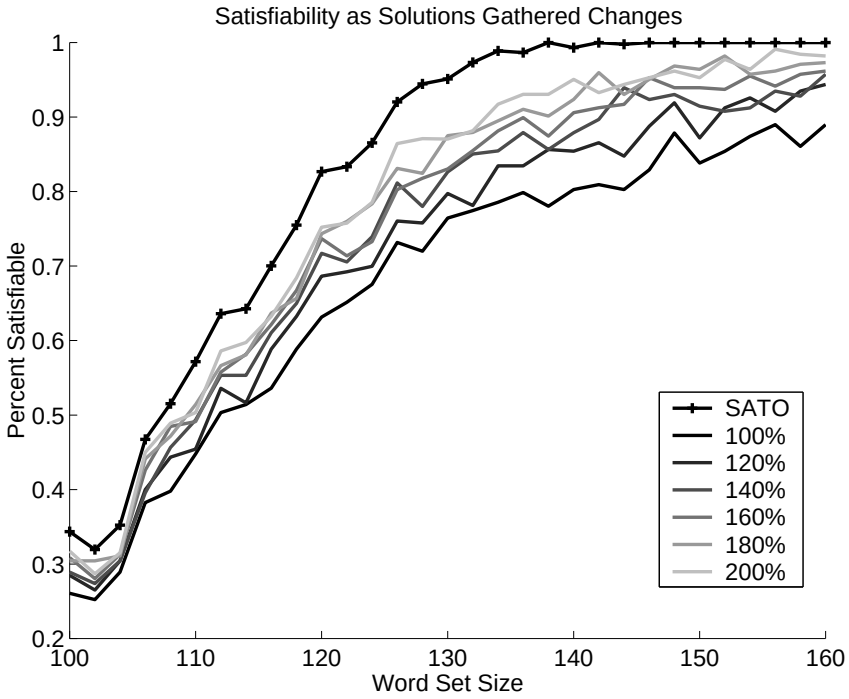

Figure 7: % Satisfiable for ND/WalkSat

0.7 Other Work

During the course of the project, I wrote some code that didn't end up getting used in the final experiments:

- Although it was not for this project specifically, at the beginning of the semester I ran experiments using ND to solve Crossword Puzzle Construction instances without the Duplicate Word Constraint, as an addendum to the original paper.
- I wrote new code for encoding Open CPC because WalkSat was not performing well. However, the problem was simply fixed and the new encoding software became unnecessary. I also implemented a binary encoding for graph coloring, which was not used because I decided to concentrate on CPC.
- Much of the code that I had previously (NDCSato, CSato, mutex, and the subsolution optimizer) had small bugs that came up during the course of using them for this project, and some of the code had to be rewritten from scratch in order to be applied to CPC (my original subsolution optimizer, described in the next section, was graph coloring specific.) I also did some code restructuring in order to more cleanly accommodate WalkSat and data processing with Matlab.

0.8 *Mutex* Optimizer, *Mutex* implementation

The *mutex* function used in NDS puts the subsolutions into a trie in order to generate the clauses that assert those subsolutions. However, the default ordering of the atoms in the subsolutions won't likely generate the smallest possible trie and hence might end up generating more clauses than necessary. By putting the most common atoms at the beginning, the trie can become more compact[4]. The algorithm essentially picks the most frequent atom, puts it at the beginning of all subsolutions that contain it (and its negation at the beginning of all subsolutions that don't), splits the subsolution set depending on whether or a subsolution contains the most frequent atom, and branches. Although the algorithm takes a lot of time, it is also possible to trade optimizing accuracy for time by cutting off the optimization at a certain depth. Even so, a small experiment that varied the optimization depth showed that the time penalty incurred from sorting the atoms was not worth any benefit in solving time. It may be possible to improve the performance of the optimizer by reducing the constant factors, since the current implementation focuses more on correctness than speed. Also, the second set of *mutex* assertions may reduce the benefit that such optimization may provide, since they allow P (for backtracking search, at least) to "start in the middle" by assigning values to any atom, whereas only including the first set of assertions may make it difficult to start from anywhere but the atoms at the top of the trie.

[4]The code to do this sorting is in the `Algorithms/GeneralSortTree.cc` file.