CS 227 Programming Assignment 1: Propositional Satisfiability

Todd Sullivan
todd.sullivan@cs.stanford.edu

Harry Robertson
harry.robertson@gmail.com

Pavani Vantimitta
pavani@stanford.edu

1 Introduction

For this assignment we implemented all of the base hill-climbing SAT algorithms (GSAT, HSAT, GSAT with random walk, WalkSAT), along with several variants: GSAT with random walk, HSAT with bounded memory (a tabu list), HSAT with random walk, HSAT without caching, and WalkSAT with memory. We implemented all algorithms in C++ (see submitted code), and ran them all on the provided set of SAT problems.

In this paper we start by briefly presenting the main algorithms used. We then explain the key optimizations we made in our implementation, and finally we present our results analysis and our conclusions. The actual results data are included in the annexes.

2 Algorithms

We have six variants of the SAT Algorithms that we have defined. Each of them differ in the implementation of the three main functions, namely, initial(), hillClimb() and pick(). Below is a brief description of the main differences in the implementations of each of them.

2.1 GSAT

a. initial() – GSAT starts off with a randomly generated assignment to its variables.

b. hillClimb() – This function returns the variables whose truth assignment when flipped increases the number of satisfied clauses the most. The number of satisfied clauses of each variable is called the score of that variable. This algorithm is called Greedy SAT because of this property.

c. pick() – Of the above generated list of variables, the pick() function randomly selects one of the variables. We then proceed by flipping its assignment and checking the satisfiability of the new assignment.

2.2 HSAT

Historical SAT – HSAT varies from GSAT only in the pick() function. It uses historical information to choose which variable to pick. It picks the variable that was flipped longest ago (in the current try). If two variables are offered which have never been flipped in this try, an arbitrary ordering is used to choose between them.

2.3 HSATBounded

a. pick() – We store the n last flipped variables (where n is the length of the memory) in a bounded-length tabu list. The algorithm randomly picks a variable in possibleFlips that is not in the tabu list. If no such variable exists, then the algorithm randomly picks one of the variables in possibleFlips.

2.4 HSATRW

HSAT with Random Walk – HSATRW again varies from HSAT and GSAT only in the pick function. It picks a variable occurring in some unsatisfied clause with probability p (which we call walkProbability). The algorithm follows the general HSAT pick function of picking the variable that was flipped longest ago with probability (1-p).

2.5 GSATRW

GSAT with Random Walk - GSATRW varies from GSAT only in the pick function. It picks a variable occurring in some unsatisfied clause with probability p (which we call walkProbability). It follows the general GSAT pick function of picking the variable that causes the most change in the number of satisfied clauses when flipped with probability (1-p).

2.6 WalkSAT

a. initial() – same as GSAT, picks an initial random assignment.

b. hillClimb() – First, randomly selects an unsatisfied clause. Then, with probability p, returns the variables within the selected clause that have the lowest break count (the number of clauses that become unsatisfied when a variable is flipped) With probability (1-p) it returns all of the variables in the selected unsatisfied clause.

c. pick() – this again is the same as GSAT (random picking).

2.7 WalkSATMemory

WalkSATMemory has the same initial() and hillClimb() functions as WalkSAT, but uses HSAT's pick function to select the variable in possibleFlips that was chosen longest ago.

3 Optimizations

Caching and other optimizations are essential for creating usable SAT solvers. We developed our solvers with efficiency in mind. Our optimizations include staying away from STL data structures, using several caching structures to minimize frequent computations and lookups for scores, unsatisfied clauses, and variable occurrence in unsatisfied clauses, and several memory allocation tweaks.

3.1 No STL Data Structures

The C++ Standard Template Library (STL) certainly contains many useful algorithms and data structures, but these niceties come at a cost. Many of the STL's data structures, such as the Vector, contain out-of-bounds checking and other features that eat up CPU cycles. We ran several simple tests, contained in the test.cpp file of our source, to show how using STL's Vector class is much slower than using our own simple data structure.

Our simple data structure called intArray contains two members, a pointer to an array of integers and an integer holding the length of the array. Before beginning any tests, we dynamically allocated memory for the intArray and used the Vector's reserve command to ensure that the Vector did not allocate space (to accommodate more data) during any of the tests. All of our timers calculated CPU time for the segments of code and not the real processing time. To ensure accurate time calculations, we ran the calculations for enough iterations so that the fastest of the data structures took at least 30 seconds to complete the test.

The first two tests involved simple reading and writing of data to the first element of each data structure using the [ ] operator. For the intArray, we tested overloading the [ ] operator on the intArray class (Op) and directly using [ ] on the array's pointer (Direct). For reading, the intArray's Direct access method was 11.052 times faster than the Vector while its Op method was 6.557 times faster than the Vector. For writing, the intArray's Direct method was 12.777 times faster than the Vector while its Op method was 12.445 times faster.

In addition to the simple reading and writing tests, we compared the two structure's performance on an integral part of GSAT's hillClimb function. GSAT's hillClimb function loops through all of the variables constructing a list of variables that will each, if flipped, give the greatest increase in the number of satisfied clauses (the score). This involves a for loop through all of the variables clearing the list if the current variable's score is greater than the current best score and adding the variable to the list if its score is greater than or equal to the current best score. In our test, we add 10 integer values to each structure's array, clear the array, and then repeat many times. We clear the intArray by setting its length equal to zero. We clear the Vector using its clear() function. The intArray as used in GSAT (with its Direct access method, not Op), is 4.541 times faster than the Vector.

3.2 Caching Structures and Algorithms

We use many caching structures to decrease the amount of redundant computations. Our caching structures are: unsatisfiedClauses, variablesToClauses, satisfiedLiteralCount, breakCounts, and makeCounts. All structures except variablesToClauses are only maintained if the current algorithm requires the structure for its hillClimb or pick functions.

As the name suggests, unsatisfiedClauses contains a set of clause IDs (an intArray) of all clauses that are not satisfied by the current assignment. variablesToClauses, an array of pairs of intArrays, stores sets of clauseIDs and maps each variable to the set of clauses that contain the normal version of the variable and the set of clauses that contain the negated version of the variable. satisfiedLiteralCounts is an intArray whose index (a clauseID) maps to the number of literals that are true within the given clause given the current assignment. breakCounts and makeCounts are both intArrays that map variables to their break count or make count respectively.

All of our solvers use either breakCounts, makeCounts, or both. For the solvers that use GSAT's hillClimb function, the score is computed by subtracting the variable's break count from its make count. The tracking of unsatisfiedClauses is only needed in WalkSAT and its variants.

3.2.1 Initialization

We initialize our caching structures in SATSearch::preprocessForCNF and SATSearch::-preprocessForAssignment. The initialization is fairly standard as we have to calculate all of the values directly and there are no fancy tricks for computing the values any faster than simply computing them. We increment a variable's break count once for every clause that it is in where the clause's satisfiedLiteralCounts value is one, and similarly for a variable's make count when the count is zero

3.2.2 Incremental Updates

When we flip a variable, we incrementally update all caching structures except variablesToClauses, which remains fixed for the entire search for a given CNF. We first flip the assignment of the chosen variable. We then identify a leavingClauseIDSet and enteringClauseIDSet. The leavingClauseIDSet contains all of the clauses that the variable is in where the variable's old assignment contributed to the clause's satisfiedLiteralCounts value. Likewise, enteringClauseIDSet contains all of the clauses that the variable is in where the variable's new value contributes to the clause's satisfiedLiteralCounts value. If the variable is being flipped from true to false, then leavingClauseIDSet is the set of clauses where the variable occurs as itself and enteringClauseIDSet is the set of clauses where the negation of the variable occurs. When the variable is flipped from false to true, the sets are the opposite.

After identifying leavingClauseIDSet and enteringClauseIDSet, we loop through all clauses in both sets. For each clause in leavingClauseIDSet, we decrement its satisfiedLiteralCounts value. If its new count is equal to one then that means that there is only one true literal remaining in the clause, so we increment that variable's break count. If its count is equal to zero then the clause is no longer satisfied, so we increment the make count of all variables in the clause, decrement the break count of the flipped variable, and add the clause to unsatisfiedClauses.

For each clause in enteringClauseIDSet, we increment its satisfiedLiteralCounts value. If the new count is equal to two then that means that the other satisfying literal (that is not currently being flipped) in the clause will no longer cause the clause to be unsatisfied if we flipped it, so we decrement its break count. If the count is equal to one then the clause is newly satisfied, so we decrement the make counts of all variables in the clause, increment the break count of the variable we are flipping, and remove the clause from unsatisfiedClauses. Figure 3.2.2 shows the gist of our incremental updates. It is important to note that while we mention updating all of the caching structures, the only structures that are actually updated are the ones that the current solver specifies that it needs in its constructor.

```
flipVariable( int variable )
   flip variable's assignment
   set leavingClauseIDSet and enteringClauseIDSet

   foreach( clause in leavingClauseIDSet )
     decrement satisfiedLiteralCounts[ clause ]
     if( count == 1 )
        increment break count of one true literal in clause
     else if( count == 0 )
        foreach( variable v in clause )
          decrement v's make count
        decrement variable's break count
        add clause to unsatisfiedClauses

   foreach( clause in enteringClauseIDSet )
     increment satisfiedLiteralCounts[ clause ]
     if( count == 2 )
        decrement break count of other true literal in clause
     else if( count == 1 )
        foreach( variable v in clause )
          increment v's make count
        increment variable's break count
        remove clause from unsatisfiedClauses
```

Figure 3.2.2: Incremental cache updates when flipping a variable

3.3 Memory Allocation and Quick Insertions

Memory allocation can be costly. To reduce the amount of memory allocation required during the search, we allocate all data structures before starting the search loop. This step is obviously required for the caching structures, but is less obvious for structures such as the list of possible flips returned by a solver's hillClimb function. We allocate an intArray for this list at the beginning of the loop, making its size the same as the number of variables in the CNF, which is the maximum amount of variables hillClimb can possibly return. Instead of using the intArray's length attribute to indicate the amount of space allocated in the array, we use it to indicate the amount of variables that are actually in the list. Thus whenever hillClimb is called we do not need to allocate memory for the list and instead set the length to zero and start filling the list as described in our GSAT::hillClimb test in Section 3.1. We also make every attempt to use pointers and not invoke a copy constructor anywhere in each solver's code.

4 Experimental Results

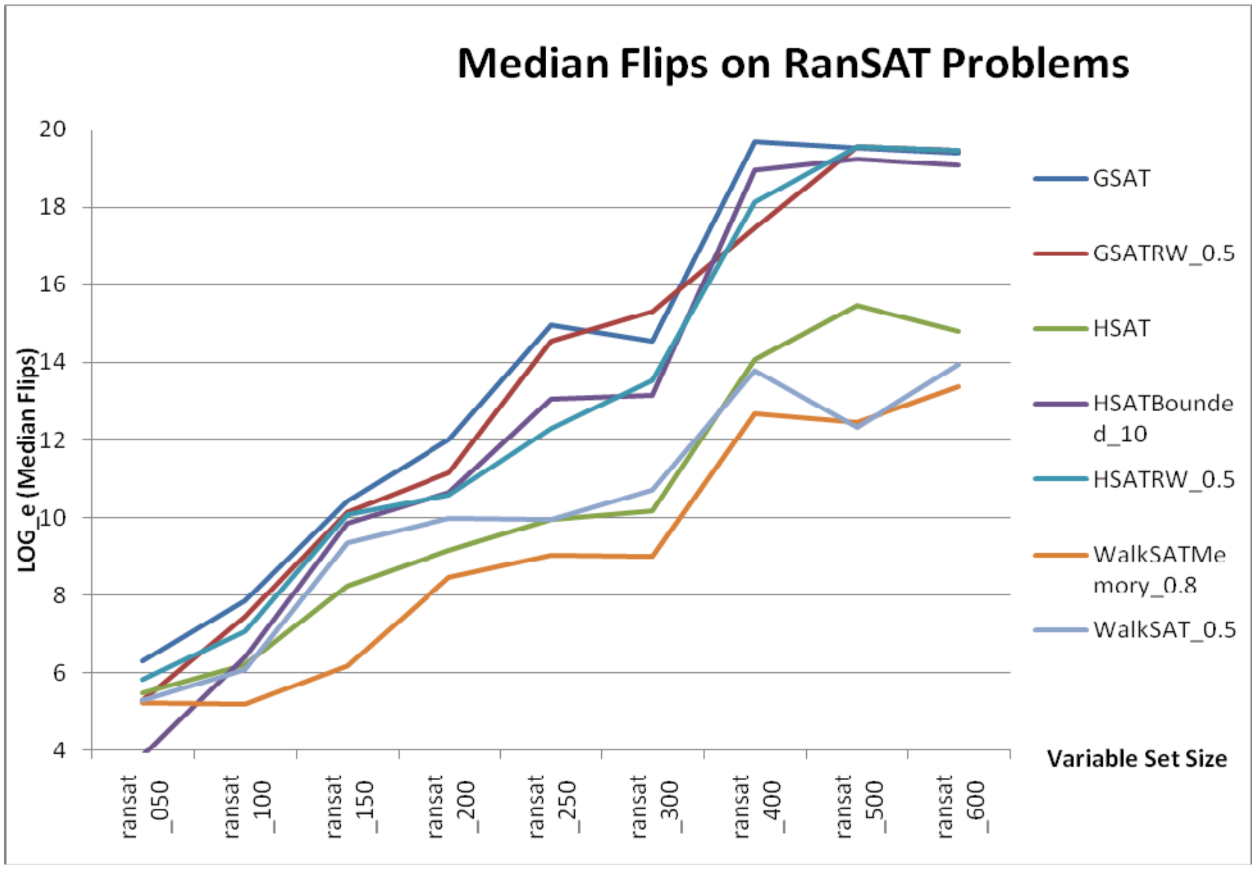
4.1 Presentation of results data

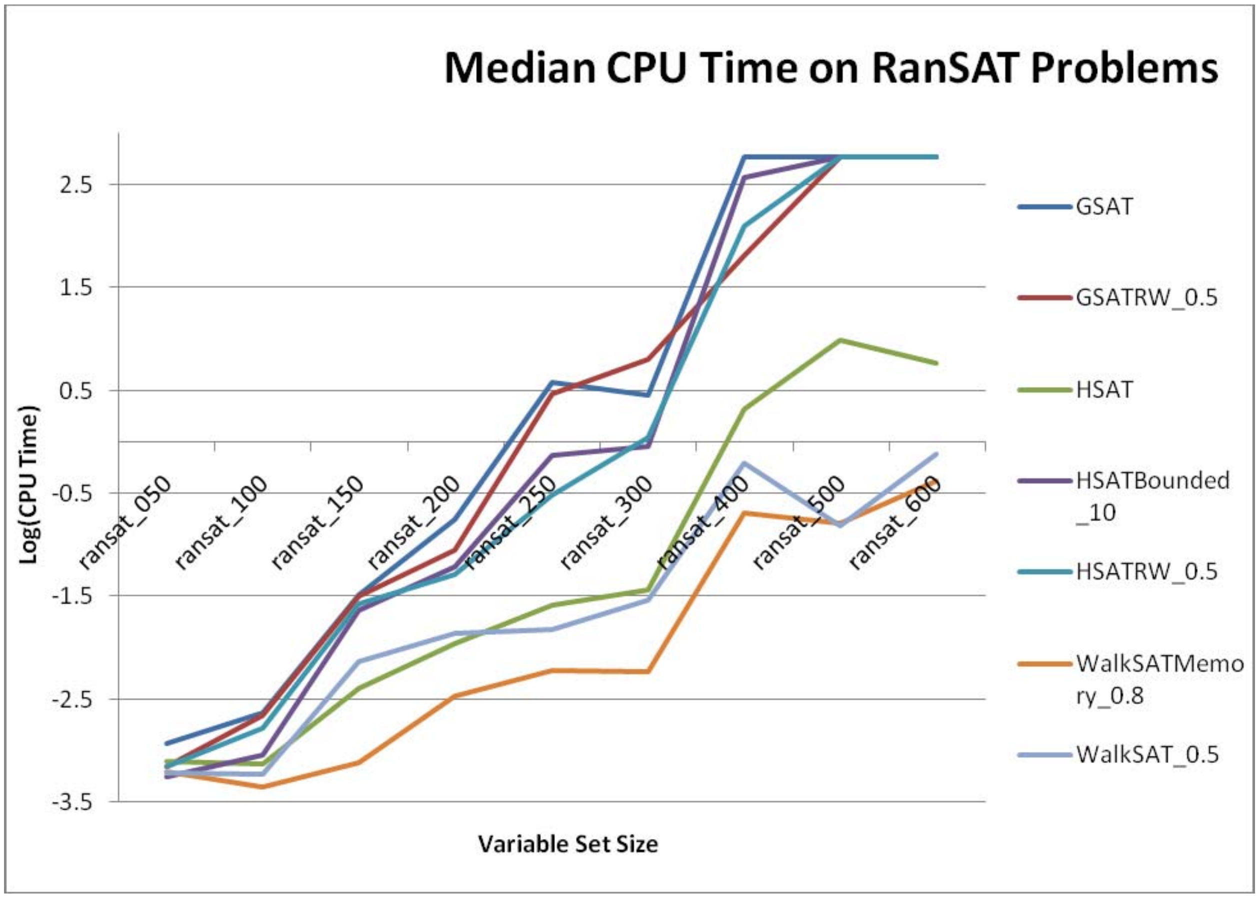
We ran all of our algorithms on the full suite of test problems. Additionally we ran GSATRW, HSATRW, WalkSAT and WalkSATmemory with three different probability parameter values each (0.25, 0.5, 0.75 for the first three, and 0.7, 0.8, 0.9 for WalkSAT with memory, for reasons which will become apparent). We obtained the data for HSAT with no caching separately from the bulk of the data, due to circumstances beyond our control.

Indeed, we experienced a series of setbacks in our endeavor to obtain this data (most notably being kicked and temporarily banned from the Stanford network due to our processes on the computer clusters being too computationally intensive…). However, the data presented in this paper is complete (with the exception of the data for problems ransat_700 and ransat_800).

4.2 General algorithm comparison (for RanSAT problems)

The two graphs below show performance for a portion of the solvers on the ranSAT problems.





The most obvious conclusion from these graphs is the dominance of WalkSAT, which outperforms the other algorithms by at least an order of magnitude in terms of both number of flips and CPU time (with the possible exception of pure HSAT).

Adding memory to WalkSAT further improves the algorithm's performance, which is intuitively encouraging: remembering our past flips allows us to alleviate some of the "aimless wandering" inherent in the random component of WalkSAT and the other algorithms, and improve the hill-climbing mechanism. Our results also show that the gains from using memory are greater than the costs of maintaining the memory data structure and using WalkSAT-Memory's pick() function.

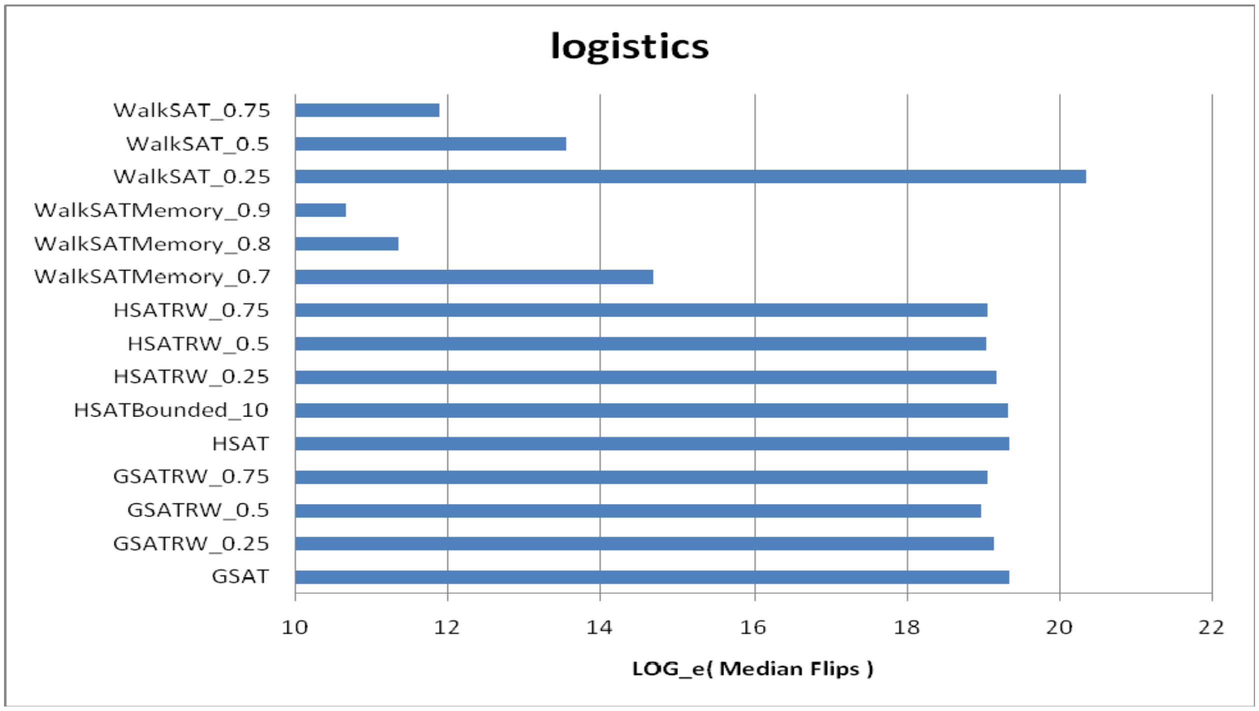
The original GSAT algorithm is the worst performer from all points of view, which was to be expected. Adding random walk to GSAT does not improve the performance as much as moving to HSAT or WalkSAT, but in most cases it performs a little better than standard GSAT.

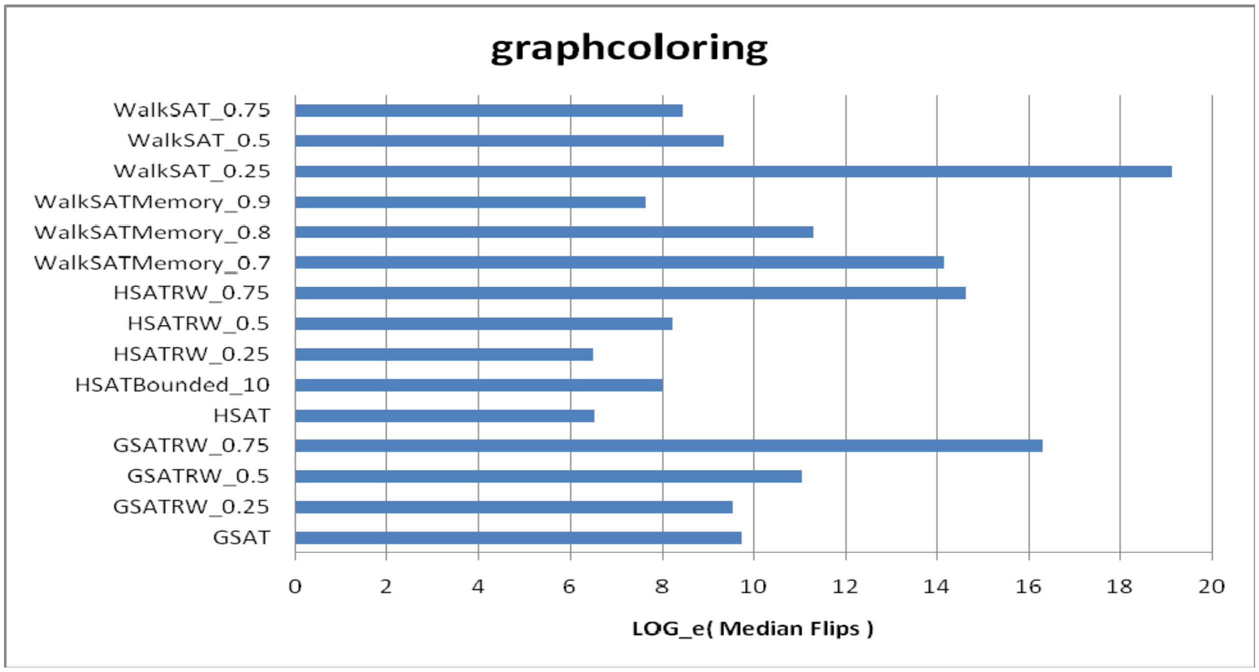
Pure HSAT is the best of the non-WalkSAT algorithms, which confirms the importance of adding memory to these algorithms. Significantly bounding the size of HSAT's memory (to 10 in these figures) destroys these performance gains. It therefore seems clear that the larger the memory at HSAT's disposal, the better.

Adding random walk to HSAT (forming HSATRW) also ruins HSAT's performance. This is somewhat dissappointing, as one might have hoped that some random walk would alleviate the problem of the algorithm getting stuck in local minima. This is clearly not the case, and we can understand why: the random steps we take with random walk damages the value of the memory, which no longer records just meaningful flips.

4.3 Comparison for other problems categories

For space's sake we will show the results for the logistics and the graph-coloring problems here.

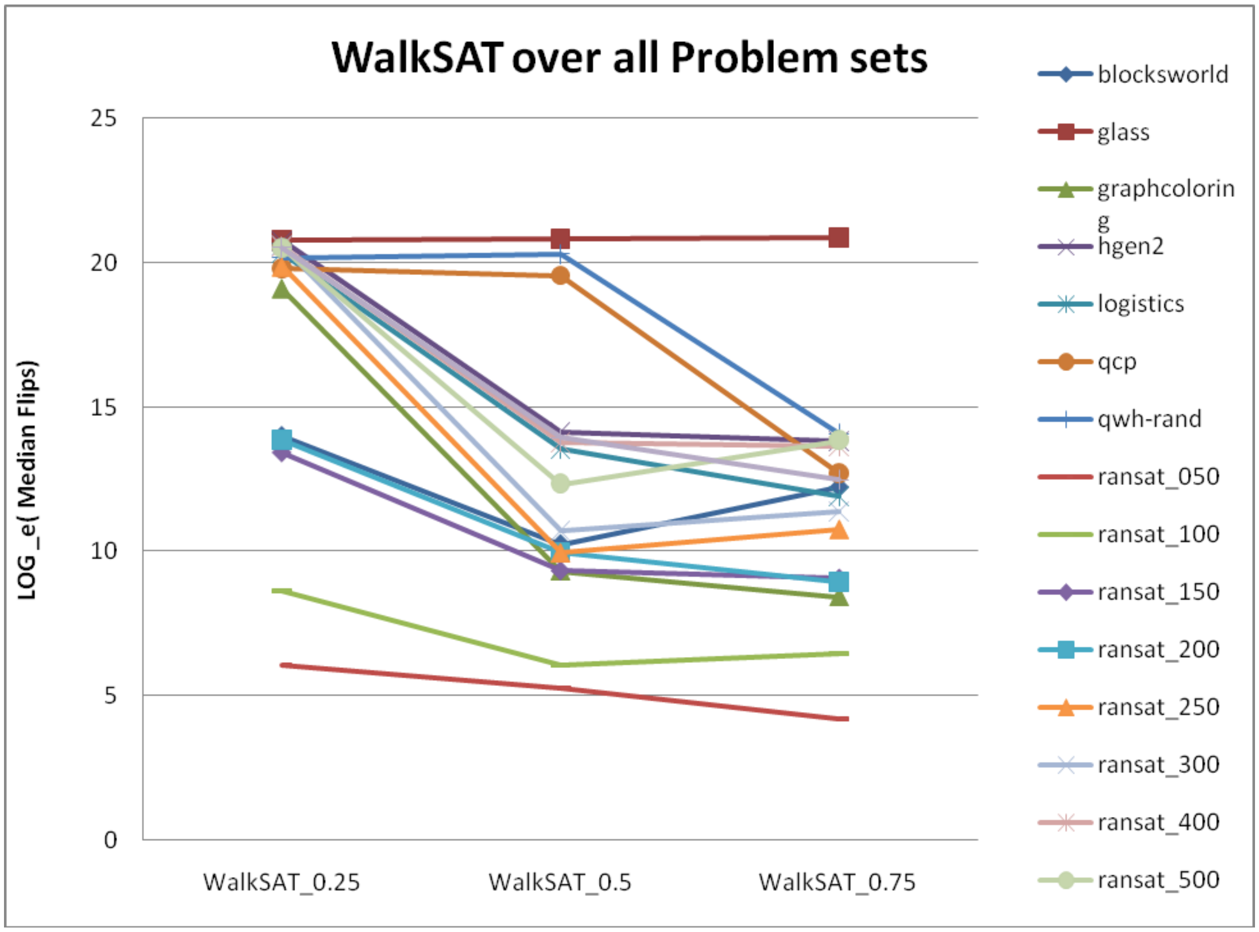




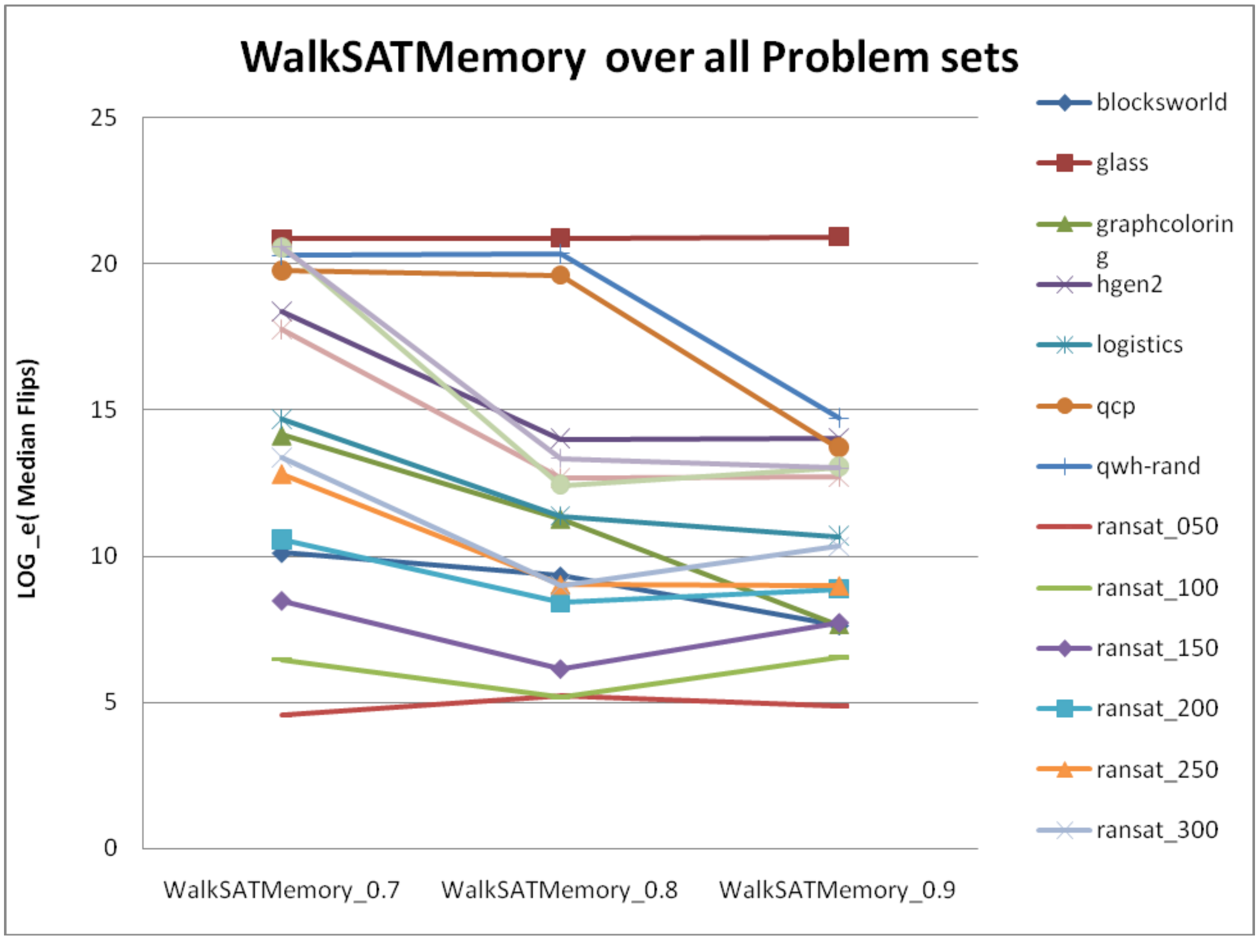
These results confirm the dominance of WalkSAT, especially with memory, and the reasonable performance of pure HSAT. It is notable that for a relatively easy problem such as graphcoloring, HSAT actually comes out on top for the flips metric.

4.4 Importance of parameter values

We tested several different parameter values for the algorithms which choose probabilistically between two modes. We show here the results for the two variants of WalkSAT, with three parameter values each.



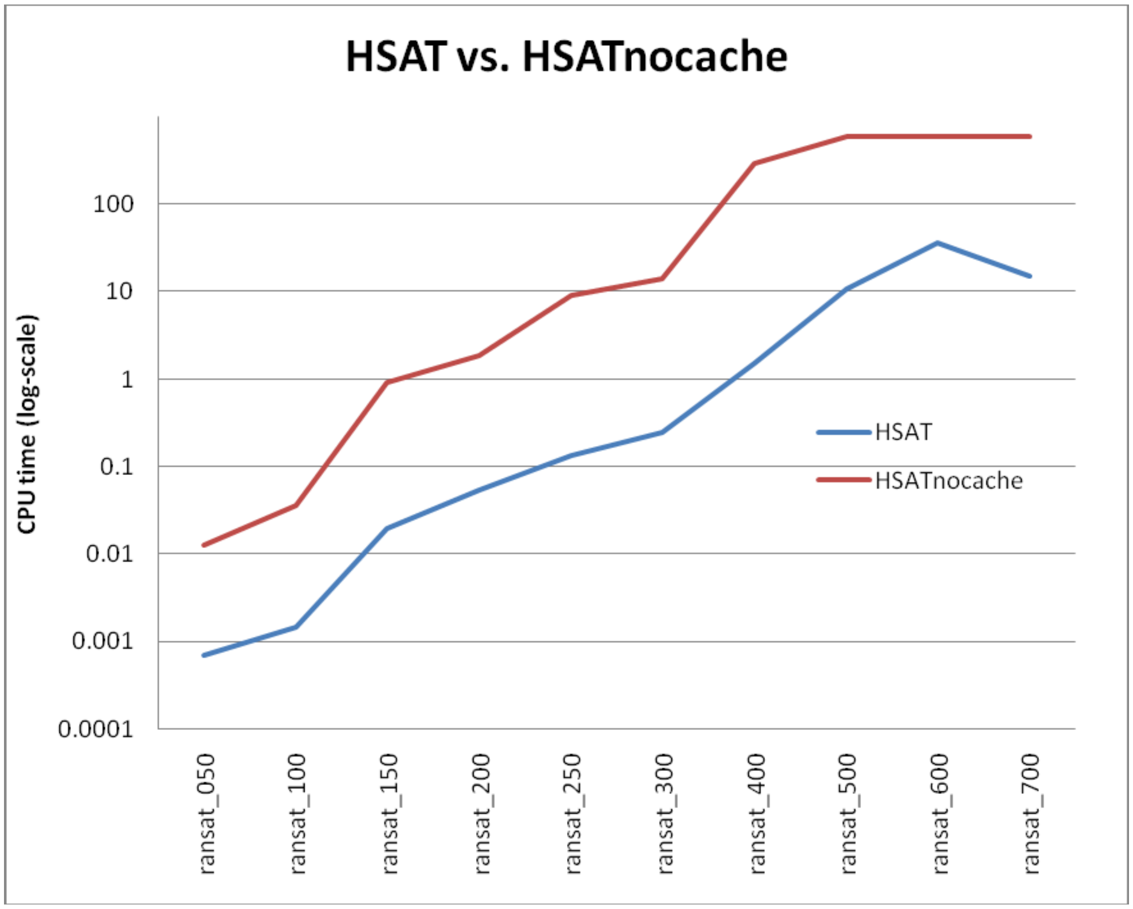
For standard WalkSAT, we note that a low parameter value (i.e. a high probability of the hillClimb function just returning all the variables in a random unsatisfied clause) is extremely bad. It is however harder to decide between 0.5 and 0.75: while p=0.75 provides a huge performance boost on the hard qcp/qwh problems, p=0.5 is better on pretty much all the other problem classes. Had we had more time it would therefore have been interesting to run the algorithm for intermediate parameter values (e.g. p=0.6, 0.7), to see if there is an “optimal” value.



For WalkSAT with memory, we note almost identical behavior, but for much higher parameter values (around 0.8 instead of 0.5). This corresponds to the idea already uncovered above: randomness is a lot more detrimental to algorithms which use memory of past moves, as random moves will just pollute the remembered history (as the remembered random moves do not correspond to failed deliberate moves).

4.5 Importance of caching

To evaluate the importance of caching, we implemented a version of HSAT without any caching. We compare the performance of HSAT with and without caching in the graph below.



These results clearly confirm the importance of caching for run-time performance (obviously caching has no effect on the number of flips). There is a pretty much constant factor of an order of magnitude between the performance of HSAT with and without caching on the RanSAT problems. Efficient caching mechanisms such as those that we implemented are therefore clearly key to good SAT solving performance.

5 Conclusion

In this paper we have presented a portfolio of hill-climbing based SAT algorithms, and the details of our C++ implementation and our optimizations. Our analysis of the results obtained for the various algorithms on the selection of problems point to several important conclusions. Notably, remembering the history of past moves can dramatically improve performance, but is fragile to greater degrees of randomness. In general, WalkSAT (especially boosted with memory) proved to be the best of our algorithms.