Andrew Sauber
Julian Kemmerer
ECEC 671
Project 2 – SAT Solver

ALAFASS

A Long Acronym for a Satisfiability Solver

Introduction

ALAFASS uses a combination of algorithms, heuristics, and data structures to achieve high performance satisfiability (SAT) solving of boolean functions. The main algorithm used is DPLL. Each recursive call to DPLL does the following: attempt to satisfy a set of clauses using boolean constraint propagation, check for the SAT/UNSAT case, heuristically choose a variable and value, then recurse on that variable choice. In the case of UNSAT, all writes made after that call to DPPL are reverted.

The second major algorithm used is that of boolean constraint propagation (BCP), specifically the method of 2-Literal Watching. It is the iterative “inner loop”, that attempts to complete most of the work. By “watching” only two literals in each clause for a change to 0-value, new implications and conflicts can be examined without needing to examine every clause. This is done using watch lists. A watch list is simply a list of clauses in which a certain variable is being watched. These lists are split into positive and negative versions (positive and negative forms of the literal being watched).

The heuristic used for making variable and value choice is the Jeroslaw-Wang score. This algorithm weights clauses in which a certain variable appears based on the number of non-zero valued literals present. This allows for more 'intelligent' choices of variables and values that are 'more important', i.e. in smaller clauses.

Data Structures

We have implemented several data structures that allow for minimal memory usage and increased speed. To begin, we have three classes of objects: variable, literal, and clause. A variable is an object that represents 'variables' from boolean algebra.

A variable can be assigned to either 1, 0 or not assigned at all. Each variable, to promote easy navigation through the data structures, has two lists: one list of clauses with positive polarities and one list with negative polarities. These lists are in additional to the watch list mentioned above. Variables

also include two pointers to corresponding positive and negative polarity literals.

Literals are a class similar to variables (and arguably could be implemented using inheritance). The literal class has two additional features: polarity and result. Polarity being the polarity of that specific literal, and when considered along with variable assignment the resulting value. Literals also contain a pointer to their 'parent' variable to provide easy navigation between the two data types.

Clause is the final implemented class. A clause is simply a collection of literals. Clause objects also contain pointers to the two watched variables in that clause. It can be said that a collection of clauses is an 'expression'.

Data Storage, Retrieval, and Runtime Evaluation

In terms of the actual data structures storing these objects, simple arrays are used. When the benchmark file is read-in, an array of clauses and variables is statically allocated. As each clause is read in, the fields of the clause, variables, and literals contain are processed and referenced via the pointers described above. Since literals are simply pointed to from within variables they are not organized with arrays but rather dynamically allocated; still, there will be at most 2V literal objects, where V is the number of variables in the expression. Arrays were chosen as they can be accessed in constant time. So accessing a specific clause or variable is very fast. Also, since literals are only one level of indirection away (one more pointer to follow) their access, even though not organized in arrays, can also be done in constant time.

There was also significant thought put into the evaluation of literals (with respect to their polarities) and the scoring done as part of heuristic variable choosing. As part of BCP, literals can have their values assigned as well as read. Assignment of variables only occurs once at the 'top' of a recursive call while representative readings of the variable's value occurs many times during BCP. By calculating the result (value) of an assignment for each literal only at the time of 'write' it prevents additional instructions being needed each time a 'read' occurs (which is more frequent). A similar technique is used for the weight values of clauses. Only when a literal is added or removed, which is only at the start of the program does the weight of that clause change. Thus, by calculating the weight only then, it avoids the additional instructions needed to calculate the weight on each heuristic iteration.

When collecting information needed for BCP, each clause on a specific watch list is visited. Within that clause it is required to gather information regarding 'the other watched literal', 'a non-zero and not the other watched literal'...etc. all for evaluating which BCP case is applicable. Rather than checking each case sequentially, a single loop through the literals of a clause is used and the

information for all 4 cases is gathered simultaneously. This eliminates unneeded iterations through lists of literals.

Improvements

As mentioned, the strengths of the implementation are it's bare minimum memory footprint and fast access times leading to high performance. The weaknesses of this approach are slow FP ops for Jeroslaw-Wang scoring. In the current version we have used pow(2, -x). In the next version we will implement a lookup table for [2^1, 2^2, 2^3 … 2^C], where C is the length of the largest clause. These will be stored in unsigned integers. Therefore, the array of these integers would look like an identity matrix. When accumulating a score for a given variable, the integers in this table will be accumulated. The winning variable would then be the variable with the lowest score rather than the highest score.

Future versions could also implement clause learning, which is where a lot of the performance advantages are realized in competition grade SAT solvers. Tournament heuristics could also be implemented, in which different branching heuristics compete on different threads. Making the solver itself multi-threaded would be a good extension. The performance benefit would have to outweigh the cost of copying literal and variable state each time a thread was spawned. Of course, random restarts could also be added to the current version.

Comparison with LSAT

ALAFASS was evaluated again LSAT. LSAT is a light weight SAT-solver tailored for diagnosing systems, based on propositional models, e.g., digital circuits. It is based on a non-destructive DPLL variant using mutually linked lists, and is able to minimize the result set with respect to a predetermined set of "components"/variables [1].

How to Run ALAFASS

Compile using 'make' with no arguments.

When running ALAFASS supply a '-l' flag to see 'level 1' debugging information. To see more detailed debug information the number 1-9 can be supplied as ' -l# '. To see no debug information, do not supply a '-l' flag.

Example Usage:

```
$ ALAFASS /cnfs/logic.cnf          (no debugging information)
$ ALAFASS -l1 /cnfs/logic.cnf      ('level 1' debugging information)
$ ALAFASS -l0 /cnfs/logic.cnf      (no debugging information)
```

Benchmark Results

The benchmarks used were selected from the DIMACS 'aim' set: Artificially generated Random-3-SATs [2]. A selection of 50 variable and 100 variable benchmarks were chosen.

Table 1. Benchmark Descriptions

Benchmark	Variables	Clauses
SAT 50-1	50	80
SAT 50-2	50	80
SAT 50-3	50	80
SAT 50-4	50	80
SAT 50-5	50	100
SAT 50-6	50	100
SAT 50-7	50	100
SAT 50-8	50	100
SAT 50-9	50	170
UNSAT 50-1	50	80
UNSAT 50-2	50	80
UNSAT 50-3	50	80
UNSAT 50-4	50	80
UNSAT 50-5	50	100
UNSAT 50-6	50	100
UNSAT 50-7	50	100
UNSAT 50-8	50	100
SAT 100-8	100	200
SAT 100-10	100	340
SAT 100-11	100	340
SAT 100-12	100	340
SAT 100-13	100	600

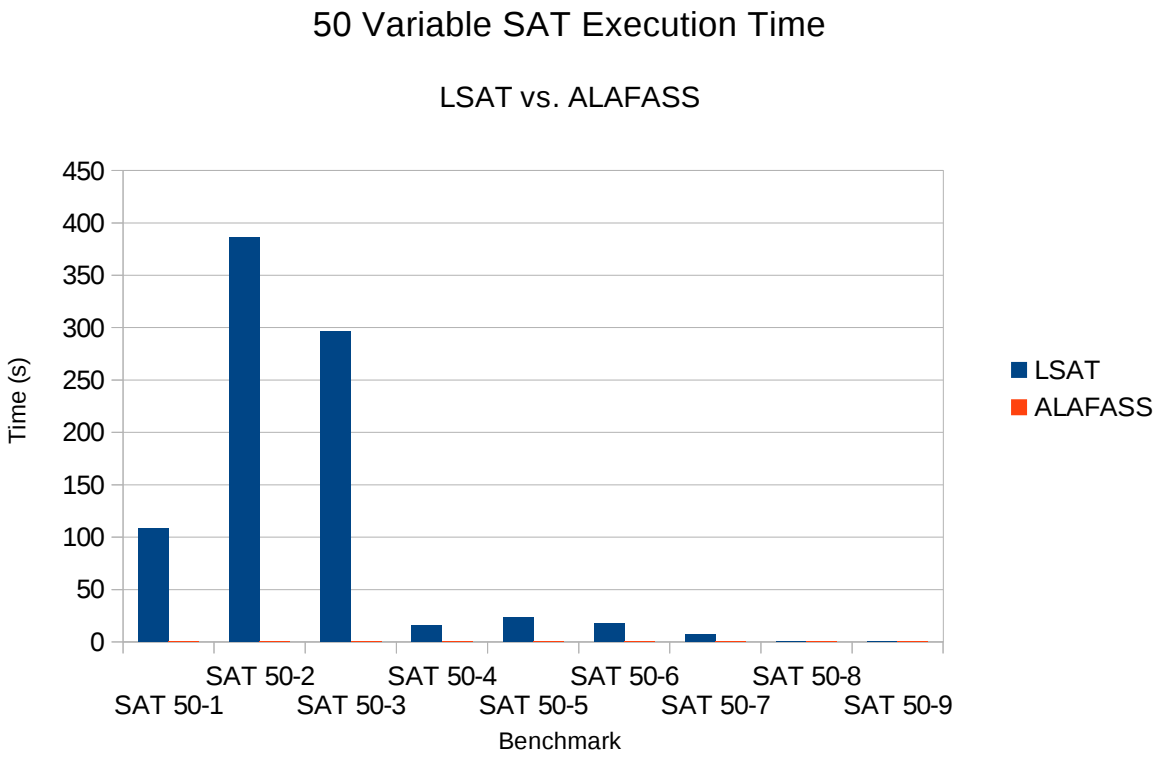

Fig. 1. SAT execution time of LSAT vs. ALAFASS using 50 variable benchmarks. In this chart, ALAFASS executes so quickly that its running time is barely visible. (Table 2.)

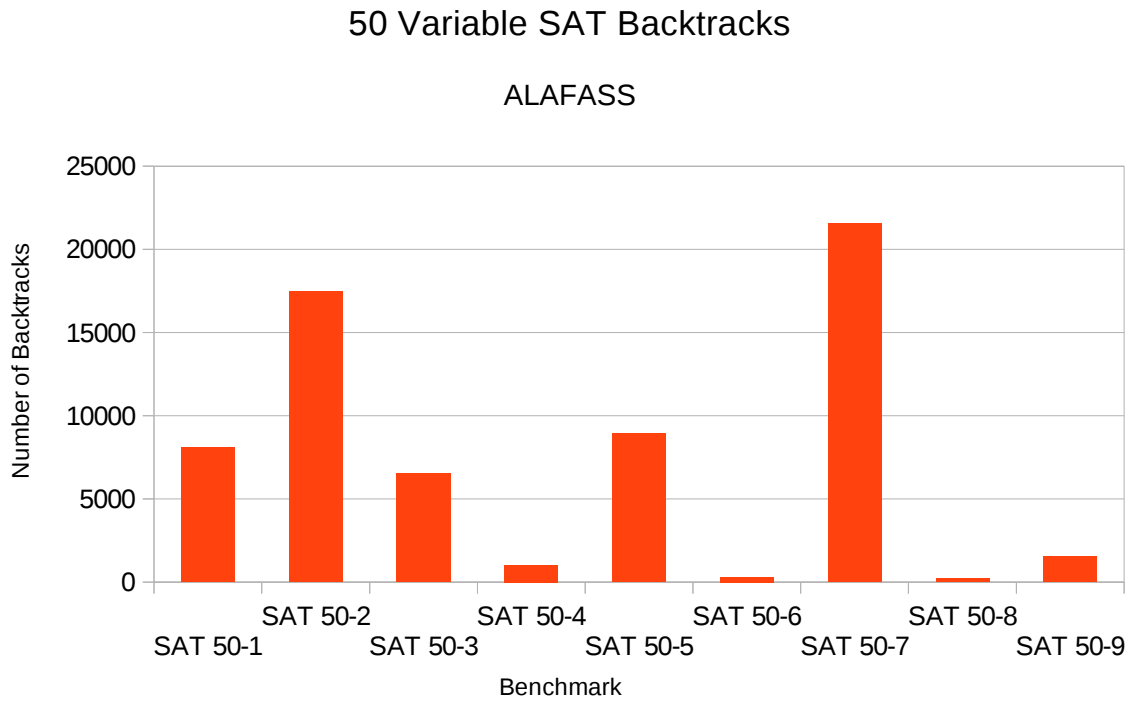

Fig. 2. Number of backtracks performed by ALAFASS finding SAT for 50 variables. (Table 3.)

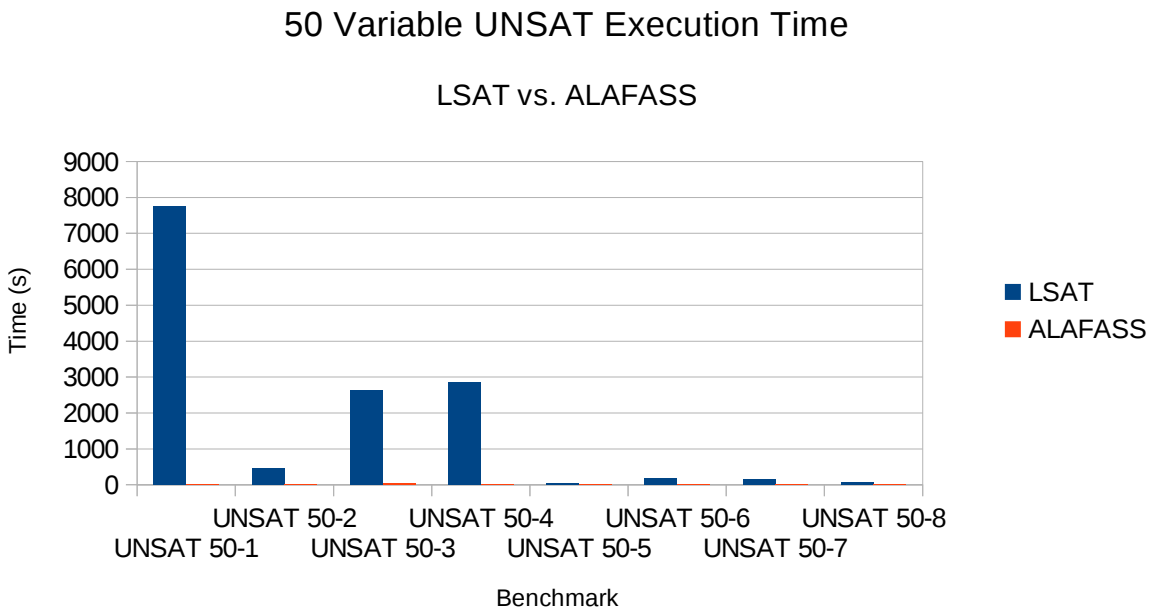

Fig. 3. UNSAT execution time of LSAT vs. ALAFASS using 50 variable benchmarks. Again, ALAFASS executes so quickly that its running time is barely visible. (Table 2.)

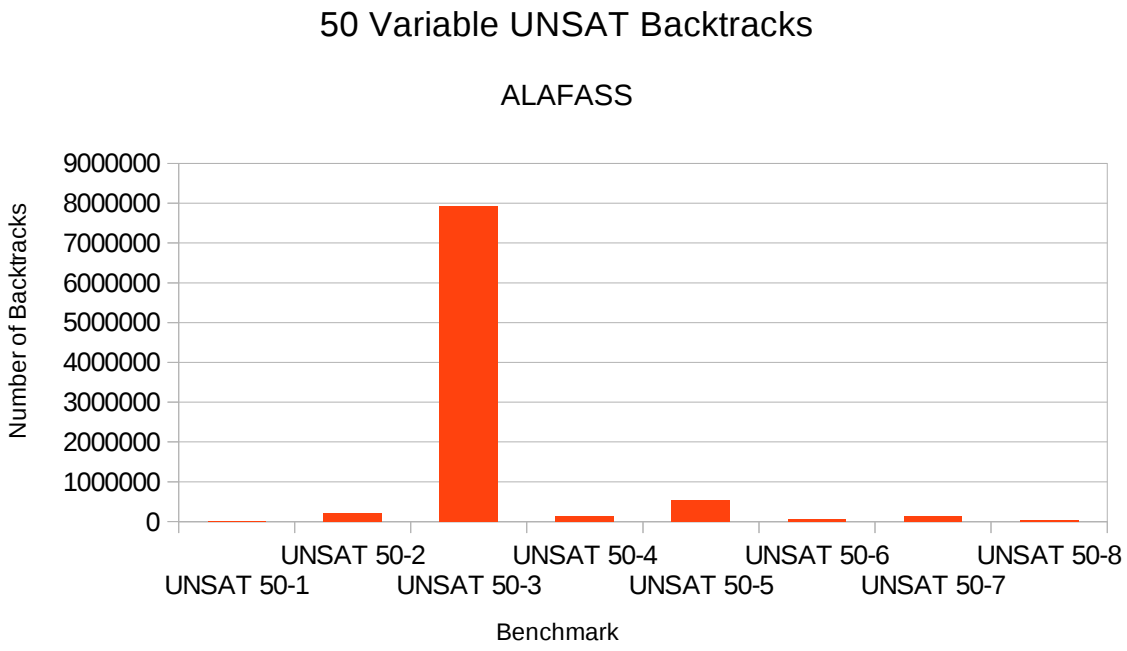

Fig. 4. Number of backtracks performed by ALAFASS finding UNSAT for 50 variables. (Table 3.)

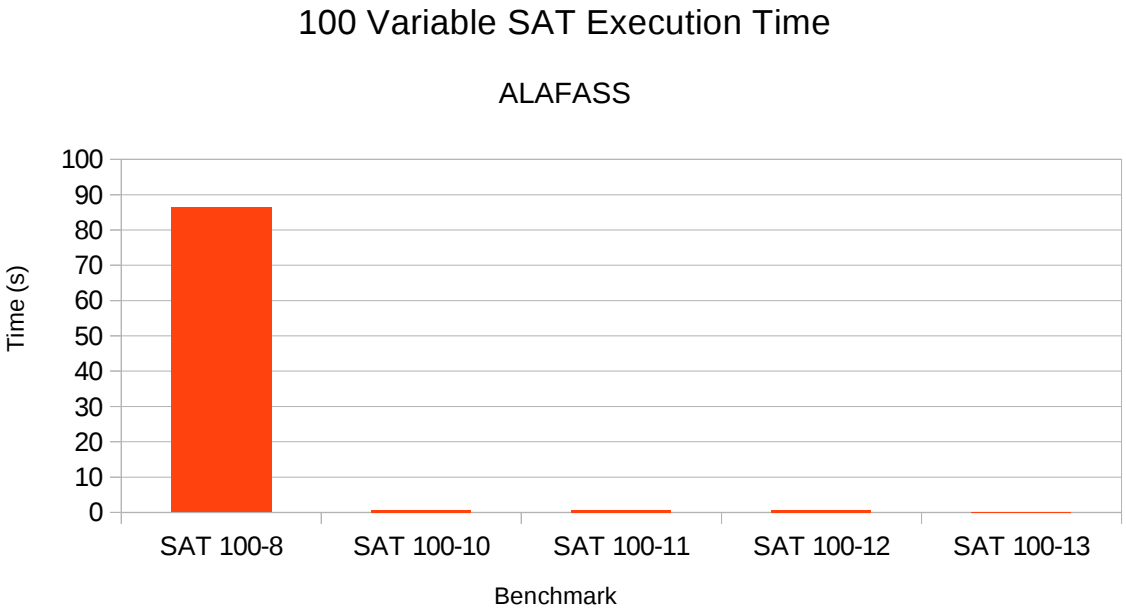

Fig. 5. Execution time of ALAFASS finding SAT for 100 variables. (Table 2.)

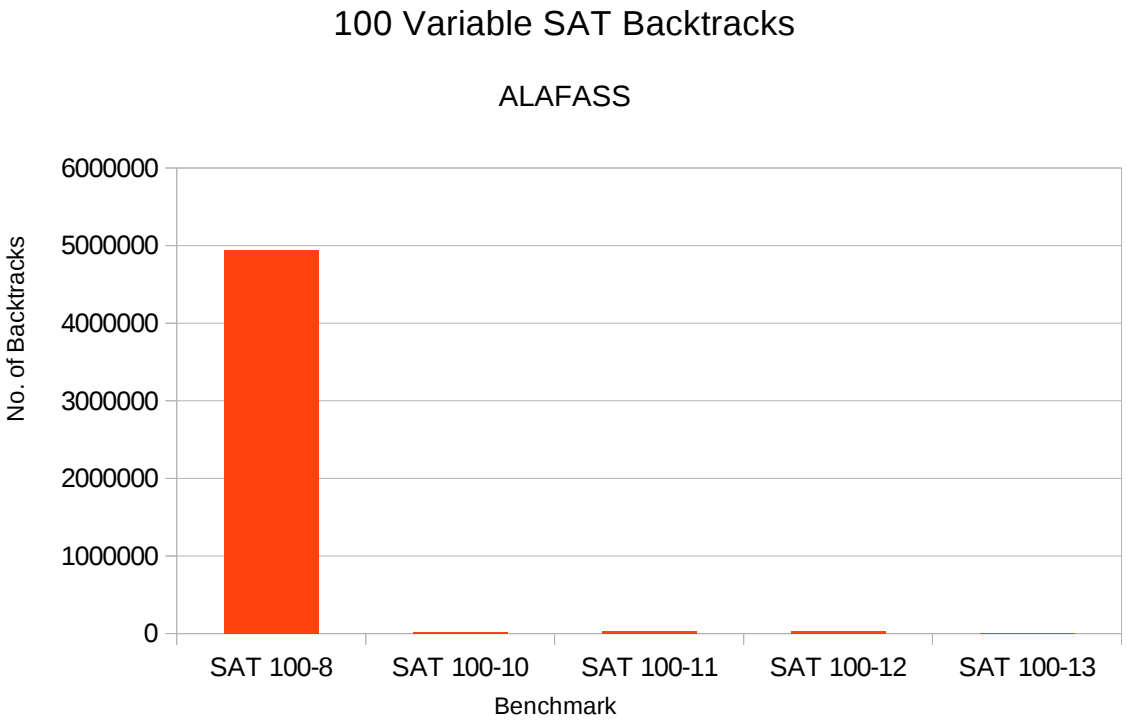

Fig. 6. Number of backtracks performed by ALAFASS finding SAT for 100 variables. (Table 3.)

Table 2. Execution time of LSAT vs. ALAFASS*

	Execution Time (s)		
Bench Mark	**LSAT**	**ALAFASS**	**LSAT / ALAFASS (speed multiple)**
SAT 50-1	108.11	0.08	1351.37
SAT 50-2	386.38	0.23	1679.9
SAT 50-3	296.19	0.08	3702.35
SAT 50-4	15.44	0.01	1544.15
SAT 50-5	23.21	0.09	257.84
SAT 50-6	17.43	0	17428.63
SAT 50-7	7.22	0.3	24.07
SAT 50-8	0.22	0.01	22.17
SAT 50-9	0.72	0.04	18.06
UNSAT 50-1	7742.14	0.09	86023.75
UNSAT 50-2	459.3	1.98	231.97
UNSAT 50-3	2621.68	56.57	46.34
UNSAT 50-4	2857.64	1.11	2574.45
UNSAT 50-5	38.75	6.52	5.94
UNSAT 50-6	179.26	0.87	206.04
UNSAT 50-7	140.53	1.97	71.34
UNSAT 50-8	80.72	0.81	99.66
SAT 100-8	14+ Hours	86.35	583.67
SAT 100-10	Not Run	0.79	--
SAT 100-11	Not Run	0.69	--
SAT 100-12	Not Run	0.7	--
SAT 100-13	Not Run	0.01	--

*Benchmarks were canceled after 14 hours.

Table 3. Number of backtracks using ALAFASS

Bench Mark	ALAFASS Backtracks
SAT 50-1	8072
SAT 50-2	17485
SAT 50-3	6516
SAT 50-4	1034
SAT 50-5	8936
SAT 50-6	305
SAT 50-7	21549
SAT 50-8	217
SAT 50-9	1534
UNSAT 50-1	12408
UNSAT 50-2	206770
UNSAT 50-3	7915030
UNSAT 50-4	138308
UNSAT 50-5	536726
UNSAT 50-6	59470
UNSAT 50-7	127034
UNSAT 50-8	45542
SAT 100-8	4947525
SAT 100-10	18182
SAT 100-11	24745
SAT 100-12	29067
SAT 100-13	194

Table 4. Number of backtracks vs. execution time using ALAFASS

ALAFASS Backtracks	Time Elapsed (sec)
194	0.01
217	0.01
305	0
1034	0.01
1534	0.04
6516	0.08
8072	0.08
8936	0.09
12408	0.09
17485	0.23
18182	0.79
21549	0.3
24745	0.69
29067	0.7
45542	0.81
59470	0.87
127034	1.97
138308	1.11
206770	1.98
536726	6.52
4947525	86.35
7915030	56.57

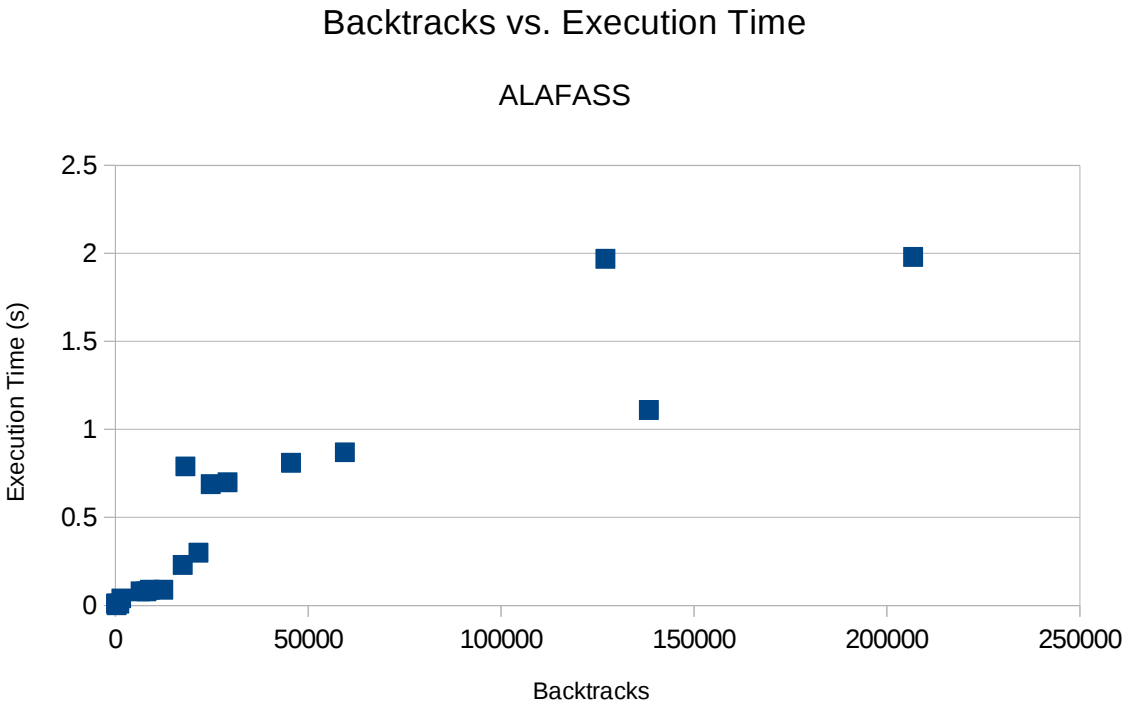

Fig. 6. Number of backtracks performed by ALAFASS vs. execution time (Table 4.)
Note: Three data points were removed for easier viewing (scale would have been over multiple orders of magnitude). The three data points also follow the linear trend.

Results Commentary

For each benchmark that was run, a function was executed in ALAFASS to verify that an identified SAT assignment indeed satisfied the boolean function. All unsatisfiable benchmarks were correctly reported as unsatisfiable by ALAFASS. ALAFASS is a functioning SAT solver.

Before discussing the execution time, a small note can be made regarding backtracks in ALAFASS. It appears as though execution time is linearly correlated to the number of backtracks (Fig. 6). This prompts for additional design effort in planning the data structures and algorithms used in backtracking.

The benchmarks that were run were of a relatively small size compared to the DIMACS benchmarks routinely used in SAT solver competitions. Those are on the order of 10s of thousands of variables. One reason that 50 and 100 variable benchmarks were chosen was to provide a reasonable platform for comparing LSAT to ALAFASS.

ALAFASS completed each of the SAT50 benchmarks in under 1 second, whereas LSAT took an average of 95 seconds for each of these benchmarks. This yields an average performance advantage with ALAFASS of 1,074x. With 50 variable UNSAT benchmarks, the performance advantage is even greater at 11,157x. This type of benchmark was where we began to see LSAT limit our ability for comparison, it having an average running time for these benchmarks of 29 minutes.

LSAT was not able to complete even one 100 variable benchmark even after running for *14 hours*. On the other hand, ALAFASS was able to find a satisfying assignment for 4 out of 5 of the 100 variable benchmarks that were run in *under 1 second*. This level of performance, combined with the minimal, statically allocated, memory footprint of ALAFASS, indicates that it is capable of satisfying much larger problems. Further testing will be done without the imposition of a time limit. It is also interesting to note that LSAT and ALAFASS are both written in C++, with ALAFASS having fewer lines of code.

References

[1] Andreas Bauer. Simplifying diagnosis using LSAT: a propositional approach to reasoning from first principles. In Proceedings of the 2005 International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR), volume 3524 of Lecture Notes in Computer Science, Prague, Czech Republic, pages 49-63, June 2005. Springer-Verlag. http://users.cecs.anu.edu.au/~baueran/lsat/index.html.

[2] Y. Asahiro, K.~Iwama, and E.~Miyano Random Generation of Test Instances with Controlled Attributes. In D.S.Johnson and M.A.Trick, editors, Cliques, Coloring, and Satisfiability: The Second DIMACS Implementation Challenge. Vol. 26 of DIMACS Series on Discrete Mathematics and Theoretical Computer Science, pages 377--394. American Mathematical Society, 1996. pages 127-154, 1996.