SATe – An experimental Java SAT Solver

Michael Whelan
E-insights High Performance Systems Ltd.

October 2017

SATe is a Java implementation of a SAT solver. Input is a boolean expression in Conjunctive Normal Form[1] and output is an indication of whether the boolean expression specified in satisfiable or not and in the case of satisfiable expressions an assignment that satisfies the expression is generated. In cases where either repeated solutions to smaller problems are needed or the solution to larger problems, SATe is comparable to MiniSat[2] when multi-threading is used; based on testing against the UF and UUF test sets available from SATLIB. The source code is available at bitbucket.org[3] under the MIT/X11 open source license.

Approach

SATe first forms the negation of the CNF expression, creating a Disjunctive Normal Form (DNF) and then attempts to find an assignment that causes the complemented expression to fail and hence would cause the initial expression to be satisfied.

The process to identify an assignment that causes the complemented expression (now in DNF form) to fail is to first split the problem into two sub problems based on assigning true and false values to a selected literal. We will discuss the selection of the literal to 'split on' later.

Each sub problem is then examined (in an order that will also be discussed later). This examination performs four simplifications which will be briefly mentioned here and more fully explained in the Appendix.

First, any clause of length 1 (that is single literals) are identified and the literal is assigned the value necessary to cause the clause to fail. (Since in DNF form a single clause being satisfied – in this case the single literal clause – satisfies the expression.)

Secondly, literal usage is examined and if a literal is identified that is used in only one form (either complemented or not), then the sub-problem is simplified by assigning this literal a value such that all clauses involving it fail. The justification for this is that for the original problem to be non-satisfiable, all such problems derived from the complement of the original must be universally[3] satisfiable.

Finally, clauses involving two literals are examined. Two literal clauses that involve the same literals (for example *a&b* or *a&!b*) must either result in 'fixed variable pairs (FVP)' or 'contrarian pairs' . A FVP would for example be *a&b* and *a&!b* with '*a*' being the fixed literal and '*b*' the variable in this case. The fixed literal can be assigned to cause both clauses to fail since otherwise one of the clauses would succeed irrespective of the value assigned to the variable literal. In case of a contrarian pair (for example *a&b* and *!a&!b*) a relationship between literals 'a' and 'b' can be inferred, in this case that 'a' and 'b' must be opposite (or *a = !b*) since otherwise, one of these clauses and hence the DNF as a whole would be satisfied.

As this process proceeds, several situations can arise. An effort may be made to assign a value to a literal that already has been assigned a value. If the new value and preexisting value are the same then the process continues. If the new and preexisting values differ then the sub-problem in question is universally satisfiable and can be discarded. The set of clauses may become exhausted

1 Conjunctive Normal Form or CNF – in DIMACS syntax.
2 MiniSat version 2.2.0 .
3 Universally satisfiable in the sense that any assignment to the literals satisfies the expression.

in which case the existing assignment must be such that it will cause the original expression to succeed and so a certificate has been found. Note that the ‘assignment’ that is associated with a sub-problem, even one that is identified as satisfying the original problem, may not necessarily assign values to all literals. Some literals may be ‘don’t care’ or constrained by other literals but not assigned specific values.

This is the full set of simplifications exploited. The algorithm proceeds to split sub-problems and perform the above simplifications until either an assignment that causes the complemented expression to fail is identified or until all sub problems have been eliminated. The only other elements to the algorithm are the choice of literal to split on and the order in which sub-problems are investigated.

The choice of literal to split on is similar to a steepest descent root finding approach. Each unbound literal in a sub-problem is examined to determine how many single literal clauses would be generated by setting it true and false respectively, and similarly how many new two literal clauses would be generated. Finally for each literal the number of long clauses (clauses involving more than three literals) that it is involved in is determined. These values define a cost function which is maximized. The literal which results in the maximum of the cost function is chosen as the next literal to split a problem on.

Once the literal to split on has been determined, each sub-problem (resulting from setting the 'split' literal to true and false respectively) is assigned a ranking based on the true/false elements of the cost function and the remaining problem complexity (number of unbound literals and remaining clauses).

Sub-problems are tackled in order of lowest remaining ranking until either an assignment causing the DNF sub-problem to fail is encountered (in which case the same assignment must cause the initial CNF to be satisfied) or until all sub-problems are shown to be universally satisfiable and hence the initial CNF is not satisfiable.

Time Comparisons

The performance of SATe was compared to that of MiniSat version 2.2.0 tackling problems from the SATLIB[2] suite of test cases. These test sets are designated ufXXX or uufXXX for solvable and unsolvable problems respectively with XXX being numeric indicating the number of literals over which the candidate expressions are formed. These are all 3CNF formulas (the tools can of course handle arbitrary CNF expressions). Time data was gathered on an Intel i7 3770K 3.5GHz processor with 16GB of RAM running OpenSUSE. Java compilation and run-time were based on JDK 8u144.

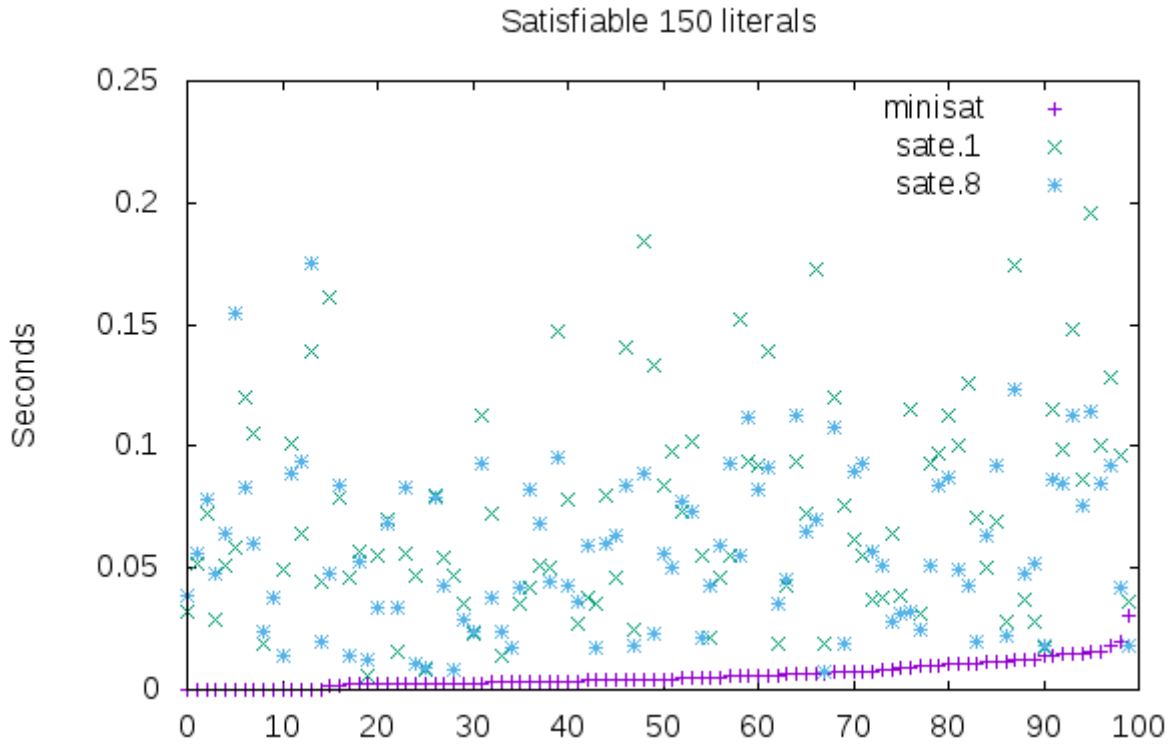

Figure 1. SATe and MiniSat timing – 150 literals satisfiable.

Figure 1 shows the performance of SATe with one and eight threads (sate.1 and sate.8 respectively) and MiniSat on the 100 elements of the UF150 test set. The results are ordered such that the MiniSat timing is increasing to the right.

Figure 2 below is similar but with the UUF150 test set – all of which are non-satisfiable. In both cases, while MiniSat performs better, SATe performance is surprisingly good given that it is implemented in Java.

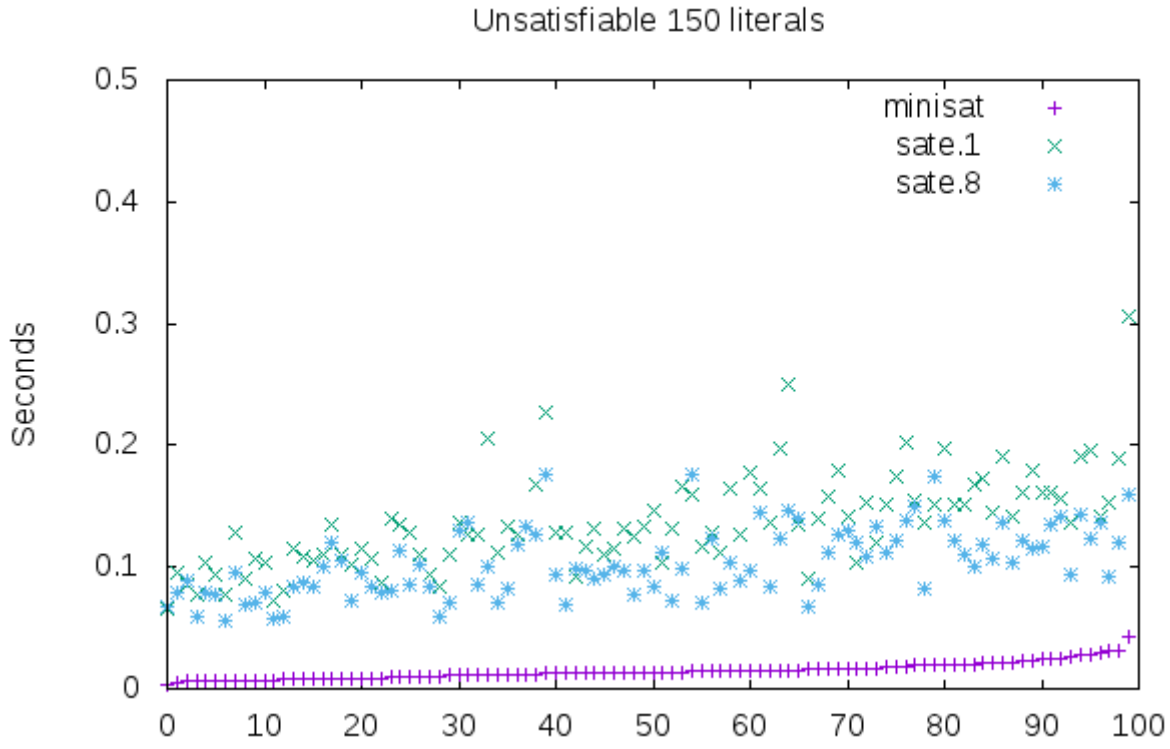

Figure 2. SATe and MiniSat timing – 150 literals non-satisfiable.

Java vs C++

A brief explanation of the differences between Java and C++ will help in understanding the rest of the discussion on performance. Java is an interpreted language with run time optimization (Just in Time or 'JIT' optimization). C++ is a compiled language with extensive compile time optimization. As a consequence, a C++ implementation of a given algorithm would be expected to be several times faster than the same algorithm implemented in Java, with Java having other advantages in terms of portability and ease of development.

The above timing results for SATe were generated by running the Java application separately[4] for each expression in the test set. The necessary class loading for the Java application then happens for each test case individually and the JIT optimizations are done as the program executes. In a situation where repeated problems need to be solved, the class loading and run-time optimization costs would be amortized across the problems rather than incurred ab-initio for each.

Figures 3 & 4 show the same test cases as Figures 1 & 2 respectively, but where the individual test cases are solved in sequence by the same Java process – thus amortizing the class loading and run-time optimization costs.

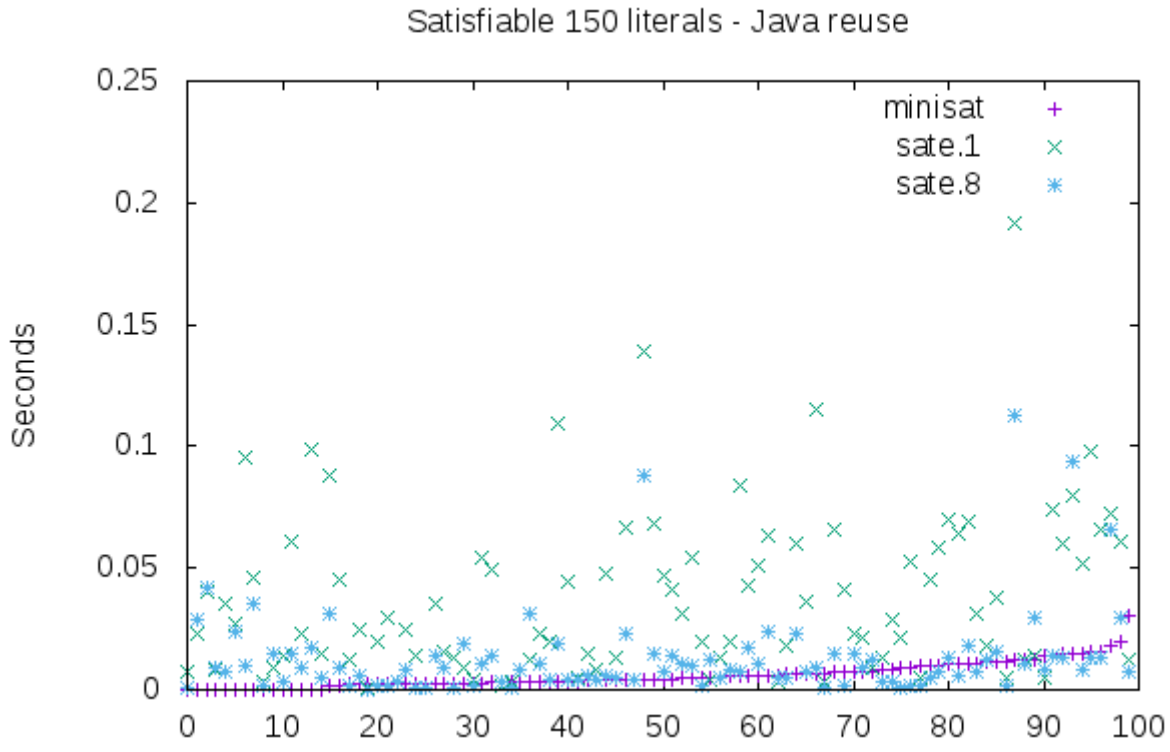

Figure 3. SATe and MiniSat timing – 150 literals satisfiable ; Java reuse.

4 A separate process.

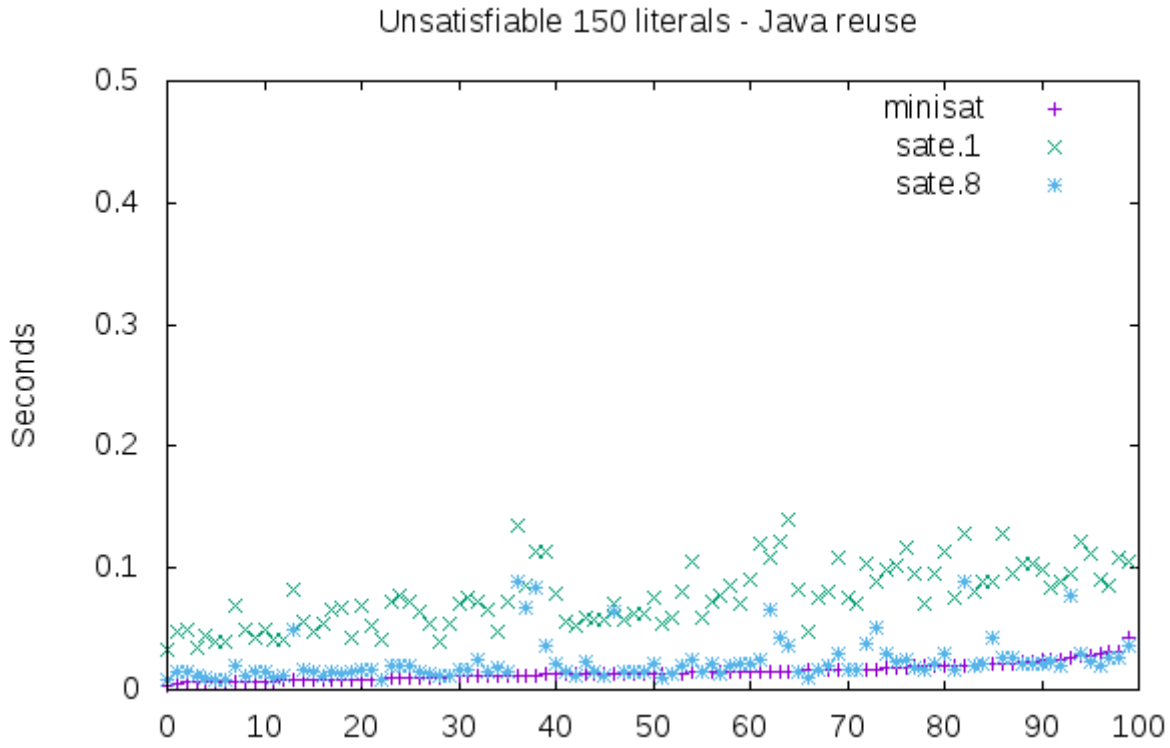

Figure 4. SATe and MiniSat timing – 150 literals un-satisfiable ; Java reuse.

As can be seen by comparing Figures 1 and 3 and more clearly in comparing Figures 2 & 4, SATe benefits very significantly in situations where repeated problem solution is necessary. The same is true as the size of the problems increases since the 'once off' costs for class loading and JIT optimization have less impact on the total running time as the problem sizes increase.

Larger Problems

Figures 5 & 6 below show performance on the UF250 and UUF250 test sets. In these cases, SATe with eight threads performs comparably with MiniSat[5] for the satisfiable tests and considerably better for the non-satisfiable cases. The plots below do not involve any Java re-use. In cases where re-use is applicable, SATe would show improved performance.

5 The version of MiniSat used here is single threaded. I was not able to locate a multi threaded version.

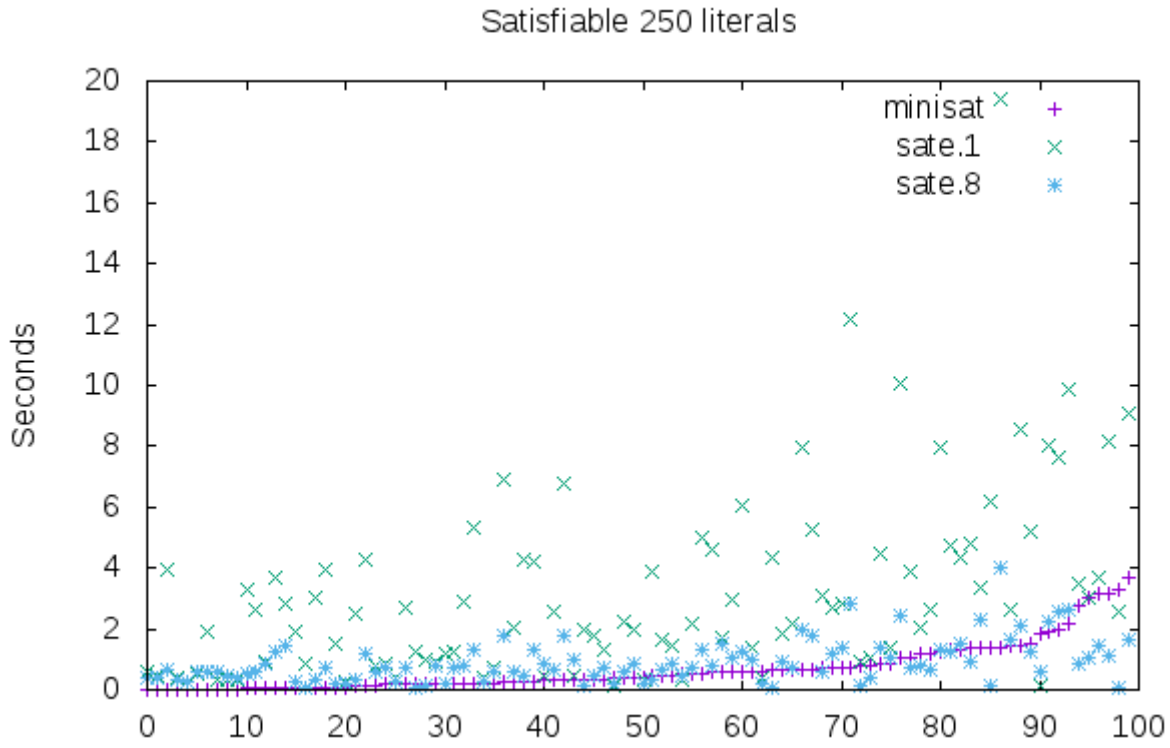

Figure 5. SATe and MiniSat timing – 250 literals satisfiable.

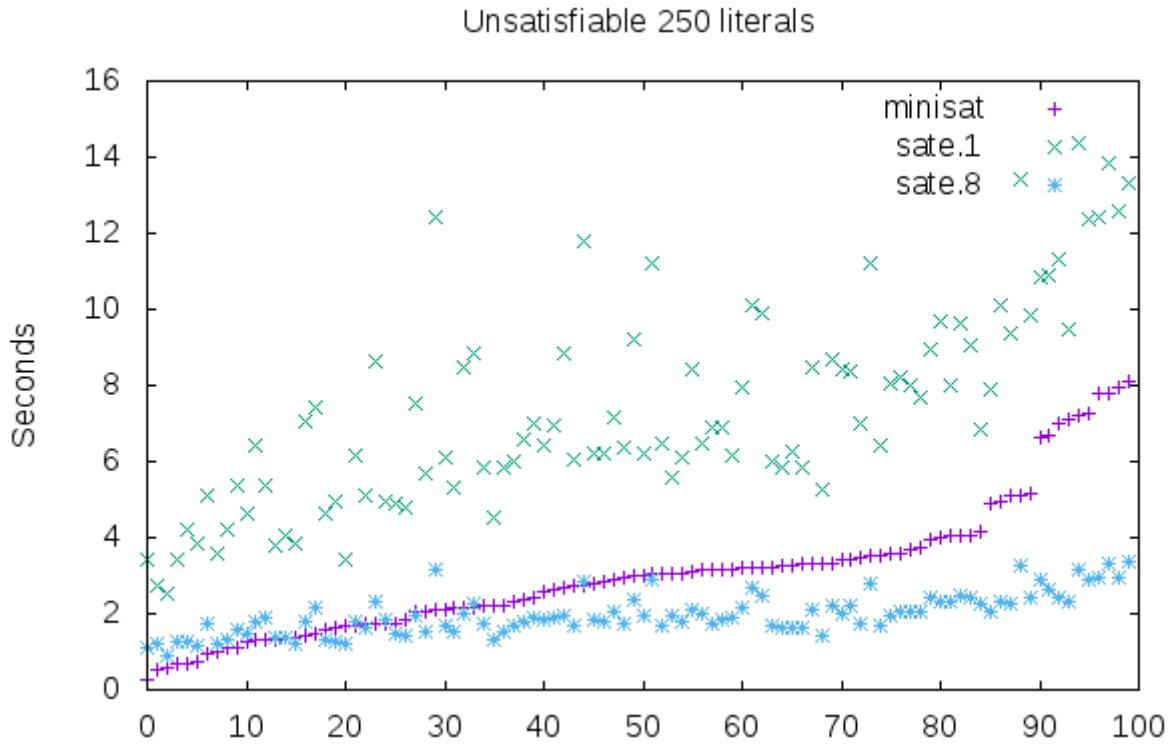

Figure 6. SATe and MiniSat timing – 250 literals non-satisfiable.

Tables 1 & 2 below show the average (time in seconds to solve a single instance) on the UF and UUF test cases as the number of literals varies from 150 to 250. As can be seen, SATe.8 compares very favorably to that of MiniSat 2.2.0 in all cases.

Literal Count	MiniSat Avg. Seconds	SATe.8 No reuse	SATe.8 reuse
150	***0.006***	0.076	0.014
175	***0.022***	0.166	0.025
200	***0.061***	0.350	0.068
225	0.212	0.490	***0.153***
250	0.696	0.965	***0.458***

Table 1: Average seconds to solve satisfiable (UF) test problems.

Literal Count	MiniSat Avg. Seconds	SATe.8 No reuse	SATe.8 reuse
150	***0.015***	0.142	0.023
175	***0.044***	0.330	0.051
200	0.172	0.585	***0.148***
225	0.710	0.986	***0.416***
250	3.087	2.048	***1.292***

Table 2: Average seconds to solve non-satisfiable (UUF) test problems.

Harder Problems

Given the underlying NP-complete nature of the SAT problem, it must be expected that there will be expressions upon which any given algorithm will perform poorly. The results presented are based on a small set of test cases. Performance on other problem sets will vary significantly. To explore harder problems, the test cases from ***Solving Difficult SAT Instances in the Presence of Symmetry*** [2] were examined. In performing these tests, the maximum run time for any individual problem was limited to 1800 seconds (half an hour). Any instance that was not solved within this time is shown in the tables below as '***' .

Table 3 below shows the results when attempting to solve the 'Pigeon-Hole' problem set. In this case SATe.8 does well in comparison to MiniSat.

Test Case	Satisfiable ?	MiniSat Secs	SATe.8 Secs
hole7.cnf	NO	0.035	0.035
hole8.cnf	NO	***0.142***	0.189
hole9.cnf	NO	1.212	***0.734***
hole10.cnf	NO	86.960	***5.689***
hole11.cnf	NO	822.540	***64.350***
hole12.cnf	NO	***	***852.580***

Table 3. Solution time for the Pigeon-Hole problem test cases.

Table 4 below shows the comparative performance on the FPGA Switch-Box test cases. In this case both MiniSat and SATe handle these tests without challenge, though MiniSat is clearly superior.

Test Case	Satisfiable ?	MiniSat Secs	SATe.8 Secs
fpga_10_8_sat	YES	***0.002***	0.006
fpga_10_9_sat	YES	***0.002***	0.006
fpga_12_8_sat	YES	***0.001***	0.004
fpga_12_9_sat	YES	***0.002***	0.007
fpga_12_11_sat	YES	***0.002***	0.009
fpga_12_12_sat	YES	***0.002***	0.007
fpga_13_9_sat	YES	***0.001***	0.004
fpga_13_10_sat	YES	***0.001***	0.030
fpga_13_12_sat	YES	***0.001***	0.209

Table 4. Solution time for the FPGA Switch-Box problem test cases.

In sharp contrast to the FPGA Switch-Box tests, the Randomized Urquhart tests are not handled well by either program. Only the smallest problem instance (Urq3_5.cnf) is solved by either program within the prescribed time limit. This (non-satisfiable) problem is solved in ***101.5*** seconds by MiniSat and in 286.6 seconds by SATe. All larger instances exceed the maximum time set. So neither approach does well.

The final test cases are the Global Routing tests. These five (all satisfiable) test cases are all solved by MiniSat in approximately one second, while not a single one is solved by SATe within the allocated 1800 seconds. This clearly shows the limits of the simple approach taken by SATe to choosing the order in which to bind variables to create sub-problems and the prioritization of sub-problems. Future work on this topic appears warranted however given the overall performance of SATe.

Appendix – Simplifications

The purpose of this appendix is to justify the simplifications that were described above. The context is that we have a DNF expression, that is an expression that consists of the OR of a set of clauses, each clause being the AND of literals or their complements. We are attempting to find an assignment to the literals which will cause the expression to evaluate to FALSE, and hence would also cause the initial CNF expression to evaluate TRUE (or be satisfied).

The four simplifications are:

Single Literals Clauses – If the DNF contains a clause with only a single literal, the literal in question can be assigned a value which causes the clause to fail. The opposite assignment could not possibly lead to the DNF as a whole failing.

Single Form Usage – If any literal used in the DNF is always used in the same form (that is always complemented or un-complemented) then it can be assigned a value that will cause all the clauses containing it to fail. To see that this is valid, consider the set of clauses in the DNF to be [L_x C] where L_x represents the clauses containing the literal in question while C represents the clauses that do not contain that literal. If we were to split this problem on the literal in question we would obtain two sub problems with the clauses [C] and [L_r C] where the literal was set to the value causing the clauses containing it to fail and to the other value respectively. L_r refers to the set of clauses which had contained the literal in question simplified by removing the literal from the clause (setting it so that the reference evaluates to true in the disjuncture).

Any assignment that causes [L_r C] to fail must by definition cause all the clauses in C to fail. Hence by focusing on just C we do not risk missing a failing assignment if one exists. It is the case that assignments in which the literal in question is set the other value and which causes the overall DNF to fail may also exist. However we are looking to find a single such assignment not all such assignments.

Fixed Variable Pairs – for example *a&b* and *a&!b* with '*a*' being the fixed literal and '*b*' the variable in this case. If we were to split an expression containing this FVP on literal *a*, the sub-problem resulting from setting *a* to TRUE would be universally true (since it contains the clauses *b* and *!b*). We can consequently immediately set *a* to FALSE in the current sub-problem without going through the split process.

Contrarian Pairs - for example *a&b* and *!a&!b*. Any expression containing a contrarian pair such as this will be universally satisfied (that is irrespective of the values assigned to any remaining unassigned literals) if literal *a* is equal to literal *b* (either TRUE or FALSE). Consequently in attempting to find an assignment that causes such an expression to fail, the literals *a* and *b* must be assigned opposite values. A similar case occurs with contrarian pairs of the form *!a&b* and *a&!b* in which case the literals must be assigned the same value. We can consequently eliminate one of the literals using the relationship to rewrite all clauses in terms of one variable only – taking care to handle clauses that may be modified so they contain two references to the same literal – either in the same form or complemented forms.

References

[1] SATLIB – Benchmark Problems – http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

[2] ***Solving Difficult SAT Instances in the Presence of Symmetry;*** Aloul, Ramain, Markov & Sakallah – Design Automation Conference 2002. Test cases available at http://www.aloul.net/benchmarks.html

[3] https://mpmwhelan@bitbucket.org/mpmwhelan/sate.git