

A Visual SAT Solver

Derrick Stolee

March 30, 2009

1 Introduction

A solver for CNF-SAT problems was implemented with the Java language. It takes input using the DIMACS suggested clause format. Using a backtrack search with forward-checking and dynamic variable-value ordering, the solutions are found relatively quickly, compared to naïve approaches. Using the Java Swing library, a visual tracker of the backtrack history was created to allow a visual method of tracking the solution progress. This interface can be run simultaneously with the search, although it does slow the implementation by a constant factor, so it can also be visualized after the search is complete.

2 Compiling and Running

In order to compile easily, the file `dstolee-hw07.zip` is a packaged Eclipse project. Use the Import wizard in Eclipse to import a project from the file system using this archive file. You can also unzip it and use the unpacked directory to import. Compiling outside of Eclipse is more difficult, due to the package structure, but information can be given by request. To run, either run a Java Application from the `SATSolver` class, or execute the JAR file `satviz.jar`. This JAR file can be run by double-clicking in a GUI environment such as Windows, OS X, or the X Window system, or by running the command

```
java -jar satviz.jar
```

3 Data structures

Several data structures were implemented in order to store the variable/clause structure of a SAT problem combined with the necessary structure for backtrack search.

3.1 Constants

The interface `Constants` allowed for global constants corresponding to the truth values of variables or clauses at any given time:

TRUE A positive result for a variable. Also a positive result for a clause evaluation, if any of the (possibly negated) variables included are positive.

FALSE A negative result for a variable. Also a negative result for a clause evaluation, if all of the (possibly negated) variables included are negative.

OPEN An undecided value for a variable. Also an undecided result for a clause evaluation, if all of the variables are negative or undecided.

CONTRADICTION A contradictory value for a variable, meaning that if the variable takes a positive or negative value, at least one clause evaluation will result negatively.

3.2 Variables and clauses

Variable values are stored as `bytes`, the entire variable list stored as a `byte` array. A `Clause` class is defined to store clauses of a SAT formula. It holds a `List` of integer values called `variables` storing the indices of the variables included in the clause. A non-negative value i corresponds to x_i contributing to the clause. A negative value j translates to variable \bar{x}_i , where $i = -j - 1$, contributing to the clause. The -1 in the translation allows x_0 to have negative contribution. This format was created to match the `byte` array that stores the variable values, which are zero-indexed.

The `Clause` class defines a method `byte eval(byte[] values)` that takes a set of variable values and evaluates the clause to a positive, negative, or undecided result, depending on the values given. This is used to determine the validity of a partial solution to a SAT instance, as none of the clauses can evaluate negatively.

Also, a method `int contribution(int i)` returns the “contribution” the given variable gives to the clause. That is, a $+1$ contribution is returned if x_i appears in the clause, and -1 is returned if \bar{x}_i appears. 0 is returned if neither or both are included. This contribution is used for the dynamic variable ordering.

3.3 Problem instances

The class `Instance` stores a SAT problem instance, as well as a partial solution in the search tree. Two constructors are defined. One constructor takes an integer for the number of variables. This is used to build the problem statement. Then, a constructor that takes an `Instance` object, variable index, variable value, and a boolean marker will copy the problem instance and modify the variables to take the given value at the given variable. The boolean marker signifies the object to be a right or left child in the search tree. Unit propagation helps to determine other values of the variables if possible. This happens by removing values from the variable domains that cause a clause to evaluate negatively. If both values are removed, the variable is set to `CONTRADICTION` and the `Instance` object is marked as invalid.

4 Search

An `Instance` object instantiated with the first constructor stores a list of `Instance` objects called `history`. This list contains all search nodes as they are created, allowing for the visualization engine to show the search progress. The method `generateHistory()` performs the backtrack search while modifying the `history` list.

First, a copy of the `Instance` object is inserted to a stack. Then, depth-first-search is performed by pulling the top of the stack and checking if that `Instance` can be extended or should be popped. If the `Instance` is recognized as valid, a variable is chosen to test a value. The copy constructor is used to create a new `Instance` and push it onto the stack. If the `Instance` is invalid, the stack is popped until a left child is found. This left child is used to create a right child with the opposite value set on its chosen variable, ensuring completeness of the algorithm.

4.1 Dynamic variable/value ordering

The overall contribution of the open variables is computed by summing the result of the `contribution` method for all `Clause` objects in the instance. The largest absolute value gives the variable that is most skewed to be positive or negative, by “request” of the clauses. The sign of the contribution is used to determine which value the variable is given.

5 Visualization

The visualizer prints rectangles onto a green background, corresponding to the values set on the variables. The columns correspond to variables, and the rows correspond to the history of the search. A white rectangle represents a `TRUE` value, black is `FALSE`, and red is `CONTRADICTION`. `OPEN` variables are left transparent, showing the green background. The history can be scrolled through using the scrollbar. Figures 1 through 5 demonstrate this visualization on a few SATLIB challenge problems.

6 Performance

Overall, the performance is weak due to the large overhead in the implementation. However, the size of the search seems to be very strong, giving sub-linear number of search nodes for some of the challenge problems (cite?). Other problems took much longer searches, but were still better than random or static variable orderings.

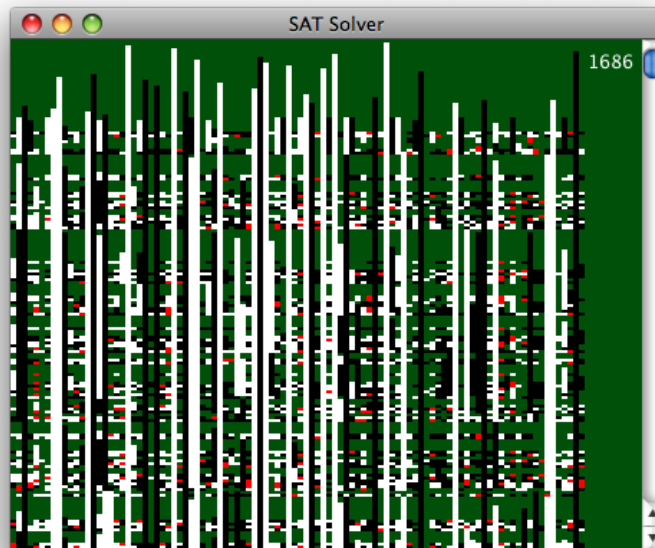


Figure 1: A SAT solution that does not fit in one screen

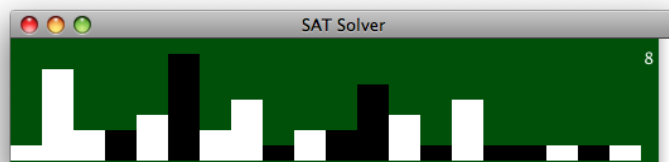


Figure 2: A SAT solution

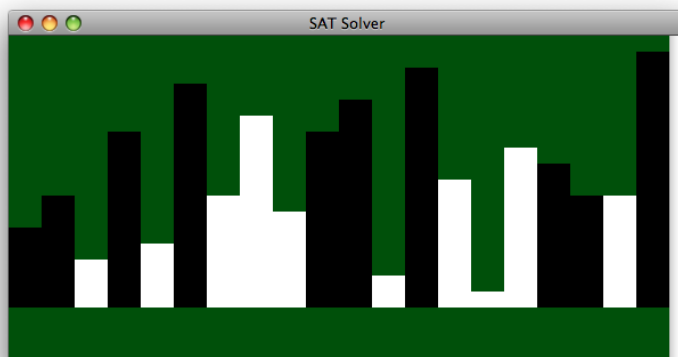


Figure 3: A SAT solution

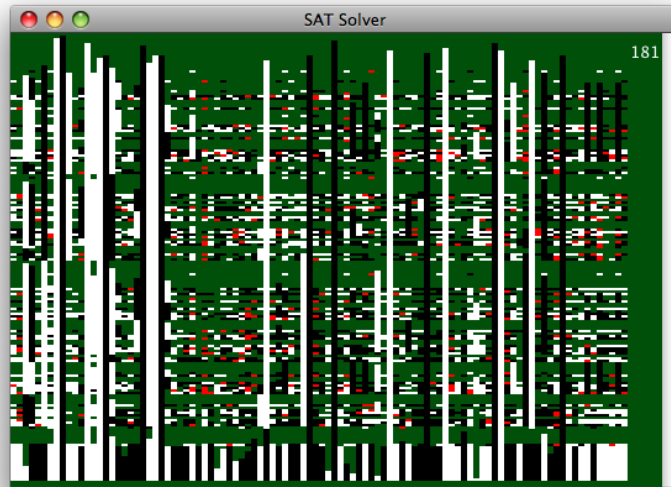


Figure 4: A SAT solution

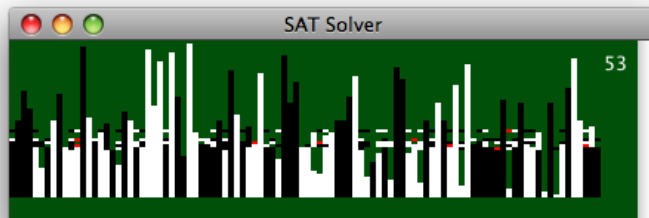


Figure 5: A SAT solution