# Ternary Tree Solver (tts-4-0)

Ivor Spence
School of Electronics, Electrical Engineering and Computer Science
Queen's University Belfast
i.spence@qub.ac.uk

31st May 2007

## 1 Introduction

The Ternary Tree Solver (tts) algorithm is a complete, deterministic solver for CNF satisfiability. This note describes the operation of version 4.x. Version 4.0 was entered into the SAT 2007 competition. The solver is very loosely based on the well-known Davis-Putnam model and has five phases, namely: Minimization; Variable ordering; Tree building; Tree walking, Rebuilding.

The solver cannot compete with the state-of-the-art solution of large industrial and random benchmarks but appears to have good worst-case performance on hand-crafted benchmarks (such as hgen8, holen, xor-chain etc.) that others find difficult.

Brief descriptions of the five phases follow.

## 2 Minimization

Here, if possible, the problem is first partitioned into disjoint sub-problems in which any variable can be reached from any other. The rest of the algorithm processes each sub-problem separately.

Any clauses that are tautologies, i.e. contain both $v$ and $\overline{v}$, are removed. If all occurrences of a particular variable are of the same sign it is safe to assign the corresponding value to the variable and hence remove any clauses containing it. This is combined with unit clause propagation.

## 3 Variable ordering

Unlike most Davis-Putnam solvers the variables are processed according to a static ordering. The overall performance depends critically on this ordering, in which variables that occur in the same clause should be processed near to each other. If the variables are regarded as nodes and the clauses as hyperedges, this corresponds to the *minimum linear arrangement* problem for hypergraphs. A perfect solution to this problem is known to be NP-hard, and so an approximation algorithm is used. Note that this approximation affects the overall performance, but not the correctness of the solver.

The approximation algorithm used for small inputs (of the order of fewer than 1000 literals) combines:

1. Simulated Annealing - this is generally regarded as providing the best approximations for MLA, but the execution time is significant;

2. A local search to see whether the simulated annealing result can be improved.

For larger inputs this algorithm is too slow and a more direct algorithm is used in which variables are chosen in turn according to weights which are derived from the number of clauses in common with variables already chosen. This is much faster so at least the solver has a chance of processing larger, easier inputs but does not give such a good ordering.

It should be noted that this phase of the overall solver is the most recently developed and is where most of the current research effort is devoted. In particular, although reducing the linear arrangement certainly improves the overall performance it is not known whether this is exactly the correct metric.

# 4 Tree building

At the heart of the algorithm is a 3-tree which represents the proposition to be solved. Each node of this tree corresponds to a proposition and each level corresponds to a variable according to the variable ordering which was determined in the previous phase. The tree is constructed as follows:

- The root of the tree corresponds to the proposition to be solved.
- Each node of the tree has three children, *left*, *middle* and *right*. The left child consists of those clauses from the current proposition that contain the literal $v$ except that the $v$ is removed (where $v$ is the current level). The right child consists of clauses that contain $\overline{v}$ except that the $\overline{v}$ is removed. The middle child consists of those clauses that don't contain $v$ or $\overline{v}$.
- When the removal of a $v$ or $\overline{v}$ leaves a clause empty this generates the proposition *false*. When there are no clauses to be included in a child this generates the proposition *true*.

In summary, the middle child consists of those clauses not containing the current variable, the left child is what remains after setting $v$ to *false* and the right child is what remains after setting $v$ to *true*. If the current variable is not contained anywhere in the proposition then no children are created at this stage - children are only assigned when the current variable is relevant. The construction of this tree can be carried out quickly.

A hash table of derived propositions is maintained during the tree building process. This is ensures that if different partial assignments lead to the same proposition only one corresponding node is created. The data structure thus contains cycles and is no longer a tree, but can be interpreted as a tree for the purposes of the next stage.

# 5 Tree walking

This phase is where the bulk of the computation occurs. Starting from the root, the walk has in principle a false/true choice to make at each level, representing an assignment to the corresponding variable, which would by itself lead to $2^n$ routes (where $n$ is the number of variables). Each node of the tree represents only a portion of the proposition, and so a set of nodes is maintained to record progress. For the false branch from a particular set of nodes, the new set of nodes consists of the union of all left and middle children of the current set. For the true branch, the new set consists of all right and middle children.

There are three outcomes which can result in a path being pruned, i.e. abandoned before the full depth of variables has been explored:

- When a set of nodes contains *false*, no further computation is performed on that branch because there can be no satisfying assignment built from the choices up to this point;
- If a set contains all *true* nodes a satisfying assignment has been found, regardless of the choice of values for subsequent variables;
- When a set of nodes has been found to correspond to an unsatisfiable proposition a record of this is made. If a subsequent request is made for the same set (or indeed a superset) it is known immediately that this is unsatisfiable without having to repeat the previous analysis. This corresponds to *clause memoization* and in this form is perhaps the main contribution of this solver.

# 6 Rebuilding

If any of the sub-problems is found to be unsatisfiable then the overall problem is unsatisfiable. Otherwise, if all the sub-problems have been found to be satisfiable, some of the actions of the initial minimization have to be undone to construct the overall model. Variables removed because they only occur with one sign are inserted with the appropriate value and the models for each of the sub-problems are renumbered as required.

# 7 Conclusions

An upper-bound for the complexity of this algorithm appears to be the sum of the number of sets of nodes at each level, which is exponential in the number of nodes at that level. This number of nodes is at most the corresponding cut of

the hypergraph and can be less than that. Improving the results from the variable ordering phase is expected to be the best way to improve the overall algorithm.

# 8 Change log

## 8.1 4.1 → 4.2

The generation of certificates of unsatisfiability, which had been present in an earlier version, was re-introduced. There is a version of the solver (ttsp) which can generate Allen van Gelder's RPT proofs or tts-specific PRT proofs (for which a checker, checkprt, is also included)

## 8.2 4.0 → 4.1

Some minor changes to the c code to permit compilation in 64-bit mode. This mode means that pointers consume more memory and it is an open question whether the solver performs better in 64 or 32 bit mode.

## 8.3 3.0 → 4.0

In version 3.0 the tree was built with a new node for child and there was a separate phase during which nodes which represented the same proposition were merged. In version 4.0 a hash table of propositions is used so that only one node is ever created for a given proposition even if it can be derived in different ways by partial assignment of different clauses.

In version 3.0 if a proposition did not contain a given variable the corresponding tree node had "true" for it's "left" and "right" children and a node representing the same proposition again for it's "both" child. In version 4.0 the number of nodes is significantly reduced by recording the first variable which is relevant for each node and only creating child nodes when the level in the tree corresponds to this variable.

In version 3.0 the static variable ordering was determined using simulating annealing to find a minimum linear arrangement. This produced good results but the time taken increased rapidly with the number of variables. In version 4.0, if the size of the input exceeds a given threshold a more direct algorithm is used which does not produce such a good ordering for the tree walk but is much faster.