Adapting the Weighted Backtrack Estimator to Conflict Driven Search

Shai Haim Toby Walsh

University of New South Wales, Australia

and

NICTA

{shaih,tw}@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-0805
February 2008

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

Modern SAT solvers present several challenges to estimate search cost including coping with nonchronological backtracking and learning. We present a method to adapt an existing algorithm for estimating the size of a search tree to deal with these challenges. We show the effectiveness of this method using random and structured problems.

1 Introduction

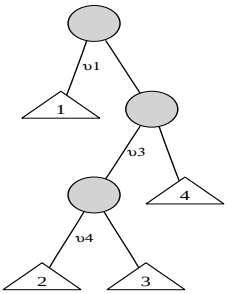
Simple backtracking SAT solvers like DPLL unfold a proper-binary decision tree. The Weighted Backtrack Estimator (WBE) [2], which is an adaptation of Knuths offline sampling method [3] can generate good estimates of search cost for such solvers. However, more modern SAT solvers present several challenges for estimating their runtime. For instance, clause learning repeatedly changes the problem the solver faces. Estimation of the size of the search tree at any point should take into consideration the expected changes that future learning clauses will cause.
WBE does not support a range of solvers which use Conflict Driven search. These solvers unfold a search tree which is not a proper binary tree. In this report we explain the challenge and suggest a solution for Our approach to these problems is to use an on-line method to predict the cost of the search by observing the solvers behaviour in a small part of search. Our method uses the size of the currently explored search tree to predict the size of the unexplored search tree.

2 Related Work

Knuth suggested a method which used random probing to estimate the size of a backtrack search tree [3]. If b_i is the branching rate observed at depth i of the probe, then $1 + b_1 + b_1 \cdot b_2 + \ldots$ is an unbiased estimation for the size of the tree. Despite its simplicity, this method is strikingly effective. Unfortunately, this random probing technique cannot be directly used during backtrack search, since it requires full knowledge of the tree structure. This knowledge is not available when solving a SAT problem, since the search tree is being unfold through the search. Inspired by Knuth's method, Kilby et. al. proposed two online methods to estimate the size of a search tree during backtracking search [2]: The Weighted Backtrack Estimator, which is discussed in depth in the next section, and the Recursive Estimator. The Recursive Estimator simply assumes that any unexplored right subtree is identical in size to the left subtree. Both methods are unbiased and independent of the problem or solver, but since they are both estimate the size of a complete binary search tree, they do not work directly with modern solvers and perform poorly for most satisfiable instances.
Kokotov and Shlyakhter suggested a technique which somewhat resembles RE ([4]). Their "Progress Bar for SAT Solvers" estimates the time remaining to solving a SAT instance by observing previously visited nodes. The estimate is calculated using either *historical* or *predictive* heuristics. Historical estimators use the average observed for previous nodes at the same depth, and are of two types: The *simple average* estimator uses a simple average, whilst the *weighted average* favors closer subtrees. Predictive estimators, on the other hand, are based on the size of the subproblem (e.g. number and size of the clauses).

$\langle stack, action, conflict \rangle$
$\langle \phi, assign(v_1, true), no \rangle$
$\langle \{v_1\}, assign(v_2, true), yes \rangle$
$\langle \{v_1, v_2\}, backtrack(1), no \rangle$
$\langle \phi, assign(v_3, true), no \rangle$
$\langle \{v_3\}, assign(v_4, true), no \rangle$
$\langle \{v_3, v_4\}, assign(v_5, true), yes \rangle$
$\langle \{v_3, v_4, v_5\}, backtrack(2), yes \rangle$
$\langle \{v_3\}, backtrack(1), yes \rangle$

(a) Sequence of actions



(b) Resulting proper binary search tree

Figure 3.1: An example for conversion of a sequence of search steps into a proper binary tree. In 3.1(a), every step is represented by a $\langle stack, action, conflict \rangle$ tuple, including the assignment stack before the action, the action taken, and whether the action resulted a conflict. In 3.1(b), Conflicts are denoted by numbers, while assignments appear on the corresponding edges. According to our technique, right edges represent a backtrack to the parent node and therefore are unlabeled. Note that since decisions v_2 and v_5 were backjumped over, they do not appear in the resulting binary tree.

3 WBE for Conflict Driven search

At every point in search, the WBE algorithm generates an estimation for the size of the tree, by calculating:

$$\frac{\sum_{d \in D} prob(d)(2^{d+1} - 1)}{\sum_{d \in D} prob(d)} \tag{3.1}$$

Where $prob(d) = 2^{-d}$ is related to the probability that we visit such a depth using random probing, and D is the multiset of branches lengths visited. By keeping the numerator and denominator, this estimate can be calculated in constant time and space at every backtrack. The resulting estimate is unbiased assuming we have a proper binary search tree.

Modern SAT solvers perform (conflict driven) backjumping. This creates a problem for WBE. By backjumping over nodes, we no longer have a proper binary tree. A second problem is that on backtracking to a decision level, modern SAT solvers are not forced to branch on the negated decision. We can instead branch on a new variable. Another challenge for WBE is restarts. At every restart point, a new tree is generated. Any method to estimate search cost must take these factors into account.

In order to construct a proper binary search tree representing the branching decisions of a SAT solver, and to compensate for backjumping, we observe the two atomic operations performed during search.

- *assign(v,b):* when v is a variable and b is a Boolean value. This action assigns the variable v the value b. This assignment will be kept in the next level of the search stack. After every assignment a unit propagation process takes place. The values that are assigned in this process are also considered to be assigned in this decision level.

- *backtrack(d):* backtrack back to decision level d. Unassign all variables assigned in any decision level equal to or greater than d. Any backtrack is also followed by unit propagation.

A binary tree can be generated as follows: we branch left from a node for every *assign* operation, and we branch right when we *backtrack* back to the node, even if the next assignment is at the same decision level. If we backjump over node n, this node is removed. Note that node depths in the binary tree no longer correspond with decision levels, Figure 3.1 shows an example of this technique. In Figure 3.1(a), we see a list of $\langle stack, action, conflict \rangle$ tuples, representing a sequence of actions and the resulted assignment stack. The tree in Figure 3.1(b) is the explicit proper binary tree corresponding to the same sequence of steps. Note that in both cases there are 4 conflicts, but also note that the node depths change.

Every time a backjump occurs, WBE needs to update the depths of leaves beneath this backjump. This is not possible if we keep just an accumulated sum for the denominator and numerator in the WBE prediction. Fortunately, the WBE prediction can be computed by observing two different parameters which are easy to adjust after backjumping. The first, C, is a simple counter of the nodes encountered so far in an in-order tree search (counting a node only after backtracking from its left subtree). The second, P, is the partial size of the tree explored assuming it is a complete binary tree.

At any point in search, the WBE prediction can be generated by calculating:

$$\frac{C}{P} - 1$$

Where C is the number of nodes encountered so far and:

$$P = \frac{1}{\sum_{n \in closed} (2^{d(n)+1})}$$

Where $d(n)$ is the depth of node n and *closed* is the subset of nodes in the current branch whose left child has been closed. When we backtrack from the left subtree of the root back to the root, $P = \frac{1}{2}$ and the estimate for the size of the tree is double the number of the nodes encountered so far. We now show that this new method gives the same prediction as the WBE method.

Theorem 1. *After encountering any conflict, except for the last one in the tree, the following holds:*

$$\frac{\sum_{d \in D} prob(d)(2^{d+1} - 1)}{\sum_{d \in D} prob(d)} = \frac{C}{P} - 1$$

when $prob(d) = 2^{-d}$

Proof.

$$\begin{aligned}
\frac{\sum_{d\in D} prob(d)(2^{d+1}-1)}{\sum_{d\in D} prob(d)} &= \frac{\sum_{d\in D} prob(d)2^{d+1} - \sum_{d\in D} prob(d)}{\sum_{d\in D} prob(d)} \\
&= \frac{\sum_{d\in D} prob(d)2^{d+1}}{\sum_{d\in D} prob(d)} - 1 \\
&= \frac{2\,|D|}{\sum_{d\in D} prob(d)} - 1 \\
&= \frac{C}{\sum_{d\in D} prob(d)} - 1
\end{aligned}$$

Note that $C = 2\,|D|$ as C is increased by 2 for every conflict (once for the leaf and again for the node we backtrack to). Finally, we can show by induction on the depth of the tree that:

$$\sum_{d\in D} prob(d) = \frac{1}{\sum_{n\in closed}(2^{d(n)+1})}$$

where $d(n)$ is the depth of node n and *closed* is the subset of the nodes in the current branch whose left child has been closed. □

Both C and P can be computed incrementally as we branch and backjump. Since the search tree is not kept explicitly in memory, *closed* is computed using a bit array. This increases the space and time complexity of calculating WBE by $O(d)$ where d is the maximum depth. We can avoid increasing the amortized complexity if we estimate search cost at only every $O(d)$ nodes.

Restarts create an extra challenge for WBE. Upon restarting, a new tree is generated. The search cost estimation therefore needs to change. Since WBE generate a *tree size* estimate, we can generate a cost estimation by adding the tree size estimated by WBE to the number of nodes explored until we reach a restart big enough to explore such a tree.

4 Experiments

We ran experiments with these two methods using MiniSat [1]. This is a state-of-the-art modern solver, which uses clause learning and clause deletion along with an improved version of VSIDS for variable ordering and a geometrical restart scheme. We did not, however, enable geometrical restarts.
We used two different distributions of SAT problems.

- *rand:* An ensemble of 500 satisfiable and 500 unsatisfiable randomly generated 3-SAT problems with 200 to 550 variables and a clause-to-var ratio of 4.1 to 5.0.
- *bmc:* An ensemble of software verification problems generated by CBMC[1] based on a binary search algorithm coded in C. The different examples used different array sizes and different number of loop unwindings. In

[1] http://www.cs.cmu.edu/ modelcheck/cbmc/

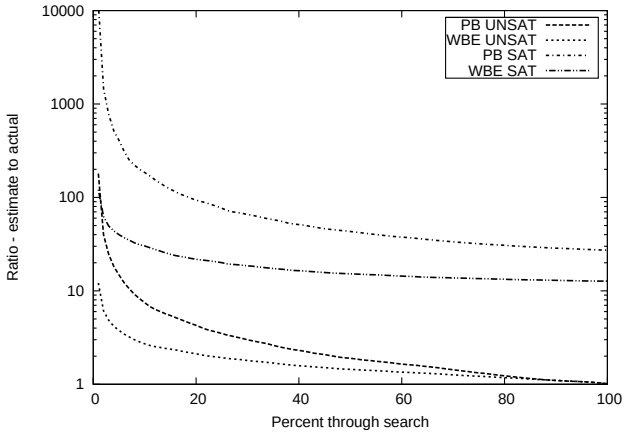


Figure 4.1: Mean ratio of WBE and PB estimates over time for the *rand* dataset.

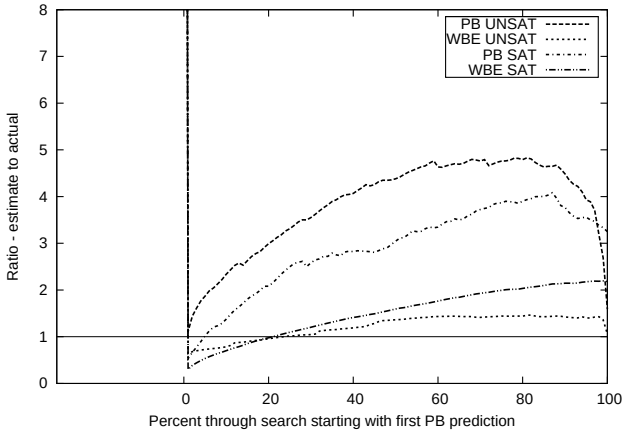


Figure 4.2: Mean ratio of WBE and PB estimates over time for the *bmc* dataset. Only starts when PB generates its first prediction.

order to generate satisfiable problems, a faulty piece of code that causes memory overflow was added to the binary search code. These problems create a very homogeneous ensemble of problems. We used 250 satisfiable and 250 unsatisfiable problems.

Since training examples can be scarce, we restricted the size of our training set to no more than 500 problems.

We compare our results with the ones obtained by the Progress Bar (PB) [4]. In order to make the comparison possible, we instrumented MiniSat with the Progress Bar. In [4], the authors proposed several different heuristics (Constant, Historical-Basic, Historical-Weighted and Clause Count). We observed similar results with all of these heuristics. We present results here for the Historical-Weighted heuristic since it performs slightly better for these data sets. We used the progress bar's default settings. Note that if the initial search is too deep, the Progress Bar may not provide any estimate.

Figure 4.1 compares the quality of the WBE prediction and the Progress Bar prediction over time, for the *rand* dataset. Both predictors return unbiased results for the unsatisfiable problems and converge to the correct value given enough time. WBE is generally more accurate than PB both for satisfiable and unsatisfiable instances. In all cases, both estimators start by over-estimating the search cost but their prediction improves with time as we backjump over nodes.
Figure 4.2 presents the same data for the *bmc* dataset. For structured problems, WBE initially over estimates search cost by a large factor (in some cases with by a factor greater than 2^{1000}). During this period the Progress Bar does not make any prediction as the tree is too deep for it to work, and the "search space left" is estimated to be 100%. At some later point in search, we often observed a sharp improvement in the accuracy of both estimators. Typically this corresponds to search backjumping over an early mistake to a node very close to the root of the tree (or the root itself). For most instances PB starts returning run-time predictions at this point. The WBE also starts returning good prediction at this point. For unsatisfiable problems in the *bmc* dataset, this point occurs after 72% of the search (on average), but it appears to occur after a smaller percentage of the search for harder instances. We found a correlation coefficient of -0.45 between the total size of the search tree and the percent through search where this improvement occurs.

5 Conclusions

We have identified some challenges in predicting the search cost to solve a SAT problem using a modern conflict driven solver in an online manner. We have presented the WBE method which observes the search tree and requires no prior knowledge of the problem distribution. We have demonstrated the effectiveness of this method on random problems, as well as on bounded model checking.

Bibliography

[1] N. Een and N. Sorensson. An extensible SAT-solver. *SAT*, 2919:502–518, 2003.

[2] Philip Kilby, John Slaney, Sylvie Thiébaux, and Toby Walsh. Estimating search tree size. In *Proceedings of the Twenty-First National Conference of Artificial Intelligence, AAAI*, 2006.

[3] D.E. Knuth. Estimating the Efficiency of Backtrack Programs. *Mathematics of Computation*, 29(129):121–136, 1975.

[4] D. Kokotov and I. Shlyakhter. Progress Bar for SAT Solvers. In *Unpublished manuscript, http://sdg. lcs. mit. edu/satsolvers/progressbar.html*, 2000.