Estimating Search Tree Size

Philip Kilby
NICTA and ANU
Canberra, Australia
Philip.Kilby@anu.edu.au

John Slaney
NICTA and ANU
Canberra, Australia
John.Slaney@anu.edu.au

Sylvie Thiébaux
NICTA and ANU
Canberra, Australia
Sylvie.Thiebaux@anu.edu.au

Toby Walsh
NICTA and UNSW
Sydney, Australia
tw@cse.unsw.edu.au

Abstract

We propose two new online methods for estimating the size of a backtracking search tree. The first method is based on a weighted sample of the branches visited by chronological backtracking. The second is a recursive method based on assuming that the unexplored part of the search tree will be similar to the part we have so far explored. We compare these methods against an old method due to Knuth based on random probing. We show that these methods can reliably estimate the size of search trees explored by both optimization and decision procedures. We also demonstrate that these methods for estimating search tree size can be used to select the algorithm likely to perform best on a particular problem instance.

Introduction

Anyone who has used a backtracking procedure will probably have observed some problem instances being solved almost immediately, and other problem instances of a similar size taking an inordinate length of time to solve. Whilst waiting for such a search procedure to finish, we might ponder a number of questions. How do I know if the search procedure is just about to come back with an answer, or has it taken a wrong turn? Should I go for coffee and expect to find the answer on my return? Is it worth leaving this to run overnight, or should I just quit as this search is unlikely ever to finish? To help answer such questions, we might wish for a search method that could estimate how long it is likely to take. In this paper, we consider how to tackle this problem.

We want to estimate the size of a particular algorithm's search tree; that is, the number of nodes that it will actually visit. We study an old method due to Knuth based on random probing. We also propose two new online methods, the weighted backtrack and recursive estimators which sit on top of a search procedure such as chronological backtracking with minimal overhead. Such methods for estimating search tree size have a number of potentially useful applications beyond informing users of how long they will have to wait. For example, they can be used to design load balancing schemes. We might, for example, give an idle processor the subproblem which is estimated to have the largest search tree. As a second example, they can be used to inform restart strategies. We might, for instance, decide to restart whenever the estimated size of the search tree is over a certain threshold. As a third example, they can be used to select the algorithm that is likely to perform best on a problem. We study such an application in this paper.

Copyright © 2006, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

Background

We shall consider systematic search procedures like chronological backtracking which can be used to solve decision problems, as well as systematic search procedures like branch and bound which can be used to solve optimization problems. We distinguish between optimization and unsatisfiability problems where we must explore all open branches, and satisfiability problems where the search may terminate with the tree only partially explored. For simplicity, we focus on binary search trees. There are several reasons for this focus. First, many of the ideas lift easily to non-binary tress. Limiting ourselves to binary trees keeps the notation simple. Second, many practical search algorithms (e.g. Davis Putnam procedures, constraint toolkits, and many TSP solvers) use binary branching. Indeed, there are theoretical results showing that binary branching can result in exponentially smaller search trees in certain situations.

Knuth's method

Knuth's method estimates N, the size of a backtrack tree as $1 + b_1 + b_1.b_2 + \ldots$ where b_i is the branching rate observed at depth i using random probing (Knuth 1975). For binary trees, if d is the depth of a random probe, then $b_i = 2$ for all $i \leq d$ and 0 otherwise, so N is $2^{d+1} - 1$. Knuth's method simply averages this estimate over multiple random probes, giving $\langle 2^{d+1}-1 \rangle$ where $\langle \ldots \rangle$ represents the average over the sample. This is an unbiased estimator: the expected value it computes is the correct tree size.

To illustrate the strengths of Knuth's method, we consider a pathological family of imbalanced search trees. Gomes *et al.* have observed that that runtime distributions for many backtracking search procedures are heavy-tailed (Gomes, Selman, & Crato 1997). They have proposed a simple model of imbalanced search trees to model such behavior. A simple instance of such a tree is where, with probability $p^i(1-p)$, we branch to depth $i + 1$ and discover the root of a complete binary tree of depth i (Figure 1). Search terminates when we have completely traversed this subtree. Note that $\sum_{i\geq 0} p^i(1-p) = 1$ as required.

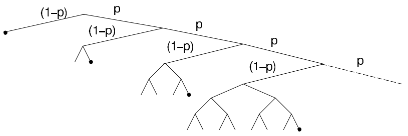

Figure 1: Imbalanced tree. Terminal nodes marked with a dot.

To ensure that the expected search tree size is finite, we assume $p < \frac{1}{2}$. This is equivalent to saying that, at any depth, our heuristic has a better than 50% chance of branching into the smaller of the two possible trees. The probability p is thus a measure of heuristic inaccuracy. The expected search tree size is given by:

$$\begin{aligned}
\langle N \rangle &= \sum_{i\geq 0} p^i(1-p)(2^{i+1}+i) \\
&= 2(1-p)\sum_{i\geq 0}(2p)^i + (1-p)\sum_{i\geq 0} ip^i \\
&= 2\frac{1-p}{1-2p} + \frac{1}{1-p}
\end{aligned}$$

For small p, this gives an expected tree size of $3(1+p) + O(p^2)$.

While Knuth's method, being an unbiased estimator, returns the correct *expected* value, the *variance* in the size of the search tree computed by random probes depends on the heuristic accuracy. If $p \geq 1/16$, we hit the heavy tail and variance in the estimated search tree size is infinite. If $p < 1/16$, then the variance is finite and is given by:

$$\begin{aligned}
\sigma^2 &= \sum_{i\geq 0} p^i(1-p)(2^{2i+2}-1)^2 - \langle N \rangle^2 \\
&= 16(1-p)\sum_{i\geq 0}(16p)^i - 8(1-p)\sum_{i\geq 0}(4p)^i + \\
&\quad (1-p)\sum_{i\geq 0}(p)^i - \langle N \rangle^2 \\
&= 16\frac{1-p}{1-16p} - 8\frac{1-p}{1-4p} + 1 - \langle N \rangle^2
\end{aligned}$$

For small p, this gives a variance of $204p + O(p^2)$. Thus, provided heuristic accuracy is large enough, random probing is successful at estimating the size of these highly imbalanced trees. This should perhaps not be too surprising since randomization and restarts works well on such trees (Gomes, Selman, & Crato 1997).

Limitations of Knuth's method

Whilst Knuth's method is very simple and surprisingly effective in practice, it has a number of limitations. Most importantly, as it is essentially an offline procedure, it requires that the search tree be available in advance of the actual search. Search algorithms which determine the tree as they go, such as SAT solvers which choose the next variable on which to branch by looking at which variables have been involved in the most nogoods, create search trees which are opaque to probing until after the event. Knuth himself notes (Knuth 1975) that his method is not suited to search trees explored by procedures like branch and bound since the bounds to be used down a particular branch are not known in advance. Another difficulty for random probing, as argued by Purdom (Purdom 1978), is that it is easily misled if the tree is very unbalanced. More systematic methods are required in such situations.

It is sometimes possible to use Knuth's method in an approximate way by substituting random decisions for the unknown ones such as the SAT solver's variable orderings. The effect of this is that the probe samples from an ensemble of trees rather than from the specific one searched. There are cases in which this can still be useful, but it destroys the unbiased nature of the estimator and in general causes performance to degrade, so again a better solution is required.

In an attempt to tackle these problems, we will propose two new online methods for estimating for search tree size that sit on top of chronological backtracking. In their different ways, they take the part of the search tree already encountered as the guide to that still to come.

Weighted backtrack estimator

Our first method is heavily inspired by Knuth's random probing method. Its cost is minimal: a (small) constant overhead at each search node. We estimate N at any point during chronological backtracking by a weighted sum of the form:

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

Where D is the multiset of branch lengths visited so far during backtracking, and $prob(d)$ equals 2^{-d} (the probability of random probing visiting such a branch). If we branch left or right at random down the tree, the expected value returned at the end of the first branch by this estimator is exact, and is thus an unbiased estimate of the search tree size.

This estimator can cope with unbalanced search trees that have been described as "problematic" for Knuth's method (Purdom 1978). Consider a tall, skinny binary tree in which at each level but the last, one child is a leaf and the other is not. If the maximum depth is n, then the total tree size is $2n+1$ nodes. We suppose that our backtracking procedure explores this tree using a random branching heuristic.

As always, the expected size of the search tree computed by Knuth's method after probing one or more branches is exact and equal to $2n+1$ nodes. Whilst a random probe is unlikely to visit a deep branch, they contribute an exponentially large term to the estimator. As a result, Knuth's method frequently over-estimates the search tree size. In fact, the variance in the search tree size computed by random probes grows as $\Theta(2^n)$ for large n. We will therefore need to average over a very large number of random probes to bring this variance down.

By comparison, the weighted backtrack estimator behaves well on these tall, skinny trees. After exploring one branch,

```
function search(node,depth)
if leaf(node) then
    if goal(node) then
        return "success"
    else return 1
else
    left[depth] := "estimated"
    result := search(left(node),depth+1)
    if result="success" then
        return "success"
    else
        left[depth] := result
        result := search(right(node),depth+1)
        if result="success" then
            return "success"
        else return 1 + left[depth] + result

function estimate(node,depth)
if leaf(node) then
    return 1
else
    if left[depth]="estimated" then
        leftsize := estimate(left(node),depth+1))
        rightsize := F(right(node))
    else
        leftsize := left[depth]
        rightsize := estimate(node(right),depth+1)
    return 1 + leftsize + rightsize
```

Figure 2: Pseudo-code for the recursive estimator of search tree size.

the expected search tree size returned by the weighted backtrack estimator is also $2n + 1$ nodes, and the variance is $\Theta(2^n)$ for large n. However, the variance quickly drops. After exploring $n + 1$ branches, we have explored the tree completely and the weighted backtrack estimator returns the correct tree size with zero variance.

Recursive estimator

Our second online method is based on a simple recursive schema. Suppose we are in the process of traversing a binary tree depth-first and from left to right. We know exactly the size of the part of the tree to the left of the current position, since we have visited it and counted the nodes, so the best estimate of the entire tree size is the sum of that (correct) figure and the best estimate of the remaining portion of the tree. Let $\mathcal{F}(n)$ be some function that estimates the size of the subtree rooted at node n of a tree. Then we can build a recursive estimator using the exact node count for the left portion of the tree and $\mathcal{F}$ for the right portion.

The pseudo-code for this estimator is given in Figure 2. To perform chronological backtracking, we call "search(root,0)". A call to this function returns "success" when it finds a leaf node which is a goal, or the number of nodes explored in proving there is no goal within the current subtree. We use a global array "left[depth]" in which to store the size of the left subtree explored at a given depth down the current branch. At any leaf node, we can get an estimate of search tree size by calling "estimate(root,0)".

For the experiments below, we used the simplest possible estimator $\mathcal{F}$: guess that the right subtree is the same size as the left subtree. That is, the line 'rightsize := $\mathcal{F}$(right(node))' in Figure 2 is instantiated to 'rightsize := leftsize'. This tends to overestimate the tree size, because the search spends most of its time in the left subtree just when that is bigger than the right one, but it is interesting that even a simple $\mathcal{F}$ leads to a useful estimator.

Maintaining the global array adds a constant time overhead to backtracking. Each call to "estimate(root,0)" takes $O(n)$ time where n is the maximum depth. Thus, if we call the recursive estimator every n nodes, we do not increase the amortized time complexity.

Clearly, when the search tree is exhausted, the size of the search tree returned by this estimator is exact. Similarly, if we branch left or right at random down the tree, the expected value returned by this method at the end of the first branch is the total tree size. The recursive estimator is thus unbiased. It can also deal with certain types of imbalance that upset the other estimators. To illustrate this, we introduce a simple pathological example. Consider a binary tree in which the left child of the root is a leaf, and the right is a complete binary tree of depth $n-1$. We suppose that our backtracking procedures explores the left child of the root before the right.

The expected search tree size computed by Knuth's method using random probing is exact and equal to $2^n + 1$ nodes. Random probing has a 50% chance to explore the short left branch. Knuth's method is therefore likely to under-estimate the search tree size. In particular, the variance in the search tree size computed by random probing grows as $\Theta(2^{2n})$ for large n. As there is such a high chance of hitting the short left branch, we need to average over very many samples to bring the variance down.

The weighted backtrack method is also misled by the first short branch. Indeed, we must explore all $2^n + 1$ nodes in the tree before the weighted backtrack method compensates for the under-estimate introduced by the first short branch. After 1 backtrack its estimate of search tree size is 3 nodes, and $\frac{2^{n-1}+2^{n+1}-1}{2^{n-1}+1}$ nodes after 2 backtracks (which tends to 5 nodes as $n \to \infty$), and $\frac{2^{n-1}+j(2^{n+1}-1)}{2^{n-1}+j}$ (which grows as $1 + 4j$ for $j << 2^{n-1}$) after $j + 1$ backtracks where $j > 0$. The estimate of the search tree size monotonically increases towards the exact size of $2^n + 1$. However, it does not correctly estimate the tree size until the whole tree is explored.

By comparison, the recursive estimator quickly compensates for the first short branch. Its estimate of the size of the search tree after 1 backtrack is 3 nodes. However, after just 2 backtracks, its estimate is 2^n+1 nodes, where the estimate stays till all $2^n + 1$ nodes of the search tree are explored.

Experimental results

We now study the effectiveness of these methods at estimating search tree size on both decision and optimization problems.

Unsatisfiable decision problems

We looked first at unsatisfiable random 3-SAT problems with 300 variables and 1260 clauses. These are problems at the phase transition in hardness. This problem set is referred to as 3-unsat. We tested 200 instances, and computed the ratio of estimated tree size over actual. The SAT solver used was *satz215* (Li 1999), a DPLL-based solver which uses lookahead in choosing the next variable on which to branch, but which does not learn nogoods. Figure 3 shows the evolution of the estimate as the search proceeds. The notion of "percent through search" makes no literal sense for probing methods such as Knuth's: the intended reading is that "100%" is reached when the number of probes equals the number of branches in the tree.

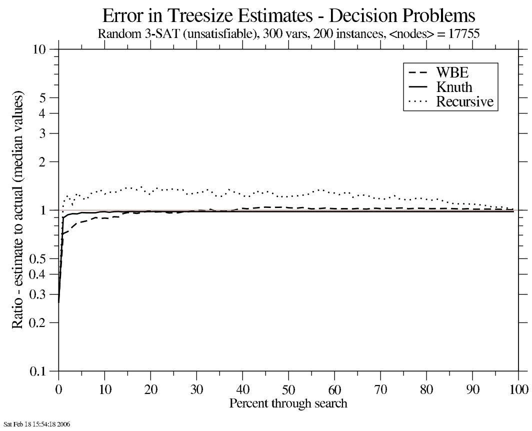

Figure 3: Treesize estimates, unsatisfiable decision problems "3-unsat"

We also examined performance on structured, unsatisfiable decision problems from SATLIB. In Figure 4 we show the result obtained on the "pigeon-hole" problem (size 8), and also the average value obtained over 4 instances of circuit fault analysis problems (the "BF" data set).

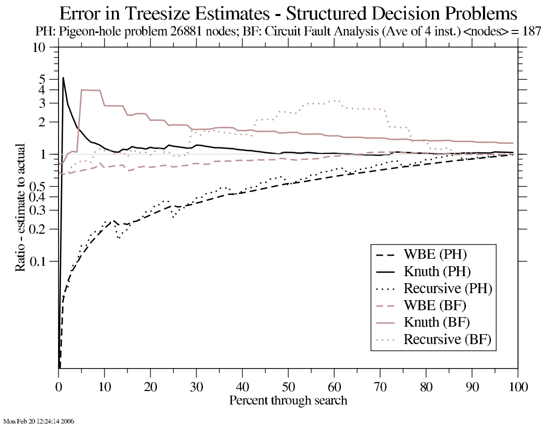

Figure 4: Treesize estimates, unsatisfiable decision problems "hole-8" and "BF"

Optimization problems

We now turn to predicting the size of a branch and bound tree. We consider the travelling salesperson problem (TSP). Our branch and bound algorithm uses depth-first search, exploring the node with the smallest lower bound at each iteration. The lower bound is calculated using a Lagrangian Relaxation based on 1-trees, which are a form of Minimum Spanning Tree (MST). The upper bound is calculated by a single, deterministic run of Or-opt testing all forward and reverse moves of blocks of size $n/2$ down to 1, where n is the number of cities. Branching forces an edge into the solution on one side, and out of the solution on the other. The MST bound is enhanced by eliminating all other edges into a node if two incident edges are already forced into the solution. Finally, the edge to force in or out at each iteration is chosen randomly from the (unforced) edges in the current upper bounding tour. Tests were conducted on problems with 50 cities distributed uniformly and randomly in the 2-D plane. This data set is referred to as "tsp". The results are shown in Figure 5.

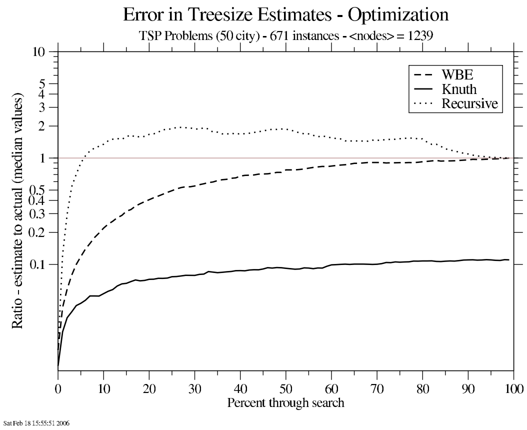

Figure 5: Treesize estimates, optimization problems "tsp"

The estimates for Knuth's method are synthetic results generated after we had solved the problems. As we argued before, Knuth's method cannot be applied directly to branch and bound search trees. For the purposes of comparison, we saved the search tree generated by our TSP algorithm, and randomly sampled it *a-postiori*. Even with this help, Knuth's method performs very badly. We conjecture this is because the search trees are very lop-sided. Because of the way bounding occurs, many nodes have one leaf and one non-leaf child. Random probes are very likely therefore to terminate at a shallow depth.

Satisfiable decision problems

We end this empirical study with the most challenging class of problems: satisfiable decision problems. With such problems, search can terminate even if there are many open nodes down the current branch. We do not therefore expect estimation methods to work as well as on complete search. We used satisfiable, random 3-SAT problems with 300 variables and 1260 constraints. This problem set is called "3-sat".

All methods naturally over-estimated the tree size, but the estimates were fairly consistent. The results are shown in Figure 6. Note that this has a log y scale.

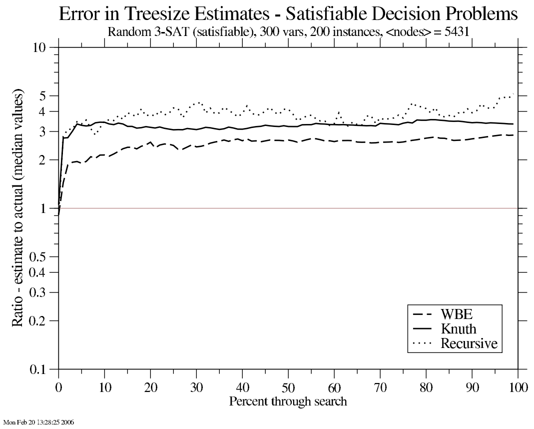

Figure 6: Treesize estimates, satisfiable decision problems "3-sat"

Midpoint prediction

To illustrate one possible application of our new online methods, we predict the midpoint of search. That is, we flag when the number of nodes searched first exceeds half the estimated size of the search tree. We cannot give a direct comparison with Knuth's method as it is not online. To permit some sort of comparison, we allowed Knuth's method one probe for each backtrack of the algorithm, though this would in effect more than double the time required. We measured the ratio of the number of nodes when the half-way point was predicted reached to the total size of search tree. Thus, 0.5 is the desired value. Table 1 gives the results for tests on the problem classes described above.

Problem	Method	Median	Mean	SD	Count	Miss
3-unsat	Knuth	0.49	0.50	0.06	200	0
3-unsat	WBE	0.51	0.50	0.13	200	0
3-unsat	Recursive	0.53	0.52	0.13	200	0
hole-8	Knuth	0.44	0.44	0.00	1	0
hole-8	WBE	0.12	0.12	0.00	1	0
hole-8	Recursive	0.12	0.12	0.00	1	0
BF	Knuth	0.43	0.47	0.04	2	2
BF	WBE	0.41	0.60	0.25	3	1
BF	Recursive	0.41	0.62	0.24	3	1
tsp	Knuth	0.07	0.15	0.20	599	72
tsp	WBE	0.18	0.29	0.27	600	71
tsp	Recursive	0.56	0.56	0.21	600	71
3-sat	Knuth	0.44	0.55	0.22	70	130
3-sat	WBE	0.42	0.47	0.24	75	125
3-sat	Recursive	0.46	0.52	0.23	67	133

Table 1: Midpoint prediction

Count is the number of instances for which a prediction was produced. "Miss" gives the number of instances for which search finished before the midpoint was predicted. The premature predictions in Hole-8 and tsp are expected given the under-estimation evident in the corresponding figures above. However, the success in prediction for *satisfiable* problems is surprising. It appears that if we do not get lucky and finish very quickly, we can make a relatively good prediction of the search tree size.

Algorithm selection

Our second application is selecting the algorithm most likely to perform well on a problem instance. One of the features of this problem is that estimation methods only need to rank accurately. A method can, for example, consistently under-estimate search tree size and still perform well at selecting the best algorithm. We consider the *satz215* algorithm (Li 1999) which is deterministic, and uses lexicographic ordering to break ties. The algorithm therefore performs very differently on the same problem presented with renamed variables. The algorithm *ShuffleSatz* given in Figure 7 exploits this variance by selecting a lexicographic ordering that is estimated to give a smaller tree size. *ShuffleSatz* can use either the Knuth, WBE, or Recursive estimators.

```
Generate 10 random lexicographical orderings
For each ordering
    Run satz215 using the ordering to break ties
    Stop after 50 backtracks (or 50 probes for Knuth)
    Calculate WBE, Recursive, or Knuth estimate
Complete search with the ordering that gives the
    lowest estimate
```

Figure 7: ShuffleSatz

Table 2 shows the results of applying *ShuffleSatz* to the 86 difficult problem instances in the "3-sat" data set that required more than 2000 backtracks to solve. We report total search effort in terms of backtracks required to find a satisfiable assignment, plus the number backtracks or probes used in selecting the best lexicographical ordering. The "Mean" column gives the mean number of backtracks required by the lexicographical ordering chosen by the estimator. Any of the estimation methods is able to speed up the *satz215* algorithm. As there are some "magic numbers" in the definition of *ShuffleSatz*, more work is required to find robust values for these parameters.

Method	Mean Backtracks	Total Backtracks	Percent Improve
Orig	5239	450587	0
Knuth	3580	350923	22
WBE	3852	374229	17
Recursive	3951	382820	15

Table 2: Number of backtracks over "3-sat" data set – *ShuffleSatz* compared to vanilla *satz215*

Related work

Knuth demonstrated the effectiveness of random probing for estimating search tree size on the problem of enumerating uncrossed knight's tours (Knuth 1975). As mentioned before, he argued that random probing cannot be used for procedures like branch and bound. However, he suggested that it could be used to estimate the work needed to test a bound for optimality. To reduce variance, Purdom proposed a generalization of Knuth's method that uses partial backtracking, a beam-like search procedure (Purdom 1978). Chen has proposed a generalization of Knuth's method called heuristic sampling that stratifies nodes into classes (Chen 1992). Knuth's method classifies nodes by depth. Chen applied his method to estimate the size of depth-first, breadth-first, best-first and iterative deepening search trees.

Allen and Minton adapted Knuth's method to constraint satisfaction algorithms by averaging over the last ten branches sampled by backtracking (Allen & Minton 1996). They used these estimates to select the most promising algorithm. Lobjois and Lemaitre also used Knuth's method to select the branch and bound algorithm likely to perform best (Lobjois & Lemaitre 1998). Cornuéjols, Karamanov and Li predict the size of a branch and bound search tree by using the maximum depth, the widest level and the first level at which the tree is no longer complete to build a simple linear model of the branching rate. (Cornuéjols, Karamanov, & Li 2006). Kokotov and Shlyakhter measure the progress of Davis Putnam solvers using an estimator somewhat similar to our recursive estimator (Kokotov & Shlyakhter 2000). Finally, Aloul, Sierawski and Sakallah use techniques based on decision diagrams to estimate progress of a satisfiability solver (Aloul, Sierawski, & Sakallah 2002).

Musick and Russell abstracted the search space to construct a Markov model for predicting the runtime of heuristic search methods like hill climbing and simulated annealing (Musick & Russell 1992). Slaney, Thiébaux and Kilby have used the search cost to solve easy decision problems away from the phase boundary as a means to predict the cost of solving the corresponding optimization problem (Slaney & Thiébaux 1998; Thiébaux, Slaney, & Kilby 2000). They showed that good estimates could be found using a small fraction of the time taken to prove optimality.

Finally, statistical techniques have been applied to predict the size of search trees. Horvitz and colleagues have used Bayesian methods to predict runtimes of constraint satisfaction algorithms based on wide range of measures (Horvitz *et al.* 2001; Kautz *et al.* 2002; Ruan, Horvitz, & Kautz 2002). Such predictions are then used to derive good restart strategies. Leyton-Brown and Nudelman used statistical regression to learn a function to predict runtimes for NP-hard problems (Leyton-Brown & Nudelman 2002). Their methods use both domain independent features (like the quality of the integer relaxation) and domain dependent features.

Conclusion

We have proposed two online methods for estimating the size of a backtracking search tree. The first is based on a weighted sample of the branches visited by chronological backtracking. The second assumes that the future (an unexplored right branch) will look much like the past (an explored left branch). We have compared these methods against a method due to Knuth based on random probing. All three are unbiased estimators; the expected value they return is the total tree size. However, our online methods offer a number of advantages over methods based on probing. They can, for example, be used within branch and bound procedures. Initial results are promising. On both structured and random problems, we can often estimate search tree size with good accuracy. We can, for instance, select the algorithm likely to perform best on a given problem.

References

Allen, J., and Minton, S. 1996. Selecting the right heuristic algorithm: Runtime performance predictors. In *Proc. of the 11th Canadian Conf. on Artificial Intelligence*, 41–52.

Aloul, F.; Sierawski, B.; and Sakallah, K. 2002. Satometer: How much have we searched? In *Design Automation Conf.*, 737–742. IEEE.

Chen, P. 1992. Heuristic sampling: a method for predicting the performance of tree searching programs. *SIAM Journal of Computing* 21(2):295–315.

Cornuéjols, G.; Karamanov, M.; and Li, Y. 2006. Early estimates of the size of branch-and-bound trees. *INFORMS Journal on Computing* 18(1).

Gomes, C.; Selman, B.; and Crato, N. 1997. Heavy-tailed distributions in combinatorial search. In Smolka, G., ed., *Proc. of Third Int. Conf. on Principles and Practice of Constraint Programming (CP97)*, 121–135. Springer.

Horvitz, E.; Ruan, Y.; Gomes, C.; Kautz, H.; Selman, B.; and Chickering, D. 2001. A bayesian approach to tackling hard computational problems. In *Proc. of 17th UAI*, 235–244.

Kautz, H.; Horvitz, E.; Ruan, Y.; Gomes, C.; and Selman, B. 2002. Dynamic restart policies. In *Proc. of the 18th National Conf. on AI*, 674–681. AAAI.

Knuth, D. 1975. Estimating the efficiency of backtrack programs. *Mathematics of Computation* 29(129):121–136.

Kokotov, D., and Shlyakhter, I. 2000. Progress bar for sat solvers. Unpublished manuscript, http://sdg.lcs.mit.edu/satsolvers/progressbar.html.

Leyton-Brown, K., and Nudelman, S. 2002. Learning the empirical hardness of optimization problems: the case of combinatorial auctions. In *Proc. of 8th Int. Conf. on Principles and Practice of Constraint Programming (CP2002)*. Springer.

Li, C. M. 1999. A constrained-based approach to narrow search trees for satisfiability. *Information processing letters* 71:75–80.

Lobjois, L., and Lemaitre, M. 1998. Branch and bound algorithm selection by performance prediction. In *Proc. of 15th National Conf. on Artificial Intelligence*, 353–358. AAAI.

Musick, R., and Russell, S. 1992. How long will it take. In *Proc. of the 10th National Conf. on AI*, 466–471. AAAI.

Purdom, P. 1978. Tree size by partial backtracking. *SIAM Journal of Computing* 7(4):481–491.

Ruan, Y.; Horvitz, E.; and Kautz, H. 2002. Restart policies with dependence among runs: A dynamic programming approach. In *Proc. of 8th Int. Conf. on Principles and Practice of Constraint Programming (CP2002)*. Springer.

Slaney, J., and Thiébaux, S. 1998. On the hardness of decision and optimisation problems. In *Proc. of the 13th ECAI*, 244–248.

Thiébaux, S.; Slaney, J.; and Kilby, P. 2000. Estimating the hardness of optimisation. In *Proc. of the 14th ECAI*, 123–127.