Learning to Select Branching Rules in the DPLL Procedure for Satisfiability

Michail G. Lagoudakis MGL@CS.DUKE.EDU
Department of Computer Science, Duke University, Durham, NC 27708, USA

Michael L. Littman MLITTMAN@RESEARCH.ATT.COM
Shannon Laboratory, AT&T Labs Research, Florham Park, NJ 07932, USA

Abstract

The DPLL procedure is the most popular complete satisfiability (SAT) solver. While its worst case complexity is exponential, the actual running time is greatly affected by the ordering of branch variables during the search. Several branching rules have been proposed, but none is the best in all cases. This work investigates the use of automated methods for choosing the most appropriate branching rule at each node in the search tree. We consider a reinforcement-learning approach where a value function, which predicts the performance of each branching rule in each case, is learned through trial runs on a typical problem set of the target class of SAT problems. Our results indicate that, provided sufficient training on a given class, the resulting strategy performs as well as (and, in some cases, better than) the best branching rule for that class.

1. Introduction

Repetitive tasks, such as adding numbers, counting votes, or solving instances of a specific problem are common for both people and computers. It is true that nowadays most such tasks are performed by machines that, unlike humans, do not get tired of doing the same thing over and over again. However, unlike machines, people tend to improve their performance on a repetitive task as they accumulate more and more experience with the task. A computer is, in general, doomed to taking the very same steps in solving a problem no matter whether it is its first or millionth time. The human quality of learning, if embedded in a machine, would have a significant impact on the way we program, think of, and use computers.

Our work proposes to combine human knowledge with machine power: the human programmers provide a set of different algorithms for a given problem and the machine undertakes the task of learning over time how to combine the algorithms to improve its performance. Our earlier work on algorithm selection (Lagoudakis & Littman, 2000) attacked recursive algorithms for sorting and order-statistic selection. This paper extends our approach to backtracking algorithms for satisfiability (SAT), and discusses our initial experimental efforts. Solving computationally demanding problems, such as SAT, is an area in which learning to do algorithm selection can have a significant impact. Not only do numerous algorithms exist for special cases of such problems, but even a modest speedup in solving such problems is highly desirable.

Section 2 provides an overview of the SAT problem and its counting variation and outlines the basic algorithm for solving them. The set of branching rules that we considered is described in Section 3 and the learning algorithm is covered in Section 4. Experimental results are included in Section 5, while Section 6 discusses the difficulties we faced along the way and future plans of this work.

2. SAT, #SAT, and DPLL

A Boolean formula in conjunctive normal form (CNF) consists of n Boolean variables $x_1, x_2, ..., x_n$ and m clauses $C_1, C_2, ..., C_m$. Each variable can take a TRUE or FALSE value. The value of the complement $\bar{x}_i$ of some variable x_i is the opposite of x_i. Each clause is a disjunction of distinct literals, where a literal is a single variable x_i itself or its complement $\bar{x}_i$. A clause is satisfied iff at least one of its literals takes a TRUE value. An empty clause is by definition unsatisfiable. A Boolean formula F is a conjunction of (nonempty) clauses, and it is satisfiable if there is an assignment to the variables that satisfies all the clauses. An empty formula is by definition satisfiable. SAT refers to the problem of finding a satisfying assignment for a given formula. If no such assignment exists, the formula is unsatisfiable.

The example SAT formula below consists of 5 variables and 8 clauses. It is satisfiable and $(x_1, x_2, x_3, x_4, x_5) =$

$(1, 1, 0, 1, 0)$ is a satisfying assignment:

$$(x_1 + x_2 + \bar{x}_3)(\bar{x}_2 + x_4)(\bar{x}_1 + \bar{x}_5)$$
$$(\bar{x}_1 + x_2 + x_3 + \bar{x}_4 + x_5)(\bar{x}_3)(x_2 + x_5)(\bar{x}_5)(x_1).$$

The number of assignments that satisfy a Boolean formula can be anywhere between 0 (unsatisfiable) and 2^n (tautology—satisfied by all assignments), where n is the number of variables in the formula. #SAT refers to the problem of finding the number of satisfying assignments for a given formula.

Both SAT and #SAT are hard problems. SAT is NP-HARD and indeed it is the first problem known to be NP-COMPLETE. In general, #SAT represents a harder problem compared to SAT in the sense that any algorithm that solves #SAT (exact number of satisfying assignments) can be used to solve SAT (is there at least one satisfying assignment?). It is #P-complete.

The problem of counting satisfying assignments is of interest in artificial intelligence because it is equivalent to problems in network reliability (Valiant, 1979) and belief network inference (Roth, 1996). An algorithm for #SAT has been used to solve probabilistic planning problems at state-of-the-art speeds (Majercik & Littman, 1998).

The Davis-Putnam-Logemann-Loveland (DPLL) procedure (Davis et al., 1962) is a search method for solving SAT problems. DPLL explores the space of partial assignments in a systematic way that eliminates the ones that lead to a contradiction, and extends incrementally the promising ones. A partial assignment can be extended by selecting an unassigned variable and trying in turn the two possible assignments (TRUE, FALSE) to that variable (*branching*), thus creating two new partial assignments. A reduced formula is created in each case; a TRUE assignment to x_i eliminates all clauses that contain the literal x_i (these clauses are satisfied) and all appearances of $\bar{x}_i$ from the remaining clauses. A FALSE assignment has the symmetric effect. The same procedure is then applied recursively to each reduced formula.

Two key operations of the DPLL procedure are the *unit propagation* and the *purification* steps. Unit propagation extends a partial assignment in the presence of unit clauses (clauses with a single literal). Such clauses can only be satisfied by a specific assignment to the corresponding variable. The complementary assignment would lead to a contradiction and therefore can be safely ignored. The elimination of a variable can create new unit clauses, and thus unit propagation eliminates variables by repeated passes until there is no unit clause in the formula. Purification applies when a variable x_i appears in the formula purely in positive (x_i) or negative ($\bar{x}_i$) form. Such variables can be safely eliminated by assigning TRUE in the positive case and FALSE in the negative one. Purification is applied repeatedly as pure variables might be created after each purification pass.

The DPLL procedure is given in a more algorithmic format below. Notice that purification and branching are implemented by triggering the appropriate unit propagations.

```
DPLL(F)
  if (F contains an empty clause)
    return "unsatisfiable"
  if (F is empty)
    output current assignment
    return "satisfiable"
  /* Unit Propagation */
  if (F contains a unit clause {l})
    Create F' from F by eliminating all clauses that
         contain l and all appearances of l̄
    return DPLL(F')
  /* Purification */
  if (F contains a pure literal l)
    return DPLL(F ∪ {l})
  /* Branching */
  *** Select a free literal l ***
  if (DPLL(F ∪ {l}) is "satisfiable")
    return "satisfiable"
  else
    return DPLL(F ∪ {l̄})
```

The DPLL procedure can be modified to solve the #SAT problem (Majercik & Littman, 1998; Birnbaum & Lozinskii, 1999). For each partial assignment that is found to be satisfying, there exist 2^k possible extensions to a full assignment, where k is the number of free variables. Thus, by exploring the whole space of partial assignments, it is possible to enumerate all possible satisfying assignments. Note that purification cannot be used in this case, since there might be assignments where pure variables can take either value. The modified DPLL for #SAT is given below.

```
#DPLL(F)
  if (F contains an empty clause)
    return 0
  if (F is empty)
    return 2^k, k=number of free variables
  /* Unit Propagation */
  if (F contains a unit clause {l})
    Create F' from F by eliminating all clauses that
         contain l and all appearances of l̄
    return #DPLL(F')
  /* Branching */
  *** Select a free literal l ***
  return #DPLL(F ∪ {l}) + #DPLL(F ∪ {l̄})
```

3. Branching Rules for (#)DPLL

Although the worst-case complexity of the DPLL procedure is exponential, the actual running time can be greatly affected by the choices of free literals in the branching step. It is possible to reduce the size of the search tree that DPLL and #DPLL explore by orders of magnitude if variables are chosen in the appropriate order.

Despite the fact that finding an optimal ordering is computationally difficult, a significant amount of research has been devoted to the invention of branching rules that pick literals in the branching step of DPLL. They work fairly well in practice compared to a random strategy, but provide no guarantees of optimality.

The branching rules, in general, compute some a $score(l)$ for each free literal l and $Score(v)$ for each free variable v. In most cases, the scores for complementary literals are combined in some way to balance the two branches of the search; if one path of search (say, l) is not fruitful, the other one ($\bar{l}$) has to be explored[1]. In our work, we used

$$Score(x) = score(x) + score(\bar{x}).$$

The variable x that maximizes the score is selected. Whether to branch on x or $\bar{x}$ first is decided in favor of the literal with the maximum individual score. This last decision affects only the DPLL procedure—#DPLL has to explore both branches anyway, so the order does not matter.

In this paper, we make use of seven branching rules, listed below. Hooker and Vinay (1995) and Li and Anbulagan (1997) provide extensive reviews of branching rules.

MAXO : This rule selects the literal with the maximum number of occurrences in the formula:

MAXO(l) = number of occurrences of l in the formula.

The idea is that splitting on such a choice will have a wide-spread effect in the formula.

MOMS : The MOMS rule is similar to MAXO, but it counts only occurrences of literals in minimum size clauses:

MOMS(l) = occurrences of l in minimum size clauses.

The rationale is that minimum size clauses play a more important role during the search as they can reveal a contradiction and/or enable unit propagations quickly.

MAMS : This is a novel combination of the previous two rules:

$$\text{MAMS}(l) = \text{MAXO}(l) + \text{MOMS}(\bar{l}).$$

The idea is that it is desirable to satisfy as many clauses as possible (MAXO(l)), but also to create as many clauses of minimum size as possible (MOMS($\bar{l}$)).

JW : The Jeroslaw-Wang rule combines the ideas behind MAXO and MOMS using exponential weighting:

$$\text{JW}(l) = \sum_{j,\, l \in C_j} 2^{-n_j}.$$

where n_j is the number of literals in clause C_j. Smaller clauses have more weight than the larger ones.

UP : This rule probes the search by making a trial assignment to each free literal and counting the number of triggered unit propagations due to that assignment:

UP(l) = # of unit props triggered by setting l=TRUE.

The more unit propagations the better, since each unit propagation eliminates one variable.

GUP : This is a greedy version of the previous rule. During the trial assignments it might be discovered that some assignment to a literal leads to contradiction or satisfaction (through the series of triggered unit propagations). If such a literal is found, it is immediately selected to branch on. Otherwise, GUP scores literals the same way UP does.

SUP : This is a selective version of UP. Due to the huge computational cost of the UP rule, SUP runs first the four inexpensive rules (MAXO, MOMS, MAMS, and JW), which suggest up to four distinct literals and then it selects among them using the UP scoring function. Thus, the cost of trial assignments and unit propagations is paid only for a fixed number of free literals (the most promising ones) and not for all of them.

4. Learning to Select Branching Rules

Algorithm selection is the problem of selecting the most efficient algorithm among equivalent ones for a given problem instance (Rice, 1976). In its original form, algorithm selection involves only a one-shot decision—the "best" algorithm is selected and then applied to the instance with no further decision making. Lagoudakis and Littman (2000) extended algorithm selection to cases that involve recursive computations. When recursive algorithms are included in the set, every time a recursive call is made, the algorithm selection process can be invoked to select any of the available algorithms. A key advantage of this approach is that the combined (hybrid) algorithm that results from the multiple decisions has the potential to perform better than

[1] This is particularly true for #SAT, where both branches must be explored anyway.

any of the individual algorithms. On the other hand, the decision-making task becomes harder, as it involves a sequence of decisions.

A natural approach to the recursive algorithm selection problem is to use the Markov decision processes (MDP) framework and ideas from reinforcement learning (RL). In this framework, an agent seeks an optimal policy—a function that selects actions in each possible state of the process—with the objective of minimizing the total expected cost. The process evolves over the state space according to the dynamics of the system and the actions taken by the agent. The agent is reinforced by a signal that indicates the immediate cost of each transition.

For the algorithm-selection problem, the state consists of a description of the current instance (or subinstance) under consideration in terms of some features (e.g. size). The actions are the available algorithms for the problem. Choosing a non-recursive algorithm results in a transition to the final state (problem solved) and the cost for such a choice is the execution time taken for that transition. On the contrary, choosing a recursive algorithm results in a transition to one or more states (one for each recursive call) from where new decisions can be made. The cost paid is the execution time taken (excluding time taken in recursive calls). The goal is to minimize the total expected cost, which by definition is the total execution time for the particular instance.

The mainstream approach for solving such problems is the construction (either by computation or learning) of a value function, such as $Q(s, a)$, which "predicts" the expected total cost when taking action a in state s and acting optimally thereafter. In most cases, the state space is fairly big and an explicit (tabular) representation of the value function would be rather expensive (in terms of both storage and computation/learning). For that reason, parametric approximators are often used to represent the value function and the problem becomes that of finding the set of parameters that maximizes the accuracy of the approximator. A common class of approximators is the so called linear architectures, where the value function is approximated as a linear weighted combination of basis functions (features) $Q(s, a) = \boldsymbol{\phi}(s, a)^\intercal \mathbf{w}$, where $\mathbf{w}$ are the weights (parameters). Linear architectures are popular as there are several well-studied methods (e.g. Least-Squares projection) for determining the appropriate parameters (Bradtke & Barto, 1996; Boyan, 1999).

4.1 Learning Setup for #DPLL branching rules

The branching rules in the #DPLL procedure[2] can be thought of as alternatives available each time the algorithm must branch. They are equivalent in the sense that success or failure of the search does not depend on them, but they can have a tremendous impact on the size of the search tree and therefore on the running time of #DPLL. None of these branching rules is best over all classes of SAT problems; their performance is dependent on the features of the current instance and the problem distribution from which they are drawn. Given that each branching node of #DPLL corresponds to a #SAT subinstance, it is plausible to be able to select the "best" branching rule for that node, based on features of that particular subinstance.

Therefore, to conform with the MDP terminology, the state of the process consists of features of the current #SAT (sub)instance; for example, number of variables, number of clauses, number of literals, etc. The main requirement on these features is that they should carry information about the performance of the branching rules; a varying value of a feature should result in varying performance for some (or all) of the branching rules. Further, these features have to be easily computable to minimize overhead.

All branching rules are available as possible actions at each branching node. No matter which branching rule is chosen, search continues in two directions (l and $\bar{l}$ branches). For each direction, a series of unit propagations is performed before arriving at a node of the following kind:

Branching Node : There are no unit clauses and unit propagation cannot continue. This is a new state where a new decision has to be made.

Contradiction Node : The current partial assignment causes a contradiction in the formula. Search along this branch is terminated here. This is an absorbing state of the MDP.

Satisfaction Node : The current partial assignment satisfies the entire formula. Search along this branch is terminated and the number of satisfying assignments is computed. This is again an absorbing state of the MDP.

Therefore, taking an action in a state results in two (possibly absorbing) new states. This one-to-two state transition violates the standard MDP definition, but it can be thought of as cloning the MDP and creating one copy for each transition. The two copies continue independently thereafter.

[2] From this point on, we focus on the #SAT problem and the associated #DPLL procedure, as they represent the most general case and fit within our goal of minimizing the *entire* search tree.

The immediate cost paid for choosing a branching rule is computed in two different ways. The total number of search nodes that results from unit propagations, between the original and the two new states, is the node cost paid for the decision at the original node. That is the immediate cost for choosing the branching rules MAXO, MOMS, MAMS, and JW. The other three branching rules perform trial partial searches to determine the branching variable. The total number of nodes created during these trial searches is added to the above node cost for branching rules UP, GUP, and SUP. The definition of the cost function implies that the total accumulated cost after a complete run of the DPLL procedure will be the size of the entire search tree (in terms of nodes) plus the total count of nodes created during the trial searches. Minimizing this total cost is a way to minimize the total running time, which is proportional to the total number of nodes explored.

4.2 State Representation

The state s of the process is a description of the SAT (sub)instance under consideration, in terms of some easily computable features. Candidate features include, but are not limited to, the number of variables, number of clauses, number of literals, minimum size of clauses, number of minimum size clauses, ratio of variables to clauses, etc. Unfortunately, the size of the state space grows exponentially with the number of features used, and, therefore, a compromise has to be made as to which features are more informative and will be part of the state. The best we can hope for is a very abstract characterization of the instance.

A series of experiments and visualizations of gathered data were performed to determine the relevance of each candidate feature to our problem. Different combinations of features were tried and tests were performed on SAT instances from different classes. These tests suggested that the only really useful feature is the number of variables n in the formula. This is not so surprising, considering that the size of the problem instance is always a crucial factor in the expected amount of computational resources needed to solve that instance. In our particular case, even though the performance of a branching rule is not a pure function of n, useful information can be gained, like the rate of increase of cost for different sizes. What is surprising is that adding features in the state representation did not improve, and sometimes worsened, performance.

For the rest of this paper, our state representation for a given SAT instance will consist of the number of variables n in the instance only. Even such a simple representation can take us a good way toward our goal. Other possibilities are discussed at the end of the paper.

4.3 Value Function Approximation

To accomplish the goal of minimizing the search tree of the #DPLL procedure, we seek to learn the state-action value function $Q(n, a)$ that "predicts" the expected total node cost when taking action a in state n and acting optimally thereafter. In general, for any fixed action a, $Q(n, a)$ is exponential in n. In fact, after extensive experimentation with learning tabular representations of $Q(n, a)$ under a variety of problem distributions, we found that $Q(n, a)$ can be very well approximated by the following parametric form[3]:

$$\hat{Q}(n,a) = \begin{cases} 0 & n = 0, \\ 2^{p^{(d)}(n,\mathbf{w}^{(a)})} & n > 0. \end{cases}.$$

Here, $p^{(d)}(n, \mathbf{w}^{(a)})$ is a polynomial in n of degree d with coefficients $\mathbf{w}^{(a)}$ and no constant term:

$$p^{(d)}(n, \mathbf{w}^{(a)}) = \sum_{i=1}^{d} w_i^{(a)} n^i$$

This approximation is certainly nonlinear, but by taking the logarithm it can be approximated by a linear architecture (for $n > 0$)

$$\log_2 \hat{Q}(n,a) = p^{(d)}(n, \mathbf{w}^{(a)}) = \sum_{i=1}^{d} w_i^{(a)} \boldsymbol{\phi}_i(n),$$

where the basis functions are $\boldsymbol{\phi}_i(n) = n^i$. Given this approximation, the goal is to find/learn the best set of parameters $\mathbf{w}^{(a)}$ that will make the approximation more accurate.

4.4 Learning Algorithm

Our learning algorithm is a combination of Temporal Difference learning and Least-Squares. For each action a, a $d \times d$ matrix $A^{(a)}$ and a vector $b^{(a)}$ are maintained, where d is the number of basis functions used. The parameters $\mathbf{w}^{(a)}$ are computed as the solution of the system $A^{(a)}\mathbf{w}^{(a)} = b^{(a)}$. $A^{(a)}$ and $b^{(a)}$ are updated incrementally in a way that the solution $\mathbf{w}^{(a)}$ is the least-squares solution to our approximation. All entries in $A^{(a)}$ and $b^{(a)}$ are initially set to 0.

Consider a sample transition (n, a, n_1, n_2, c)—in state n, action (branching rule) a was chosen causing transitions to states n_1 and n_2 with total node cost c. Note that for transitions to branching nodes, n_1 and/or n_2 are greater than 0, but for contradiction and/or satisfaction nodes, n_1 and/or n_2 are set to 0 (the absorbing state). The new sample $\hat{q}(n, a)$ for $\hat{Q}(n, a)$ is then computed using the Bellman optimality equation

$$\hat{q}(n,a) = c + \min_{a'} \left\{ \hat{Q}(n_1, a') \right\} + \min_{a'} \left\{ \hat{Q}(n_2, a') \right\}.$$

[3]Note that the base 2 of the exponential is only a convenient choice. Any other base could be used instead; the net change would be a multiplicative constant in the coefficients $\mathbf{w}^{(a)}$.

The values $\hat{Q}(n_1, a')$ and $\hat{Q}(n_2, a')$ are computed straight from the current approximation. Finally, the new sample $\hat{q}(n, a)$ is inserted in $A^{(a)}$ and $b^{(a)}$ as follows:

$$A^{(a)} = A^{(a)} + \lambda(n) * \boldsymbol{\phi}(n)\boldsymbol{\phi}(n)^\intercal,$$

$$b^{(a)} = b^{(a)} + \lambda(n) * \boldsymbol{\phi}(n)\hat{q}(n, a).$$

$\lambda(n)$ is a weight factor that is used to determine the importance of each sample in the set. Our reason for using reweighting is that during a typical run of the #DPLL procedure there are exponentially more samples from the region around the leaves of the tree compared to samples from the region around the root of the tree. This inherent weighting can bias the least-squares solution. Our reweighting scheme attempts to cancel this effect by defining $\lambda(n)$ as

$$\lambda(n) = 2^{n/N} - 1,$$

where N is the total number of variables in the original SAT instance the samples are taken from.

The use of the current approximation in determining new samples can be problematic, especially at the very beginning of the learning process when the approximator is inaccurate. A way around this problem is to go through two learning phases. In the first phase, the value functions $\hat{Q}^{(a)}(n)$ for each fixed policy consisting of a single branching rule (action) is learned. In this case, the new samples are computed as

$$\hat{q}^{(a)}(n) = c + \hat{Q}^{(a)}(n_1) + \hat{Q}^{(a)}(n_2).$$

The use of the approximation here is not problematic, because, within any run of #DPLL with a fixed a, the $\hat{Q}^{(a)}(n)$ values for smaller n are learned first before values for bigger n. Since $n > n_1$ and $n > n_2$, $\hat{Q}^{(a)}(n_1)$ and $\hat{Q}^{(a)}(n_2)$ will be fairly accurate when they are invoked. In other words, in learning $\hat{Q}^{(a)}(n)$, all interactions between the branching rules are ignored and each is evaluated separately. In the second phase, the value functions $\hat{Q}^{(a)}(n)$ are used as a first approximation to $\hat{Q}(n, a)$. Through the use of the minimum operator in determining the new samples, all interactions between branching rules are now taken into account.

4.5 Policy Construction

The policy for selecting branching rules can be constructed dynamically through the use of the value function. Depending on how the value function is learned, different policies may result. In particular, we consider three cases:

IND In state n, select the branching rule a that minimizes $\hat{Q}^{(a)}(n)$. This is the policy that results from the individual value functions for each branching rule.

ALL In state n, select the branching rule a that minimizes $\hat{Q}(n, a)$. In this case, $\hat{Q}(n, a)$ has been learned using $\hat{Q}^{(a)}(n)$ as a first approximation (two phases).

SCR Like ALL, in state n, select the branching rule a that minimizes $\hat{Q}(n, a)$. However, in this case, $\hat{Q}(n, a)$ has been learned from scratch.

RND In state n, select one of the branching rules uniformly at random. This is a purely randomized policy and is used as measure of comparison.

5. Experimental Results

The SAT instances that were used for experimentation fall into the following classes:

- *Graph Coloring*: These are SAT encodings of graph coloring problems available from the online Satisfiability Library (SATLIB)[4]. We used the `flat050-115` set that encodes problems with 50 nodes, 115 edges, and 3 colors. The SAT encodings contain 150 variables and 545 clauses.

- *Random 3-CNF*: These are random satisfiable 3-CNF instances, generated using the `mkcnf` generator by Allen Van Gelder [5].

- *Network Reliability*: These are SAT encodings of feed-forward networks. The #SAT solution of such an instance indicates the number of possible routes from the source to the sink and therefore provides a measure of the reliability of the network.

- *RDUP*: These are "random deep unit propagation" instances with 60 variables and 157 clauses that consist of two disjoint subinstances. The first subinstance (40 variables) allows for deep unit propagations (chained clauses) and it is fixed:

  $$(\bar{x_1} + \bar{x_2} + x_3)(\bar{x_2} + \bar{x_3} + x_4)...(\bar{x_{38}} + \bar{x_{39}} + x_{40})$$
  $$(x_2 + \bar{x}_3)(x_4 + \bar{x}_5)...(x_{38} + \bar{x}_{39}).$$

  The second (20 variables) is a random 3-CNF generated by the `mkcnf` generator. We designed this distribution so it would exhibit different behavior from the others.

Results are presented from the Graph Coloring and the RDUP classes only. The results from the other classes are similar to those from the Graph Coloring class.

[4] URL: `http://www.satlib.org`

[5] Available from `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/contributed/UCSC/instances/`.

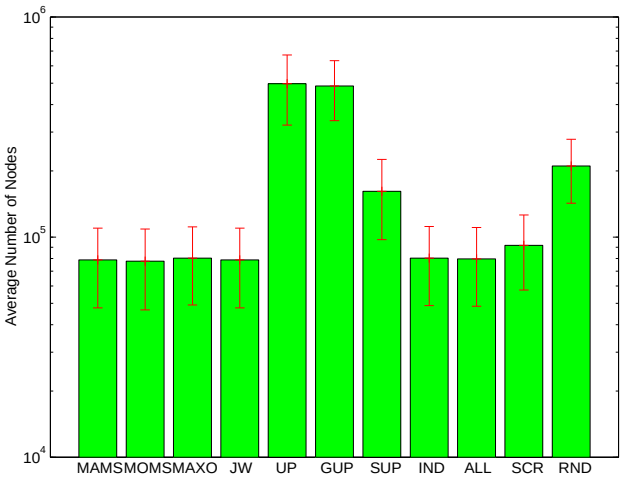

Figure 1. Performance (nodes) on the Graph Coloring class.

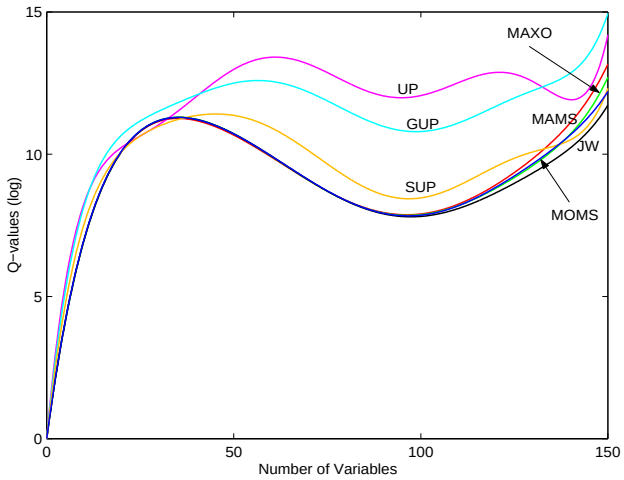

Figure 2. Learned value function for the Graph Coloring class.

5.1 Methodology

For each class of problems, a set of 100 instances was used for training. For the two-phase learning, the individual value functions were learned first by going through the training set once for each branching rule using solely that branching rule. Then, in the second phase, 10 more passes through the set were executed with a $1-\epsilon$ randomized policy ($\epsilon = 1.0$ for the first 7 and $\epsilon = 0.4$ for the rest), for a total of 17 passes. When learning from scratch, a total of 17 passes were performed (14 with $\epsilon = 1.0$ and 3 with $\epsilon = 0.4$). Three sets of learned parameters were stored corresponding to the policies IND, ALL, and SCR described previously. A polynomial of degree 7 was used in the approximation.

Performance of the learned policies was tested on a separate test set of 100 instances from the same class. All instances in the test set were solved and the average number of nodes, as well as the average running time, were recorded. The results were compared against the performance of each individual branching rules on the test set and against the purely random policy RND. All code was written in *C* and all experiments were performed on an Alpha 164LX machine. Running times correlated closely with node counts (at approximately 100k nodes/sec), and are not reported here.

5.2 Performance Results

Figure 1 shows the performance of all branching rules and the learned policies on instances taken from the Graph Coloring class (error bars indicate the 95% reliability intervals). The learned policies (IND, ALL, SCR) achieve a performance level equal to that of the best branching rules, but not better. Nevertheless, they do better than chance, as the RND bar indicates. This behavior can be easily explained by looking at the learned value function (Figure 2). Using the number of variables as the state of the process, the branching rules MOMS, JW, MAMS and MAXO are almost indistinguishable and clearly dominate the others (UP, GUP, SUP). In this case, our selection scheme cannot take advantage of switching to different branching rules, and the resulting performance matches that of the dominating branching rule.

Performance results on the RDUP class are shown in Figure 3. In this case, the learned policies IND and ALL are statistically better than any of the individual branching rules. Learning from scratch (SCR) did about as well as the best individual branching rule; it seems that the two-phase learning yields better approximations in this case. What is more interesting is that moving from IND to ALL did not improve performance, although the corresponding value functions were quite different.

Looking at the value function for ALL (Figure 4), it is easy to see that there is no dominating branching rule and therefore switching between branching rules at run time can be used to advantage. One can see that in the region 40–60, MAXO is presumably used to solve the 20-variable subinstance, whereas in the region 0–40, the UP and GUP branching rules are used for the 40-variable subinstance, for which they are the most appropriate. In this case, using size as the single state feature was sufficient to find a cut between these two regimes and to make a large performance difference.

6. Limitations and Future Work

The main weakness of this work is clearly the insufficient state representation. The number of variables provides only

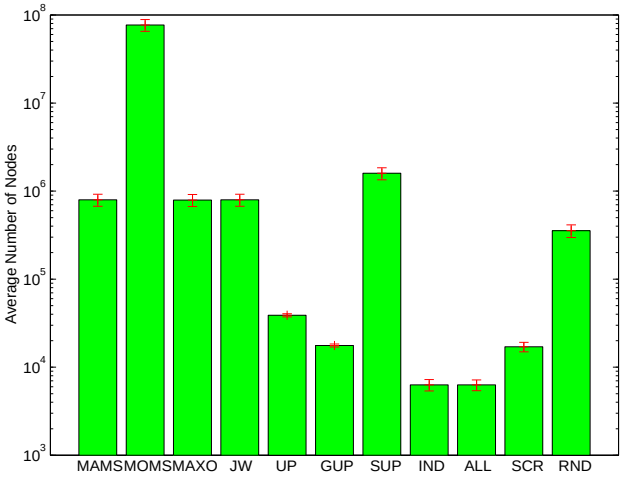

Figure 3. Performance (nodes) on the RDUP class.

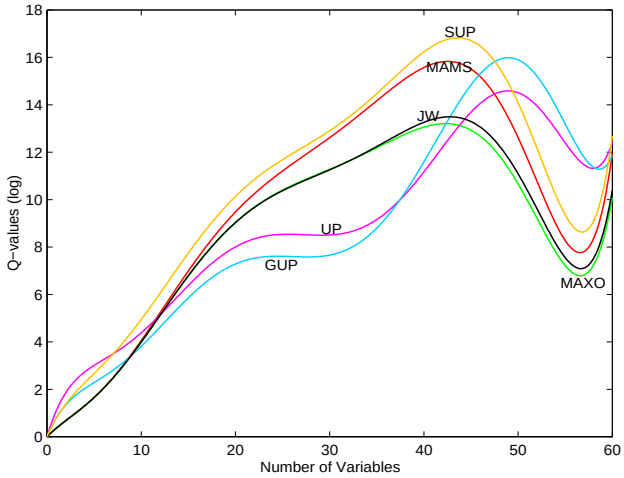

Figure 4. Learned value function for the RDUP class.

a coarse partition of the space of possible SAT instances and throws away much of the structure in the problem. A good state description would partition the space of instances in a way that different branching rules are best in different regions of the partition. However, such good state features for SAT are yet to be discovered. Features such as the induced graph width might be good candidates, but they come at a significant computational cost that might outweigh their usefulness.

Nevertheless, this work demonstrates that some degree of reasoning, learning, and decision making on top of traditional search algorithms can improve performance beyond that possible with a fixed set of hand-built branching rules.

Acknowledgments

Research supported in part by NSF grant IRI-9702576. The first author was also partially supported by the Lilian-Voudouri Foundation in Greece. The authors gratefully acknowledge the influence of Don Loveland, Ron Parr, and Henry Kautz in helping to shape this work.

References

Birnbaum, E., & Lozinskii, E. L. (1999). The good old Davis-Putnam procedure helps counting models. *Journal of Artificial Intelligence Research*, *10*, 457–477.

Boyan, J. A. (1999). Least-squares temporal difference learning. *Machine Learning: Proceedings of the Sixteenth International Conference* (pp. 49–56). Morgan Kaufmann, San Francisco, CA.

Bradtke, S. J., & Barto, A. G. (1996). Linear least-squares algorithms for temporal difference learning. *Machine Learning*, *22*, 33–57.

Davis, M., Logemann, G., & Loveland, D. (1962). A machine program for theorem proving. *Communications of the ACM*, *5*, 394–397.

Hooker, J. N., & Vinay, V. (1995). Branching rules for satisfiability. *Journal of Automated Reasoning*, *15*, 359–383.

Lagoudakis, M. G., & Littman, M. L. (2000). Algorithm selection using reinforcement learning. *Proceedings of the Seventeenth International Conference on Machine Learning* (pp. 511–518). Morgan Kaufmann, San Francisco, CA.

Li, C. M., & Anbulagan (1997). Heuristics based on unit propagation for satisfiability problems. *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence* (pp. 366—371). Nagoya, Aichi, Japan.

Majercik, S. M., & Littman, M. L. (1998). MAXPLAN: A new approach to probabilistic planning. *Proceedings of the Fourth International Conference on Artificial Intelligence Planning* (pp. 86–93). AAAI Press.

Rice, J. R. (1976). The algorithm selection problem. *Advances in Computers*, *15*, 65–118.

Roth, D. (1996). On the hardness of approximate reasoning. *Artificial Intelligence*, *82*, 273–302.

Valiant, L. G. (1979). The complexity of enumeration and reliability problems. *SIAM Journal of Computing*, *8*, 410–421.