Learning Search Control in Adversary Games

Yngvi Björnsson and Tony Marsland
University of Alberta, Department of Computing Science,
Edmonton AB, Canada T6G 2H1
E-mail:{yngvi,tony}@cs.ualberta.ca

Abstract

Recently there has been increased interest in applying machine learning methods to adversary games. However, the emphases has been mainly on learning evaluation function parameters and opening book lines, with little attention given to other aspects of the game. In contrast, learning as applied in the domain of planning and scheduling has focussed on ways of speeding up the search process, primarily by deriving rules for controlling the search. Unfortunately, most board games are far too complex to make such a rule-based approach feasible. In this paper we introduce a new framework for learning search control, and give experimental results in the domain of chess.

1 Introduction

The number of nodes visited by the $\alpha\beta$-algorithm grows exponentially with search depth. This obviously limits how far the search can "see", especially because game-playing programs must meet external time-constraints — often having only a few minutes to make a move decision. The question now becomes: how can the program best use the available time to find a good move? Although, the basic formulation of the $\alpha\beta$-algorithm explores all continuations the same number of plies, it has long been evident that this is not the best search strategy. Instead, interesting continuations are explored more deeply while less interesting alternatives are terminated prematurely. In chess, for example, it is common to resolve forced situations, such as checks and recaptures, by searching them more deeply. The search efficiency — and consequently the move-decision quality — of the $\alpha\beta$-algorithm is greatly influenced by the choices of which lines are investigated deeply and which not. Therefore, the design of a search-extension scheme is fundamental to any game-playing program using an $\alpha\beta$-like algorithm. Several studies have been conducted to quantify the relative importance of various extension schemes [7, 2, 1]. Unfortunately, the more elaborate the search-extension scheme, the more difficult it is to parameterize to achieve its full search-efficiency potential.

In this paper we introduce a method for automatically learning search-extension parameters. The method is equally well suited to learn either during on-line play, or by analyzing game-positions off-line. We start by presenting an unified framework of search-extension schemes, and follow with a description of the learning system itself. We show how the learning can be formulated as a function approximation task, thereby allowing us to employ existing learning techniques. This requires a special cost-model that is described later. Finally, learning results are presented for a strong chess program.

2 Search Control

Although all based on the same principles, the specific search-control schemes employed by the various game-playing programs differ somewhat from one program to the next. Therefore, to make our learning system as widely applicable as possible, we introduce a unified framework that attempts to encapsulate the various implementations.

2.1 A Unified View

Figure 1 shows an example game-tree that is being searched to an arbitrary depth, say d'. For any node x in a tree let P_x stand for the move-path leading from the root of the tree to that node. For example, the path P_H consists of the move sequence connecting nodes *A-B-E-F-G-H*. In our unified framework a function $D(P, \vec{w})$ decides how deep to expand each line of play, that is, the current move-path is expanded until:

$$D(P_x, \vec{w}) \geq d'.$$

The function takes the current move-path as its first argument and returns its *depth*. Note that the depth of the path is not necessarily the same as its length. The *length* is simply the number of moves on the path, whereas the *depth* can be determined by whatever criteria we like. When a path's depth is less than its length, the path will be extended beyond the nominal search horizon (search-reductions occur when the opposite is true). The special case where the depth of each move-path equals its length results in a search strategy that explores all continuations by the same fixed number of plies. The second argument, $\vec{w}$, is a vector of search-control parameters that influence the depth calculations. These parameters are made explicit because they are the ones we want to learn. In practice, there are probably additional parameters that must be passed to the function (e.g. the root game position and the α and β search bounds). However, to simplify our notation we do not show them, but we may assume arbitrary many such parameters (as long as they are not a function of $\vec{w}$).

Our framework is quite general and incorporates most of the different search extension schemes known to us. On the other hand, when implementing a com-

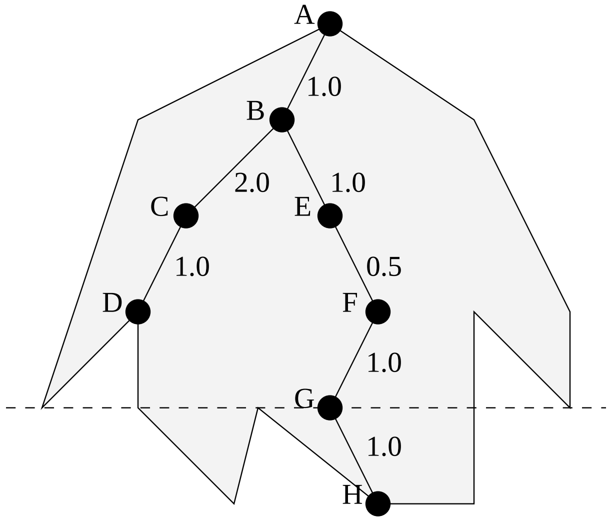

Figure 1: Search-control schemes - unified view

petitive game-playing program, one typically does not have an explicit function for calculating the depth of the move-path at each frontier node — instead the depth is updated incrementally. However, this does not pose a problem as long as there exists a conceptually equivalent formulation in form of a depth function.

2.2 Fractional-Ply Extensions

Here we show how a commonly used search-extension scheme, often referred to as *fractional-ply extensions* [5, 4], can be trivially formulated within the unified framework (the game-playing program we use for our experiments employs this type of extension scheme). The existence of predefined move-classes is assumed, where each class has a weight associated with it. Examples of move-classes in chess could be: checking moves, recaptures, and advanced passed-pawn pushes. During the search each move is categorized as belonging to one of the move-classes, and the depth of the current move-path is the sum of the class-weights of the moves on the path. Referring to Figure 1, the numbers on the paths show the class-weight of each move. For example, the depth of path P_D is $1.0 + 2.0 + 1.0 = 4.0$, and similarly the depth for P_H is 4.5. More formally, if we assume that there are N predefined classes, the depth function becomes:

$$D(P, \vec{w}) = \sum_{i=1}^{length(P)} w_j \mid j \equiv Class(m_i)$$

where m_i is the i-th move on the path, and the vector $\vec{w}$ contains the weights for each of the N move-classes (the element w_j is the weight of class number j).

The $Class(m_i)$ function categorizes each move as belonging to one of the move-classes $1, ..., N$. The search-control parameters to be tuned are the weights of the move-classes.

3 The Learning System

The main advantage of the general framework above is that search-control learning can now be viewed as a function-approximation task, namely approximating the $D(P, \vec{w})$ function. In other words, the task of the learning system is to find the most appropriate weight vector $\vec{w}$. As for all such tasks, we need to specify the training experience, the exact representation of the target function (i.e. the function we are trying to approximate), and the algorithm for adjusting the weights.

3.1 Training Experience

We want the game-playing program to learn from its mistakes and adapt the search behavior accordingly. However, for that to be possible the program must first recognize when it makes a mistake. For human players this is generally not a difficult task. Experienced players will in retrospect realize where they went wrong; the player might have over-estimated his or her chances in a particular position, chosen a dubious plan, or simply overlooked some tactical continuations. On the other hand, this is a challenging task for a computer-player. Obviously, if the program loses a game in an abrupt fashion it is clear that a mistake was made, but to pin-point exactly what move or moves were the cause of the defeat is not trivial. This problem is sometimes referred to as the *credit assignment* problem, and is hard in the general case. However, there are situations where mistakes can be identified with a high degree of certainty.

Figure 2 shows a search tree for a game in progress: the moves connected by the solid lines have already been played, and currently the program is searching game position C. Based on the search the program determines the principal continuation to be m_1,...,m_n (shown as dotted lines), and assesses the position as having a value v_C. Now, assume that when it was the program's turn to move at position A the assessment was significantly higher, or

$$v_C < v_A - \tau$$

where τ is a positive constant representing the significance margin. The program evaluates its chances much worse now than just a move ago: clearly something must have gone wrong! But what caused this undesirable change of fortune? One of two things could be responsible. It might be that position A was already bad but that the program just didn't realize it. Alternatively, it could be that position A was fine and the move m_A was a mistake — and only now does the program see the bad consequences of that move. However, in either case, position A was assessed incorrectly. Thus, A is referred to as a *critical position*, and the move sequence m_A, m_B, m_1, ...,m_n as the *solution path* of the position.

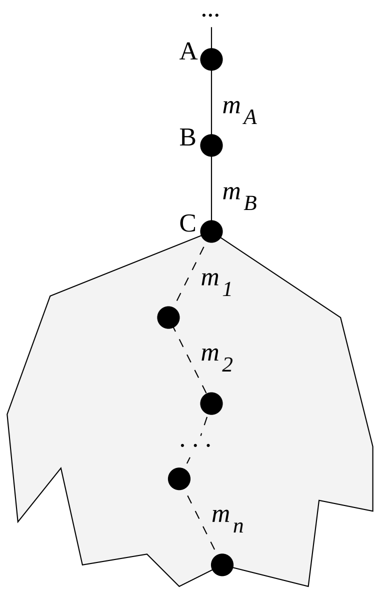

Figure 2: Identifying mistakes.

The basic assumption that we make here is that: *if the search is to correctly assess position A, its solution path (S_A) must be fully explored.* [1] This implies that the game-tree for position A needs to be explored as far as the depth of its solution path. Critical positions and their solution paths form the training input for our learning system. Many existing problem test-suites consist of a collection of game positions and their corresponding solution paths, meaning that they can also serve as a training input for our learning method.

It is interesting to notice that we cannot learn from cases where the positional estimate increases from position A to C. The reason is that the in-between move made by the opponent (m_B) might simply be a blunder. The search might have explored that move at position A deeply enough to correctly discard it as a bad move, in which case there is no need to adapt the search.

3.2 Target Function

The function we want to approximate is the depth function $D(P, \vec{w})$. However, it is not clear how the training experience from the above example helps us do exactly that. Although we know position A and its solution path, there is no information about the "correct" depth for the path. This renders supervised learning methods practically useless. Instead, we must go about this indirectly.

[1] Note, this is not a sufficient condition for correctly assessing the position, because other lines in the game-tree might also need to be explored more deeply. We are only assuming this to be a necessary condition.

One way of reformulating the problem is to ask: which weight vector results in the search expanding the fewest nodes possible to find the given solution path? Given our previous assumption, we know that to find the solution the position must be expanded to at least the depth of its solution path. Therefore, we alter the question slightly to: *when expanding the position to the depth of its solution path, which weight vector causes the search to expand the fewest nodes possible?* The only problem is that without actually performing the search we have no way of telling how many nodes it will take to explore position A that deeply (and during a game we cannot revisit the position to search it again).

Suppose that we have a cost-model, $C(p, \vec{w}, d)$, that predicts how many nodes it takes to search position p to depth d using weight vector $\vec{w}$. We could use this cost-model to answer the question we posed above, that is, we could predict the number of nodes it takes to expand position A to the depth of its solution path as:

$$C(A, \vec{w}, D(S_A, \vec{w})).$$

More generally: given a set of training samples, T, where each sample is a pair $\langle p_t, S_t \rangle$ consisting of a game position (p_t) and a solution path (S_t), we are interested in finding the weight vector $\vec{w}$ that minimizes the total number of nodes (as estimated by the cost-model) it takes to "solve" all the samples. In other words we want to minimize the function:

$$F(\vec{w}) = \sum_{t \in T} C(p_t, \vec{w}, D(S_t, \vec{w})).$$

If the function $C(p, \vec{w}, d)$ were known this could be done numerically, or even analytically. However, for games of any complexity it is practically impossible to analytically model this function. Not only does it depend on the weight vector but also on various positional features. A key observation here is that it is *not necessary to formally model the function over the entire search-space* to be able to minimize it. When using a hill-climbing like method, it is sufficient to be able to approximate it for any individual point in the search space.

3.3 Learning Algorithm

A standard hill-climbing method, *gradient-descent* [6], is used to minimize $F(\vec{w})$. Although the method guarantees finding a global minimum only for concave functions, nonetheless, in practice it is a highly effective heuristic approach to optimization and forms the bases of various learning systems (e.g. the back-propagation rule in artificial neural networks). The gradient-descent method starts with some initial setting for the weight vector $\vec{w}$, and then repeatedly iterates over all the training samples, each time updating the weight vector in the *opposite* direction of the gradient. The gradient of $F(\vec{w})$ specifies the direction of weight changes that produce the steepest increase in the value of $F(\vec{w})$. Therefore, by adjusting the parameters in the opposite direction, one expects the value of the function to incrementally decrease at each iteration. This process is continued until some termination condition is met. The condition could be as simple as doing a fixed number of iterations, or a more elaborate one like: continue until negligible progress is being made. The gradient provides the sign and relative size of each weight change, while the step size — that is, how much the weights are altered — is controlled by the learning rate parameter μ. The learning rate is typically decreased after each iteration to avoid stepping over the minimum and to ensure eventual convergence. The exact procedure for decreasing μ depends on the search domain, and is often determined by trial and error.

Our adapted implementation is outlined as Algorithm 1 below. The algorithm starts by initializing the search control parameters (w_i) to 1 (lines 1-3), and then repeatedly iterates over the test-suite data T (consisting of game positions p_t and corresponding solution paths S_t). In our experiments a fixed number of iterations is done. Before starting each iteration we initialize the variables that record the total node count information (lines 6-8). The variable *nodes* indicates the total number of nodes that our cost-model predicts it will take to solve all the problems in the test-suite, whereas the $\Delta nodes_i$ variables store how much this node count would change if we were to alter each of the search control parameters. The node count information accumulate as we go through the test-suite sample by sample (lines 9-12). After finishing looking at all the game positions the search control parameters are updated proportionally to how much a change in them will affect the total node count (lines 13-14). The Δw_{max} constant is used for controlling the step size. Basically, a parameter change that causes 100% increase in the node count would result in a weight change of exactly Δw_{max} (given a learning rate of 1.0). Larger or smaller node count changes are adjusted proportionally. Finally, before starting the next iteration, the learning rate (μ parameter) is decreased.

Algorithm 1 *Gradient – Descent*

```
1:  // Initialize $\vec{w}$
2:  for $i = 1, N$ do
3:    $w_i \leftarrow 1$
4:  // Iterate until a sufficiently good $\vec{w}$ is found.
5:  while not terminate do
6:    $nodes \leftarrow 0$
7:    for $i = 1, N$ do
8:      $\Delta nodes \leftarrow 0$
9:    for all $p_t \in T$ do
10:     $nodes = nodes + C(p_t, \vec{w}, D(S_t, \vec{w}))$
11:     for $i = 1, N$ do
12:       $\Delta nodes_i \leftarrow \Delta nodes_i + \partial C(p_t, \vec{w}, D(S_t, \vec{w}))/\partial w_i$
13:   for $i = 1, N$ do
14:     $w_i \leftarrow w_i - \mu \ \ \Delta w_{max} \ \ (\Delta nodes_i / nodes)$
15:   $\mu \leftarrow Decrease(\mu)$
```

A detail one might have noticed is that there is no direct reference to the actual game-playing program in the learning algorithm, only to the cost-model. How can that be? The truth is that our cost-model uses the game-playing program to provide information about how many nodes the search actually expands.

Although, we show the algorithm here as operating on an existing test-suite of training samples, it is also suitable for learning from on-line game-play. Then, instead of updating the weight vector after each iteration, it is updated after each training sample (or a subset of samples). This is a more convenient approach when learning during on-line game-play, where we want to update the weights either immediately after encountering a critical position (see section 3.1) or, alternatively, in between games. This approach is sometimes referred to as *incremental gradient-descent* [6]. When using an incremental version of the algorithm it is important to use a slower learning rate (a smaller μ) to make sure the weights are not changed drastically based only on a single learning sample. An alternative approach would be to log to a file all critical positions and solution paths encountered during on-line play, and then learn off-line from the resulting test-suite using Algorithm 1.

4 Modeling the Search

So far we have assumed the existence of a cost-model, but at the same time implied that it is impossible to accurately model the search. This seems paradoxical. However, as we mentioned before, it is not necessary to formally model the search over the entire search-space. When traversing the hypothesis space of possible weight vectors, the gradient-descent algorithm requires information about only a relatively few individual points in the search-space. Fortunately, we have a way of approximating these points by using actual searches!

4.1 Cost-Model

The cost-model assumed by the learning algorithm returns an estimate of how many nodes the search expands when position p is explored to depth d.[2] Because the node count typically grows exponentially with increased search depth, the cost function must be of the basic form:

$$C(p, \vec{w}, d) = B(p, \vec{w})^d. \tag{1}$$

The $B(p, \vec{w})$ function measures the growth rate of the search, for example, $B(p, \vec{w}) = 4$ means that it takes 4 times as many nodes to search position p to depth $d+1$ than to depth d. For uniform game trees $B(p, \vec{w})$ would be equal to the branching factor of the tree. However, in practice game trees are far from

[2]Alternatively, one could measure the running time of the algorithm. However, that measure is a little more problematic because of hardware dependence, and non-deterministic behavior when running experiments on a multi-user platform. In any case, the number of nodes explored per second by the search algorithm is fairly constant within each phase of the game, and so these two measures are approximately equivalent.

being uniform. Nonetheless, we can talk about an average branching factor for such trees. The model above can therefore also be used for irregular trees as long as the the growth rate is relatively constant with respect to the search depth.

It is important to understand how altering the search-control parameters $\vec{w}$ affects the node count estimate. Recall that for any given position p and corresponding solution path S, we are interested in knowing $C(p, \vec{w}, D(S, \vec{w}))$. Modifying any weight has two fundamental effects:

- the exponential growth rate $B(p, \vec{w})$ changes, and
- the required search depth $D(S, \vec{w})$ is affected.

Typically, these two are contra-effective, for example, a change that reduces the depth of the solution path also tends to inflate the growth rate of the search. Intuitively, one would expect that altering the weights such that the required search depth is reduced would result in the least node count. However, it is quite possible that the modified weights will affect the exponential growth of the search in such a way that the estimated node count for the reduced search depth will indeed be higher than the one before. The right balance must be found.

For the learning we are particularly interested in knowing the partial derivatives of $C(p, \vec{w}, D(S, \vec{w}))$. Applying the chain rule for a function of two variables gives:

$$\frac{\partial C(p, \vec{w}, D(S, \vec{w}))}{\partial w_i} = C(p, \vec{w}, D(S, \vec{w})) \times \left(\frac{D(S, \vec{w})}{B(p, \vec{w})} \frac{\partial B(p, \vec{w})}{\partial w_i} + \ln(B(p, \vec{w})) \frac{\partial D(S, \vec{w})}{\partial w_i} \right) \quad (2)$$

The depth function $D(S, \vec{w})$ is external to our model and is assumed to be known, so are its derivatives. The only unknown quantities in the above equation are therefore the $B(p, \vec{w})$ function and its partial derivatives.

4.2 Approximating $B(p, \vec{w})$ and its Partial Derivatives

In our cost-model the growth-rate function $B(p, \vec{w})$ is independent of the search depth. Therefore, by knowing the node count for only a single search depth we can determine the growth rate. For instance, in the example given in Figure 2 we know how deeply position A was actually explored, say to depth d_A, and how many nodes were expanded, say n_A. Presumably, d_A is less than the depth of the solution path of position A. Now, by substituting d_A and n_A in for d and $C(p, \vec{w}, d)$ in equation 1, respectively, we get:

$$n_A = B(p, \vec{w})^{d_A} \Rightarrow B(p, \vec{w}) = n_A^{\frac{1}{d_A}}. \quad (3)$$

The resulting approximation of the growth rate will allow us to use the cost-model, and we can now estimate how many nodes the search will explore if expanding position A to the depth of its solution path. Because, in practice, the growth rate is not truly constant, this in only an estimate. Nonetheless, given that the depth d_A is reasonably close to the depth of the solution path, the estimate will be sufficiently accurate.

The partial derivatives are a little more problematic. Recalling that the partial derivatives simply state how much the value of the function is expected to change if each weight is increased by a small amount, they can be approximated as:

$$\frac{\partial B(p,\vec{w})}{\partial w_i} = \frac{(B(p,\vec{w}+\vec{\Delta}_i) - B(p,\vec{w}))}{\delta_i} \tag{4}$$

where $\vec{\Delta}_i$ is a vector whose the $i-th$ element equal to δ_i and all the other elements zero. This requires us, though, to know the value of each of the $B(p,\vec{w}_i+\vec{\Delta}_i)$. One approach to come up with these values is to perform N (number of weight parameters) additional searches using a differently altered weight vector each time, and then use equation (3) to estimate the growth rate of the search for each of the altered weight vectors. Unfortunately, this is not feasible because of our requirement that the learning system be used during on-line play. Instead, *during the normal search we simultaneously estimate for each of the N altered weight vectors ($\vec{w}_i+\vec{\Delta}_i$) how many nodes the search would expand if using that vector instead.* In addition to the normal depth, separate depths and node counts are recorded for each modified weight vector. The node count information gathered this way allows us to estimate each of the $B(p,\vec{w}_i+\vec{\Delta}_i)$ in the same way as we did for $B(p,\vec{w}_i)$, by using equation (3).

This is illustrated in Figure 3 for the fractional-ply extension scheme (see Figure 1). Assume that the tree shown is expanded using weight vector $\vec{w} = \{1.0, 2.0, 0.5\}$, that is, the first, second, and the last move-class have weights 1.0, 2.0, and 0.5, respectively. The dark shaded area shows the subtree we would expect the search to expand if using an altered weight vector $\{1.0+\delta_1, 2.0, 0.5\}$ ($\delta_1 > 0$). At position G, for example, if the depth $D(p_g, \vec{w}+\vec{\Delta}_1)$ exceeds the search-depth limit, node H would not be expanded. Therefore it is not included in the total node count for that weight vector. Similarly the node count information for the other two modified weight vectors are simultaneously gathered (not shown in the figure).

Note that this approach only approximates how many nodes are searched; if we really were to use a modified weight vector different values would be propagated up the tree, likely causing another set of branches to be expanded in some of the sub-trees. Nonetheless, this approximation gives us a good measure of the sensitivity of the search to changes in the search-control parameters. For this approximation to work, each of the weights must be altered such that the move-paths become shorter — otherwise, the actual search would terminate before the altered depths reach the search-depth limit. In the previous example this means adding a positive constant to each of the weights. However, because we do not put any restrictions on the form of the depth function, this might in other extension schemes imply that a weight has to be reduced. One can even envision schemes where changing a parameter in either direction causes some

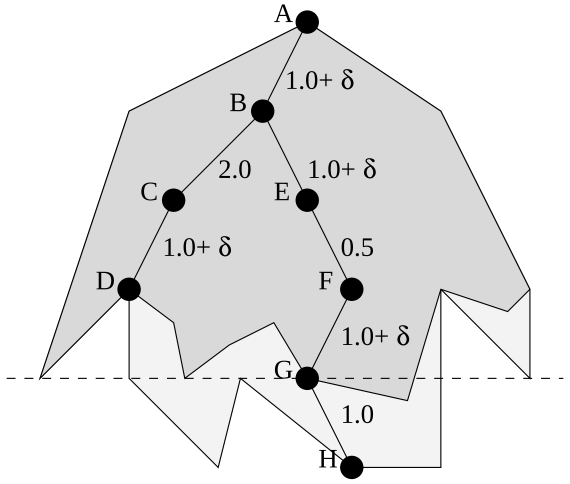

Figure 3: Approximating $B(p, \vec{w_1} + \vec{\Delta}_1)$

move-paths to shorten but others to lengthen. In such cases it might be possible to replace the troublesome parameter with two new ones, such that a parameter adjustment now causes consistent changes in the move-path depths. When that is impossible, it might be necessary as a last resort to explore some paths in the tree beyond the depth the actual weight vector does, although, this would impose undesirable overhead on the search.

5 Experimental Results

To obtain practical experience with the learning method we implemented it in the chess program Crafty [4].[3] The program uses a fractional-ply based extension scheme with five different move categories: checks, re-captures, forced-replies to checks (i.e. only one possible reply), advanced passed pawn-pushes, and mate-threats. The task is to learn the extension weight for each of the move-classes. The last category does not fit directly into the framework we introduced earlier, so we did not include the weight for that class as one of the parameters to learn. We ran two independent sets of experiments. In the first, the program was trained using a suite of chess-problems, while in the second the program learned during actual game play.

[3]Crafty is one of the strongest, if not the strongest, of the publicly available chess programs. On the on-line chess servers it consistently ranks among the highest rated players, out-performing both some of the commercial chess programs and strong chess masters. The source code is publicly available via ftp at ftp.cis.uab.edu/pub/hyatt. Our learning scheme was implemented in version 14.13 (a bug in the extension scheme was also removed; this bug was officially fixed in version 16.4 of Crafty).

5.1 Test-suite

Within the computer-chess community it is common practice to benchmark performance of chess programs against standard test-suites. In the first set of experiments we observed the performance improvement of the program as it learned using the well-known WAC test-suite [3]. This suite consists of 300 tactical chess positions. Initially, the weights of the move categories were set to 1.0, and allowed to vary within the range [0.0,2.0]. For each of the problems, if the right move was not found after examining half a million nodes the search was stopped.

In Figure 4, the top graph shows how the program's performance improves from one learning iteration to the next. The dotted line shows the percentage of problems solved (out of the 300), and the solid line shows the total number of nodes searched relative to the first iteration (53.5 million nodes). It is worthwhile recalling that the learning algorithm minimizes the total number of nodes required to solve the problems, the increase in problems solved follows indirectly as a side effect! The performance improves significantly as we can

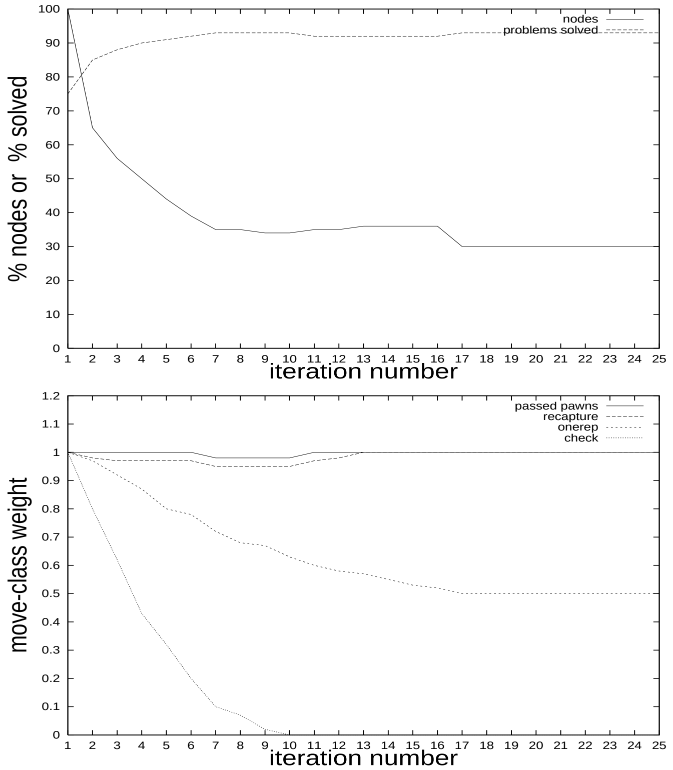

Figure 4: Learning results from test-suite data

see. After seventeen learning iterations the weights have converged to values where the total node count is reduced to only 30% (16.5M) of the original, at which level 93% (281) of the problems are solved correctly as opposed to 75% (227) in the beginning. For a comparison, using the hand-set weights (0.25 for all the classes), the program solves 94% (282) of the problems, but requires substantially more nodes to do so, or 18.5M. The bottom graph shows how the move-class parameters evolve. The check extension weight rapidly drops to zero, indicating that checking moves are a particularly important category to extend on. The forced-reply weight also decreases, but more slowly, and finally converges to a value of 0.5. The two remaining weights, re-captures and passed pawn pushes, do not change. This implies that these kind of extensions are not important for this particular test-suite. Also of interest is the sharp drop in the node count in iteration seventeen. By comparing the two graphs we see that this occurs when the forced-reply weight becomes exactly one half. This is a typical behavior when using a fractional-ply extension scheme. Sometimes a marginal change in the fractional-ply value decides if an extension is being done or not.

Many test-suites, including the one we used, provide only the best move for each position instead of the complete solution sequence. Because our learning method requires that the full solution-path be known, we had to make some compromises. If the best move returned by the program agrees with the move suggested by the test-suite, we assume that the principal-variation given by the program represents the correct solution path. However, if the move returned does not agree with the test-suite we simply ignore the problem for learning purposes. As a consequence, in each iteration we are minimizing the total number of nodes needed to solve only a subset of the problems in the test-suite, that is, those problems we have been able to derive a solution path for. However, this sub-set gradually increases with each iteration and eventually contains over 90% of the problems in the test-suite. A different approach that we could have taken is to find solution paths for all the positions in the test-suite, by pre-searching them to a much greater depth. The drawback of that approach is that a few of the more difficult problems require extremely deep searches to be solved. These few problems would dominate the total node count needed to solve all the problems. This is undesirable, and the compromise approach we take avoids this problem altogether.

5.2 Game playing

In the second set of experiments the program learned from playing games. A version of the program using the learning scheme played 60 games against an unmodified version of the chess program (with a 5 minute time limit for each side for completion of entire game). As before, the move-class weights of the learner are initialized to 1.0. The program learns from critical positions encountered during the game (see Section 3.1). The evaluation drop threshold for a position to be considered critical is set to half a pawn. Once the game position of the learner is considered to be lost (the position evaluation is more than the value of 1 pawn down) the learning is disabled for the rest of the game. The reason is that once the position is already significantly worse, it is almost inevitable that one will lose more material and eventually the game. To learn from such losing examples is not particularly instructive.

In Figure 5 we see how the parameters evolve during the training match. One thing we notice is that the parameters fluctuate more than in the previous

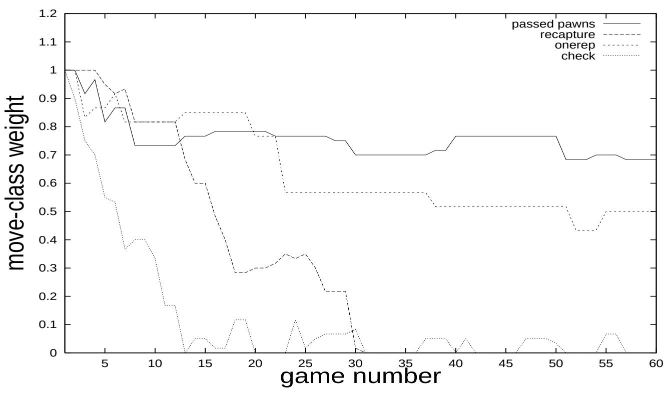

Figure 5: Learning results from game data

experience. When learning from a test-suite, each weight update is based on a collective data from looking at all the game positions in the suite. When learning from games the weights are updated after each game, where each game provides at most only one or two training samples. Given the low number of samples each weight update is based on, it is quite normal to see such fluctuations. On the other hand, there is a clear trend in the weight movements. The checking and re-capture weights quickly converge to zero, whereas the two remaining weights converge to less aggressive extension values.

It is interesting to compare the weights learned from game-play to the hand-set ones, and the ones we learned from the test-suite data. Table 1 lists the different weight settings. As we can see, none of the weight settings agree. Of particular interest is that the weights learned from game-play differ substantially from the ones learned from the test-suite. The check and forced-reply parameters agree, whereas the passed-pawn and, in particular, the re-capture weights have a totally different emphasis. In game-play re-capture extensions are considered important and a full extension is done (i.e. the weight is 0.0), whereas for the test-suite re-captures are not extended (have weight 1.0). This result supports criticism that test-suites mainly consisting of tactical game positions (and most do) are not that representative of actual play. To tune the parameters to achieve optimal search efficiency on such data does not necessarily result in the best overall play. The WAC test-suite is clearly an example of a test-suite over-representing tactical features.

We still have not addressed the question how good the weights are that we

Table 1: Learned weights

Move-class	Hand-set	Learned	
		Game-play	Test-suite
Checks	0.25	0.00	0.00
Re-captures	0.25	0.00	1.00
Forced-reply	0.25	0.50	0.50
Passed-pawn	0.25	0.68	1.00

learned from game-play. Unfortunately, we have no way of telling what the optimal weights are and, thus, we cannot really say how close to optimal the learned weights are. On the other hand, we can compare them with the hand-set weights. To evaluate the quality of the learned weights, we matched a version of the program using the learned weights against another version using the hand-set weights. The only difference between the versions was the value of the search-control parameters. The match consisted of 200 games played at time controls of five minutes per game.[4] To prevent the programs from repeating move sequences in the opening, each game was started from a different, well-established opening position. The programs played each starting position once as White and once as Black. Table 2 lists the result of the match. The program using the weights learned from game-play won the match with a comfortable margin. Furthermore, we can state with 95% confidence that the program using the learned weights does perform better than the program using the hand-set weights.[5]

Table 2: Match results

$Crafty_{Learn}$ vs. $Crafty$		
Time control	Score	Winning %
5 minutes	108.5 - 91.5	54.25

The extension scheme employed by our test program is a relatively simple one, using only a few parameters. These parameters have been hand-tuned to reasonable values, and thus opportunity for drastic improvement is small. On the other hand, we expect the benefits of the automatic tuning to become even more relevant for more sophisticated extension schemes, which require the tuning of many additional parameters.

[4]The match was played on a single Intel PIII/450 computer. In the chess-program all the default parameter settings were used, except that pondering (thinking on opponent's time) was turned off. Otherwise, the programs would compete for CPU time.

[5]The *student's t-test* was used to compare the mean of the score distribution of the two programs.

6 Conclusions and Future Work

In this paper we introduced a new method for learning search-control parameters in adversary search. By using a cost-model to model the search, the learning task can be formulated as a function approximation task, allowing us to use well-established machine learning techniques for determining the most appropriate parameter vector. The learning method was tested in a strong chess program, where it learned a parameter vector that outperformed the hand-set parameter vector (one chosen by a leading computer-chess expert). Another lesson to learn is that the common practice to measure programs' performance against tactical test-suites is suspect, and can lead to parameter settings that are not optimally suited for actual game-play.

The automation of the tuning of search-control parameters opens up many new opportunities for improved search-control schemes. Traditionally, the effort it takes to hand-tune complex extension schemes — possibly using many disparate parameters — imposes a limitation on how complex the schemes can be in practice. However, by automating the tuning process it is possible to experiment with more sophisticated schemes. This is a logical next step. One can envision schemes that use many more move-classes, where each class is not only dependent on the phase of the game but also on other positional features. For example, different extensions would apply for open vs. closed positions, or positions where the king is exposed. However, in this paper, we were mainly concerned with describing the new learning method and evaluating its soundness and applicability.

Finally, the learning method itself is not specific to adversary search. The only assumption we make here is that there is a parameterizable depth function that controls how deeply each branch of the tree is expanded (and that we can reasonably well model the search by our cost model). It is definitely worth experimenting with it in other tree search domains; single-agent search is one that comes to mind.

References

[1] T. Anantharaman, M. S. Campbell, and F. Hsu. Singular extensions: Adding selectivity to brute-force searching. *Artificial Intelligence*, 43(1):99–109, 1990.

[2] D. Beal and M. C. Smith. Quantification of search extension benefits. *ICCA Journal*, 18(4):205–218, 1995.

[3] F. Feinfeld. *Win At Chess*. McKay, New York, 1945. Also (1958), Dover, New York.

[4] R. Hyatt. Crafty - chess program. 1996. ftp.cis.uab.edu/pub/hyatt.

[5] D. Levy, D. Broughton, and M. Taylor. The sex algorithm in computer chess. *ICCA Journal*, 12(1):10–21, 1989.

[6] T. M. Mitchell. *Machine Learning*, pages 92–94. WCB McGraw-Hill, 1997.

[7] C. Ye and T. A. Marsland. Experiments in forward pruning with limited extensions. *ICCA Journal*, 15(2):55–66, 1992.