Learning Extension Parameters in Game-Tree Search

Yngvi Björnsson and T. A. Marsland

Department of Computing Science
University of Alberta
Edmonton, Alberta
CANADA T6G 2E8

Abstract

The strength of a program for playing an adversary game like chess or checkers is greatly influenced by how selectively it explores the various branches of the game tree. Typically, some branch paths are discontinued early while others are explored more deeply. Finding the best set of parameters to control these extensions is a difficult, time-consuming, and tedious task. In this paper we describe a method for automatically tuning search-extension parameters in adversary search. Based on the new method, two learning variants are introduced: one for offline learning and the other for online learning. The two approaches are compared and experimental results provided in the domain of chess.

1 Introduction

In the planning and scheduling domain, learning methods are applied successfully to improve search efficiency [18]. These methods work primarily by deriving and refining control rules. Unfortunately, such a rule-based approach is not feasible for learning how to control the search in two-person games such as chess, checkers and Othello. First of all, experience gained over the decades shows the difficulty of producing rules that generalize well from one game position to the next. Secondly, efficiency is of paramount importance and the overhead of manipulating complex search-control rules can easily outweigh the possible benefits.

Even though various researchers have acknowledged that automatic learning of search control in two-person games is an important avenue for research, machine learning in games has not focused so much on search control but on other aspects of the game. For example, many different schemes exist for

learning evaluation function parameters [4,7,2] and, more recently, work is being done on dynamic adjustment of opening books [8,13]. On the other hand, attempts so far at automatically learning search control have not proved particularly successful. For example, *explanation-based learning* [20] and *case-based reasoning* [15] approaches, although interesting, have yet to demonstrate improved search efficiency in competitive play. Likewise, an attempt to use an evolving neural network to focus the search of an Othello program has been at best only moderately successful [21]. The authors acknowledge that the method is not applicable to more complex games like chess in a straightforward way. In games like Go, where search is of a lesser importance, some success has been achieved recently by learning search-control rules [9].

Fürnkranz provides a recent survey of learning in games. In that overview the disappointing results in learning search-control parameters led to the following conclusions:

> ... the tuning of various parameters that control search extension techniques would be a worth-while goal ... It may turn out that this is harder than evaluation function tuning because it is hard to separate these parameters from the search algorithm that is controlled by them. [10]

In this paper we introduce a novel method for learning search-extension parameters. In the next section we give a brief overview of search control in adversary games, focusing on search extensions. After that we formalize the problem of learning search extensions and then describe the learning method. Two variants of the learning method are given. The first learns by analyzing labeled training examples *offline*, whereas the latter uses an *online* method that learns during game play. Finally, we demonstrate the usefulness of the new learning method by using it to improve the playing strength of a high-performance world-class chess-playing program.

2 Search Control in Two-Person Games

The $\alpha\beta$-algorithm [14] is the search strategy of choice for such board games as chess, checkers and Othello. The search efficiency of the algorithm can be improved in a couple of ways: either by better move ordering or by selecting dynamically how deeply to explore each line. This raises the question: *how should one use the available time to find a good move*?

2.1 Improving Search Efficiency

The better the move ordering, the fewer nodes the $\alpha\beta$-algorithm expands. Recently there have been attempts to improve existing move-ordering schemes for several games using specifically trained neural networks [11,16]. However, existing hand-crafted move-ordering techniques are already very successful, not leaving much scope for further improvement.

On the other hand, there is still much performance improvement to be gained from selective expansion of game trees. Whereas the basic $\alpha\beta$ formulation explores every continuation 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 promising alternatives are terminated prematurely. In chess, for example, it is common to resolve forced situations, such as checks and re-captures, by searching them more deeply. Several studies have been conducted to quantify the relative importance of the various extension schemes [1,24,3], showing that the move-decision quality is greatly influenced by the depth-selection strategy. Therefore, the design of a search-extension scheme is fundamental to any game-playing program using an $\alpha\beta$-like algorithm. Unfortunately, the more elaborate the search-extension scheme, the more difficult it is to parameterize to achieve its full efficiency. In this paper we focus on learning the search-extension parameters.

2.2 Generalized Search-Extension Scheme

Although based on similar underlying principles, search-extension schemes differ from one game-playing program to the next. Thus, to make our learning method as widely applicable as possible, we introduce a general search-extension framework that attempts to encapsulate the various schemes used in practice. Our learning method can be used to parameterize different extension schemes as long as they fall within the generalized framework.

Here we are only concerned with depth-first expansion of game trees. Furthermore, we assume that the search-extension strategy used for controlling the expansion may be expressed in a form of a function that can be applied at every node in the tree. [1]

[1] When implementing a high-performance 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 the form of a depth function.

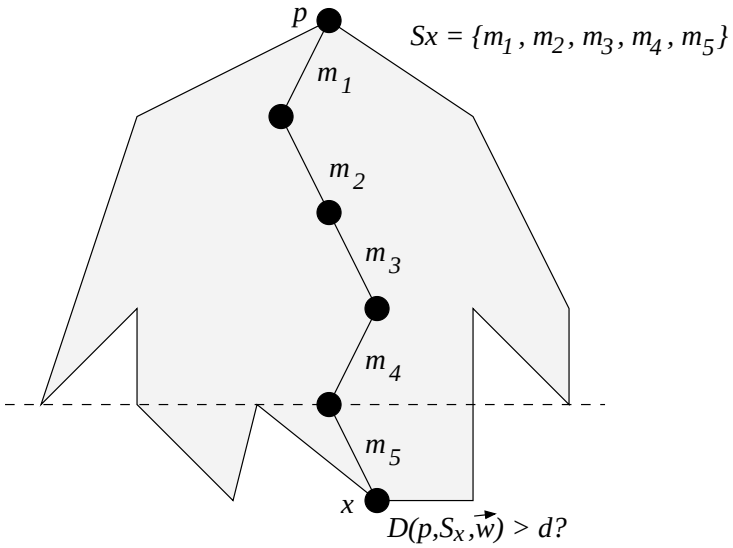

Fig. 1. Generalized search-extension/reduction model.

The generalized search-extension/reduction scheme is shown in Figure 1. For any node x in a tree, let S_x stand for the move path leading from the current game position at the root of the tree, p, to node x. The depth-calculation function takes the current game position and a move path as arguments and returns the *depth* of the path. That is, the current move path is explored until:

$$D(p, S_x, \vec{w}) \geq d.$$

where d is the depth of the current search iteration. The third argument, $\vec{w}$, is a vector of search-control parameters that influence the depth calculations. These are the parameters we are interested in learning.

Note that in our notation a *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 only restriction we put on the depth function is that it be *monotonically non-decreasing*, that is, the depth of a move path will never decrease by adding moves to the end of the path.

3 Formulating the Learning Problem

The main advantage of the generalized framework above is that learning of search extensions can now be viewed as a function-approximation task, namely approximating the $D(p, S, \vec{w})$ function. In other words, the task of the learning system is to find the most appropriate weight vector $\vec{w}$. Unfortunately,

even though we were given training data in the form of game positions and their solution paths, we have no information about the "correct" depth for the path, and so cannot use supervised learning methods. In the absence of labeled training data, reinforcement learning methods have in recent years often been used with success. The basic idea is that the learning algorithm will (eventually) receive feedback from its environment indicating the goodness of the actions taken. For example, in games the final outcome is often a strong indicator of the merit of previous game states. Temporal-difference learning, a reinforcement based algorithm, has been used successfully to learn evaluation functions for games like backgammon, chess, and checkers [23,2,22]. The basic underlying property that makes reinforcement learning successful in this setting is that the game history can be viewed as an episodic task, where evaluation of later game states tends to be more accurate than the earlier ones. In our case, unlike evaluation function learning, it is quite unclear how learning of search extensions can be formalized as an episodic task over a set of game states, thus it is not clear how to apply reinforcement-learning techniques. Instead, we must go about the search-extension learning indirectly.

3.1 Minimization Problem

One way of reformulating the problem is to ask, for a given position, which weight vector results in the search expanding the fewest nodes, while finding the given solution path? More generally, given a set of training samples, T, we want to find the parameter vector $\vec{w}$ that minimizes the total number of nodes it takes to "solve" all the samples. Specifically, we want to minimize the cost function:

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

where the $C(p_t, S_t, \vec{w})$ function is a cost model providing the number of nodes visited by the game-playing program before discovering the solution S_t, when it searches position p_t using parameter vector $\vec{w}$.

Given that such a cost model exists we can use standard techniques to minimize this function. For now, let us assume that this is the case; in later sections we see how to derive such a model (the offline and the online learning variants use different cost models).

3.2 Learning Algorithm

Well-known hill-climbing methods, such as *gradient-descent*, can be used to minimize $F(\vec{w})$. Although the gradient-descent method only guarantees finding a global minimum for concave functions, nonetheless, in practice it is

a highly effective heuristic approach to optimization and forms the basis of various learning systems (e.g. the back-propagation rule in artificial neural networks). The method starts with some initial setting for the weight vector $\vec{w}$ and then repeatedly iterates over all the training samples, updating the weight vector after each iteration. The gradient of $F(\vec{w})$ specifies the direction of weight changes that produces the steepest increase in the value of $F(\vec{w})$. Therefore, by adjusting the weights in the opposite direction, one expects the value of the function to decrease. This process continues until some termination condition is met: such as doing a fixed number of iterations, or because negligible progress is being made.

The gradient-descent method as adapted to our task is outlined as Algorithm 1. First the search-control parameters (w_i) are initialized to 1.0 (lines 2-4). Alternatively, random values could be assigned. Next the algorithm repeatedly iterates over the test suite data T. Before starting each iteration it initializes the variables that record the total node-count information (lines 7-10). The variable *nodes* stores the total number of nodes that our cost model predicts it will take to solve all the problems in the test suite, whereas each $\Delta nodes_i$ stores how much this node count would change if we were to alter the corresponding search control parameters, w_i. The node-count information accumulates as we go through the test suite sample by sample (lines 11-16). The gradient (line 14) is used to tell how much the node count will change if a weight were to be altered. After 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 17-19). 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 μ =1.0). Larger or smaller node count changes are adjusted proportionally. Finally, before starting the next iteration, the learning rate may be decreased.

3.3 Cost Model

The problem with the aforementioned approach is that we do not know the cost model $C(p_t, S_t, \vec{w})$ and hence cannot compute its gradient! Furthermore, it is impossible to analytically model such a function. Not only does it depend on the weight vector, but also on various game-dependent 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. Fortunately, we have a way of doing that by performing actual searches. Our offline and online learning systems use somewhat different methods for approximating the cost model and its gradient.

Algorithm 1 The Learning Algorithm

```
1: // Parameter description.
2: w⃗: A vector of the parameters to be learned (w_i).
3: N: The number of parameters.
4: T: The training data, a set positions and their solutions.
5: μ: The learning rate.
6: // Initialize the value of the parameter vector w⃗.
7: for i = 1, N do
8:    w_i ← 1.0
9: end for
10: // Iterate until a sufficiently good w⃗ is found.
11: while not terminate do
12:    // Reset node count to zero.
13:    nodes ← 0
14:    for i = 1, N do
15:       Δnodes_i ← 0
16:    end for
17:    // Now loop over all the positions in the training set, accumulating the
18:    // total number of nodes that need to be searched (according to the
19:    // cost model) when solving these positions. Also, using the partial
20:    // derivatives of the cost model, keep track of how many more/less
21:    // nodes would be searched if each of the parameters were to be changed.
22:    for all (p_t, S_t) ∈ T do
23:       nodes = nodes + C(p_t, S_t, w⃗)
24:       for i = 1, N do
25:          Δnodes_i ← Δnodes_i + ∂C(p_t, S_t, w⃗)/∂w_i
26:       end for
27:    end for
28:    // Adjust the parameters values based on the cumulative node counts.
29:    for i = 1, N do
30:       w_i ← w_i − μ Δw_max (Δnodes_i/nodes)
31:    end for
32:    // Gradually decrease the learning rate in between iterations.
33:    μ ← Decrease(μ)
34: end while
```

4 Offline learning

Figure 2 shows the basic architecture of the offline learning system. One of the main design objectives behind this architecture is to isolate the learning module from the game-playing program, thus minimizing the changes needed to the game-playing program itself. The learning system consists of three main parts: the learning module, the game-playing engine (a separate process) and, finally, a pre-generated database of training examples (where each example

consists of a game position and information about the corresponding best move).

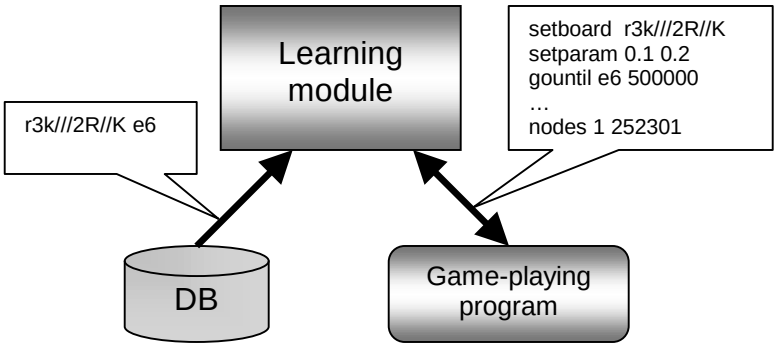

Fig. 2. The architecture of the offline learning system.

The learning module, which is also the main driver, reads in the game positions from the database and then repeatedly calls the game-playing engine, asking it to solve each of the positions using different search-control parameters. In the following subsections we describe each of the components in more detail.

4.1 Learning Module

The offline learning system uses the gradient-descent algorithm introduced earlier. Whenever the $C(p_t, S_t, \vec{w})$ function is called, the game-playing program is used to provide the necessary information. The interaction with the game-playing program is abstracted away in the cost-model function. From the learning module point of view, the function is simply a cost model predicting the number of nodes the game-playing program expands, when solving game position p_t using the supplied parameter vector.

As for the gradient, recall that it is used to tell by how much the value of the cost function changes if a parameter value is adjusted slightly. Instead we use a "pseudo-gradient" by substituting line 14 in Algorithm 1 with the following call to the cost model:

$$\Delta nodes_i \leftarrow C(p_t, S_t, \vec{w}_i) - nodes_i$$

where the vector $\vec{w}_i$ is the same as $\vec{w}$ except that the $i-th$ parameter has

been adjusted slightly by a small positive constant δ.

Note also that most existing test suites provide only the best move in each position as a solution (as opposed to the whole solution path). This does not pose a problem here, we can simply think of the best move as a complete solution path; that is, a problem is considered to be solved when the game-playing program proposes a move that matches the one in the database.

4.2 Game-Playing Program

The only changes required to the game-playing program is augmentation of its interface to support the following three commands:

- **setboard** *position*
  Set the current game state to be *position*. The learning module is indifferent to the representation of a game state or position (it simply relays this information from the database), but the game-playing program needs to understand the format. This command also resets the state of the game engine such that a new search can be performed independently of previous searches (e.g. the transposition table and other history information must be cleared). No return value is expected.
- **setparam** w_1 w_2 ... w_n
  Specify the values of the search-control parameters. The arguments $w_1, ..., w_n$ are real numbers and represent the values that the search-control parameters take. The game-playing program can scale these parameters or map them to integers (if the program's internal representation requires so). No return value is expected.
- **gountil** *move maxnodes*
  This command instructs the game-playing program to search the current game position until the program agrees that *move* is the best continuation in the given position, or an imposed search limit of *maxnodes* is reached (it is important to have such an upper limit on the number of nodes searched, otherwise a single extremely difficult test positions can dominate the total node count for the entire test suite). The *gountil* command returns the number of nodes searched and also a flag indicating whether the suggested move was found by the search. The return string has the following format:

  nodes *flag count*

  where *flag* is set to 1 if the problem was solved, otherwise 0. The *count* tells how many nodes were expanded by the search (for an unsolved position *count* is the node-count limit *maxnodes*).

The cost-model function $C(p_t, S_t, \vec{w})$ sends the three commands described above (*setboard*, *setparam*, and *gountil*) to the game-playing program and then waits until it receives the expected return string ("nodes ..."). Many game-playing programs already have commands built-in with similar capabilities, e.g. a command to set up a game position, a command for specifying the value of a (search) parameter, and a command to perform a search. Thus, implementing the above three commands is typically as simple as mapping them onto already supported interface functions.

5 Online Learning

Implementing an online learning system is a much more challenging task. First of all, the game-playing program needs to understand where is goes wrong and adapt its search behavior accordingly. Secondly, it is essential that an online method be computationally efficient, so as not to inflict excessive overhead on the game-playing program. For example, using additional searches to estimate the gradient, as we did in the offline learning scheme, would be impractical in an online learning system. In here we give an overview of how we overcame these hurdles. A more thorough discussion appears elsewhere [6,5].

5.1 Training Samples

When learning during online play we do not have test data pre-labeled with information about what the right move decision is. Instead the game-playing program must learn from its mistakes. For that to be possible the program must recognize when it makes a mistake. For human players this is generally not that difficult a task, whereas for a computer player this is challenging. [2] However, there are situations where mistakes can be identified with a high degree of certainty.

Figure 3 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

[2] Similar problems are encountered during opening-book learning [13]. One difference though is that in our setting it is important to pin-point an exact move as being a mistake, whereas opening-book learners often get away with not doing that. Instead, without necessarily understanding where a mistake was made, they can simply discourage playing particular opening lines that often result in a loss.

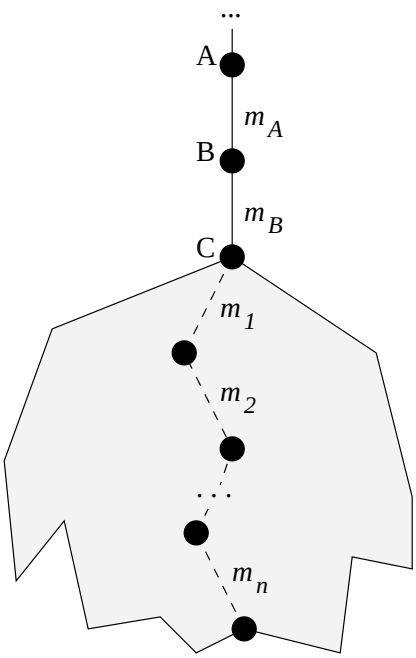

Fig. 3. Identifying mistakes.

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 now evaluates its chances as being poorer than just one move before; 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 is known as the *solution path* of the position. 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.*[3] This implies that the game tree for position A needs to be explored to 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 note that it is not instructive to learn from cases where

[3] 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.

the positional estimate increases from position A to C. The reason is that the in-between move made by the opponent, that is move 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 change the search parameters.

5.2 Learning Module

Again the underlying learning algorithm is gradient-descent. Whereas Algorithm 1 shown earlier operates on a test suite of training samples, it can also be adapted to learn from online 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 online game play, since we want to update the weights either immediately after encountering a critical position (see above) or, alternatively, in-between games. This approach is sometimes referred to as *incremental gradient-descent* [19]. With the 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.

In the offline learning we could replace the cost model with a call to the game-playing program. However, when encountering a critical position during a game, we cannot afford additional search to find how many nodes it would take to explore the position to the depth of its solution path. Instead we use the following estimator of the cost:

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

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. Even though the game trees themselves are highly irregular, the model above can be used as long as the growth rate is almost constant with respect to the search depth. It is important to understand in this model how altering the search-control parameters $\vec{w}$ affects the node-count estimate. Modifying any weight has two fundamental effects: the exponential growth rate $B(p, \vec{w})$ changes, and the solution path depth $D(p, S, \vec{w})$ is affected. Typically, these two are counter-acting, 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 smallest 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. This is what the gradient of the

cost model, as given by Equation (2), provides:

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

The derivation of the gradient is shown in Appendix A.1. The only unknown quantities in Equations (1) and (2) are the $B(p,\vec{w})$ function and its partial derivatives. Appendix A.2 shows how to estimate that function and compute its derivatives. In our cost model the growth-rate function is constant with respect to the search depth. Therefore, for any given search, by knowing the node count for only a single search depth we can determine the growth rate. In turn, once we have determined the growth rate for that particular search we can use our cost model to predict how many nodes the search expands when exploring any given solution path.

The partial derivatives are more problematic. When performing a search we simultaneously estimate for each of the altered weight vectors $\vec{w_1}$, ... , $\vec{w_N}$ how many nodes would be expanded if they were used instead (recall that weight vector $\vec{w_i}$ is identical to $\vec{w}$ except that the i-th weight has been slightly adjusted). In addition to the normal depth, separate depths and node counts are recorded for each of the modified weight vectors $\vec{w_i}$. The node-count information gathered this way allows us to estimate each of the $B(p,\vec{w_i})$ in the same way as for the unmodified weight vector. An example of this procedure is shown in Appendix A.2.

6 Experimental Results

To obtain practical experience with our new method we used it to learn search-control parameters for the chess program CRAFTY [12]. [4] This program uses a so-called fractional-ply extension scheme. Instead of every move counting a full ply toward the search depth, some move types are worth only a fraction of a ply. For example, if a move class is worth half a ply, two such moves can be expanded on the same path during a one ply search. The smaller the fraction,

[4] CRAFTY is one of the strongest, if not the strongest, of the chess programs whose source code is publicly available. On the online chess servers it consistently ranks among the highest rated players, outperforming 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 originally implemented in version 16.4 and later re-implemented in version 16.17. The results reported here are based on that latter version.

the more aggressive are the extensions. The depth function can be expressed as:

$$D(p, S, \vec{w}) = \sum_{j=1}^{length(S)} w_i \mid i \equiv Class(m_j)$$

where m_j is the j-th move on the path, the vector $\vec{w}$ contains the weights for each of the N move classes (the element w_i is the weight of class number i). These weights are the search-control parameters we want to tune. The $Class(m_i)$ function categorizes each move as belonging to one of the move classes $1, ..., N$. The chess program uses six different move categories: checks, re-captures, forced-replies to checks (i.e. only one legal reply), advanced passed pawn-pushes, null-move threats, and other moves. The weight of the last move class is fixed to one, and serves as a baseline. Unfortunately, the null-move threats move-class does not fit directly into the framework we introduced earlier. The reason is that a move can only be classified into this category by actually performing a null-move search. Thus, to keep things simple, we chose to disable it in our experiments. This leaves four search-control parameters to be tuned.

We ran two independent sets of experiments. In the first, we compare the offline and online methods as they learn parameter values from data in a test suite of chess problems. In the second experiment a program augmented with the online scheme learns the weights during the playing of actual games.

6.1 Test Suite

We observed the performance improvement of the program as it learned using the extensive ECM test suite [17]. This suite consists of 879 (mostly tactical) middlegame chess positions. Initially the weights of the move categories were set to 1.0, and allowed to vary within the range [0.1,1.9] (1.0 ± 0.9). If the right move is not found after examining half a million nodes the search is stopped for that problem. The learning rate μ was determined by trial-and-run. For the online learning μ is initially 0.75 and decreases gradually, whereas for the offline learning μ is fixed to 1.0. The step size δ is 0.15 in both cases.

The learning algorithms try to minimize the number of nodes required to solve the problems, whereas the increase in the number of problems solved follows indirectly, because of the improved efficiency. The result of the experiment is shown in Figure 4. The two lower graphs show the offline learning algorithm, whereas the two upper graphs show the corresponding results using the online learner.

In the beginning the program solves only 39% of the problems, expanding in total 309 million nodes. The left-hand graphs show how the search efficiency

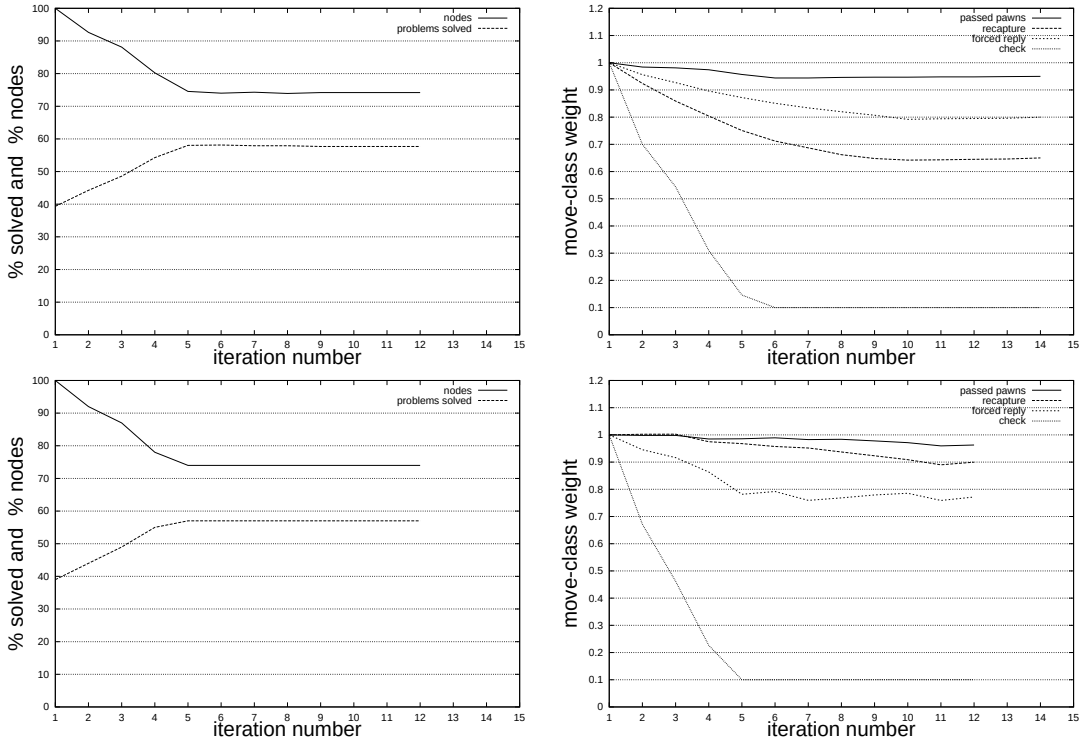

Fig. 4. Comparison of online (upper) vs. offline (lower) learner using ECM suite.

improves with each learning iteration (relative to the first iteration), both in terms of number of nodes searched and the number of problems solved (represented by a solid and a dotted line, respectively). Both the offline and online learners converge after only a few iterations, in the end solving the same fraction of the problems (57%) while searching approximately the same number of nodes (229M). In contrast, the default hand-set weights (see Table 1) solve only 56% of the problems while searching 233 million nodes.

The right-hand graphs, on the other hand, demonstrate how the move-class weights evolve. In the online case, the four move-class parameters *check*, *forced-replies*, *re-captures* and *passed-pawn pushes* converge to fractional-ply values of 0.10, 0.64, 0.79 and 0.95, respectively. For the offline learner, the values are 0.10, 0.77, 0.90 and 0.96, respectively. The weight vectors learned by the two methods differ slightly. However, the difference doesn't affect the performance: both weight vectors perform equally well as we noted above. Also, the relative ranking of each move category is the same in both cases, checks and forced-reply extensions seem to be the most critical to extend on, while the two remaining move classes, re-captures and passed-pawn pushes are of a lesser importance.

Beforehand we had hypothesized that given the same training data the offline learning method might outperform the online learner. The reason is that the offline learner works with accurate node-count information, whereas our on-

line learner uses estimates. What is important about this result is that both learning methods perform equally well. This is reassuring and adds credibility to the claim that the approximate information used by the online learner is sufficiently good for use in practice.

Many test suites, including the one we used, provide only the best-move for each position instead of the complete solution sequence. Because our online learning method requires that the full solution path be known, we had to make some adjustments. 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 and there is no stored solution path for that particular problem (i.e. from solving it in a previous iteration), we simply ignore the problem. 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 for which we have been able to derive a solution path. However, this subset gradually expands with each iteration and hopefully converges to a significant portion of the total test suite. In our case 57% of the problems were solved.

6.2 Game Playing

In the second set of experiments we incorporated our online learning method directly into the program, and used it to learn in real-time while playing games. A version of the program using the learning scheme played 100 games against an unmodified version of the chess program (with a 5 minute time limit for each side for completion of an 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. The threshold for a position to be considered critical is an evaluation drop of 1/3 of a pawn. Once the game position of the learner is considered to be lost (the position evaluation is more than 3 pawns 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 may not be instructive.

The chess program used in the experiments distinguishes between three different game phases: the opening, middle-game, and end-game. The program evaluates game positions differently depending on game phase, but search extensions are done the same way in all phases. However, by automating the tuning-process of the weights we can easily learn a different sets of weights for each game phase. Thus, our learning program is set up to use three different sets of weights, one for each game phase. In our experiments, we did not receive

Table 1. Learned weights.

Move class	Hand-set	Learned test suite		Learned game play	
		Offline	Online	Middle game	End game
Checks	0.00	0.10	0.10	0.30	0.10
Forced reply	0.25	0.77	0.65	0.47	0.25
Re-captures	0.25	0.90	0.80	0.10	0.10
Passed pawn	0.25	0.96	0.95	0.10	0.15

any learning samples in the opening phase. This is not surprising because not only is the opening phase rather short, but also the game-positions balance equally for both sides (thus not triggering any learning experience).

Table 1 shows the weights learned from playing games and compares them to some of the weight vectors discussed earlier. The left-most weight column shows the hand-set weights (tuned by the author of CRAFTY, a leading computer-chess expert), followed by the weights learned from the *ECM* test, both from using the offline and the online learner. Finally, the two right-most columns show the weights learned from playing games (separate weight vectors for middle-game and end-game play). The first point of interest is that the learned weights for middle-game and end-game play differ. In particular, we note that checks and forced-reply moves are extended more aggressively in the end game, as one might hypothesize. Next we note that the weights learned from game play differ substantially from the weights learned using the *ECM* test suite. In particular, re-captures and passed-pawn pushes are judged important extensions according to the game-play data but not the test-suite data. Most readily available test suites, including *ECM*, tend to focus almost entirely on tactical (mating) problems. In such positions checking moves are generally the key moves, whereas passed-pawn pushes and re-captures rarely are. By contrast, passed-pawn pushes play a critical role quite frequently in actual games, even more so than forced mating attacks. This may explain the discrepancy between the weights learned from playing games versus using a test suite.

6.3 Matches

We are also interested in knowing how well the different weight vectors perform in actual game play; in particular comparing the vector learned from playing games to both the hand tuned one and the one learned using the *ECM* test suite as training input.

To evaluate the quality of the weights in actual game play, we matched six

different versions of the program against each other. The only difference between the versions was the value of the search-control parameters. Each match consisted of 100 games played at time controls of five minutes per game. [5] 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.

The first program version, C_{games}, uses the weights learned from game play. For the opening phase the same weights are used as for the middle-game. The C_{ECM} version uses the weights learned by the online scheme using the ECM test suite as training input, whereas the C_{hand} version uses the hand tuned weights. The three remaining programs treat all the move-extension classes the same. In the C_{100} version all the weights are set to 1.00, which is the same as not using search extensions. The C_{010} version extends aggressively on all move classes (all weights set to 0.10), whereas the C_{050} version uses more conservative extensions (all weights set to 0.50). The result of the matches is shown in Table 2. The program using the parameters learned from game play performs the best overall, scoring 282.5 points out of the 500 games. The program using the parameters learned from the test suite does not do as well. This is not too surprising. Most test suites focus on the tactical abilities of programs. Although tactics are important, the test suites sometimes overemphasize their importance compared to actual game play. As expected, the program using no extensions (C_{100}) performs by far the worst. On the other hand, it is interesting to see how close the other programs' performance is, even though they are using quite different weight vectors. The C_{hand}, C_{010}, C_{050} all end up with a similar score. It is a little surprising to see how well the C_{050} version does, intuitively we would have thought it would rank lower. The fact that this version outranks the C_{010} and the C_{ECM} versions shows that in actual game play it is not necessarily good to extend too aggressively, since it results in many irrelevant lines being searched too deeply. Although this might improve the tactical ability of the program, it may hurt the positional play and hence the overall performance.

Unfortunately, we have no way of telling what the optimal weight vector is, and thus we cannot really say how close to optimal the learned weights are. However, based on the above results, we can state with over 90% confidence that the program using the weights learned from game playing performs better than the program using the hand-set weights. [6]

[5] The matches were played on Intel PII/400 and PIII/450 computers. Each match was played on a single computer. In the chess-program, all the default parameter settings were used, except that pondering (thinking on opponent's time) was turned off. This is necessary so that the programs do not compete for CPU time.

[6] *Student's t-test* was used to compare the mean of the score distributions of the two programs.

Table 2. Match results.

vs	C_{games}	C_{050}	C_{hand}	C_{010}	C_{ECM}	C_{100}	Points
C_{games}	-	54.5-45.5	51-49	54-46	59.5-40.5	63.5-36.5	282.5
C_{050}	45.5-54.5	-	53.5-46.5	49.5-50.5	53-47	66-34	267.5
C_{hand}	49-51	46.5-53.5	-	50-50	50.5-49.5	64.5-35.5	260.5
C_{010}	46-54	50.5-49.5	50-50	-	48.5-51.5	64.5-35.5	259.5
C_{ECM}	40.5-59.5	47-53	49.5-50.5	51.5-48.5	-	58-42	246.5
C_{100}	36.5-63.5	34-66	35.5-64.5	35.5-64.5	42-58	-	183.5

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 the opportunity for substantial improvement is small. On the other hand, the benefits of automatic tuning becomes increasingly relevant for more sophisticated extension schemes that require the tuning of many more parameters.

7 Pros and Cons

Which of the two methods for learning search-control parameters is more appropriate depends on the situation. In here we contrast the offline and the online approach, and show that the two methods, in a way, complement each other.

One of the main appeals of the offline approach is that it offers an easy way to tune search-control parameters in almost any search-based game-playing program. Moreover, only minimal modifications are required to the game-playing program itself. The changes are as simple as augmenting the game-playing program's interface with the three high-level commands introduced earlier, and the program can then be "plugged" into the learning module. On the other hand, this ease of use comes with a price. First, it requires annotated training data; that is, the correct move in each test position must be known. Secondly, it is computationally expensive. [7] These drawbacks make it impossible for it to be used to learn in real-time while playing games. The online learning method, on the other hand, is well suited to learn from either existing test suites or from game-play. This is definitely a big bonus. On the negative side, the method is quite intrusive and elaborate code modifications are necessary to the game-playing program.

We saw that existing tactical test suites, even though extensive, are not necessarily representative of the features that are important for actual game-play.

[7] However, if sufficient computer resources are available, the method is trivial to parallelize. All searches within one learning iteration can be executed in parallel — the only synchronizing point is at the end of the iteration.

This somewhat undermines the offline learning. One way of overcoming this drawback is to combine the generation of training samples from the online method with the ease of use of the offline method. That is, during a game critical positions are logged to a database. A human or computer analysis would then be performed to label the positions with the best move choice. The resulting test suite could be used as a training input for the offline learning system. Although this would require some manual labor, it might be well worthwhile.

Finally, it is worth mentioning that whereas the online method is designed to learn only search-extension parameters, the offline method can be used to learn any type of parameters that influence the search process, for example, move-ordering parameters.

8 Conclusions

The automation of the tuning of search-control parameters opens up many new opportunities for improved search-control schemes in game-playing programs. Traditionally, the effort it takes to hand-tune complex extension schemes imposes restrictions on how elaborate the schemes can be. However, by automating the tedious tuning process, it becomes possible to experiment with more sophisticated schemes using far more parameter values.

The work presented here is one of only a few successful approaches for learning search-control in competitive game-playing programs. However, this is only a first step and there is still much work to be done. For example, in this work we assumed that the search-extension features are given, and the task of the learner is simply to decide their relative importance by tuning various parameters. Research into methods for automatically discovering and constructing new search-control features for use in two-person games is a difficult, but possibly rewarding, avenue for future research.

References

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

[2] J. Baxter, A. Tridgell, and L. Weaver. Learning to play chess using temporal differences. *Machine Learning*, 40(3):243–263, 2000.

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

[4] D. F. Beal and M. C. Smith. Learning piece values using temporal differences. *ICCA Journal*, 20(3):147–151, 1997.

[5] Y. Björnsson. *Selective Depth-First Game-Tree Search.* PhD thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, April 2002.

[6] Y. Björnsson and T. A. Marsland. Learning search control in adversary games. In H.J. van den Herik and B. Monien, editors, *Advances in Computer Games 9*, pages 157–174. University of Maastricht/University of Paderborn, 2001.

[7] M. Buro. Experiments with multi-probcut and a new high-quality evaluation function for Othello. In H.J. van den Herik and H. Iida, editors, *Games in AI Research*, pages 77–96, Maastricht, The Netherlands, 2000. Universiteit Maastricht.

[8] M. Buro. Towards opening book learning. In H.J. van den Herik and H. Iida, editors, *Games in AI Research*, pages 47–54, Maastricht, The Netherlands, 2000. Universiteit Maastricht.

[9] T. Cazenave. Generation of patterns with external conditions for the game of Go. In H.J. van den Herik and B. Monien, editors, *Advances in Computer Games 9*, pages 275–293. University of Maastricht/University of Paderborn, 2001.

[10] J. Fürnkranz. Machine learning in games: A survey. In J. Fürnkranz and M. Kubat, editors, *Machines That Learn To Play Games*, pages 11–59, Huntington, New York, 2001. Nova Science Publishers, Inc.

[11] K. Greer. Computer chess move-ordering schemes using move influence. *Artificial Intelligence*, 120:235–250, 2000.

[12] R. M. Hyatt. Crafty - chess program. *FTP Site*, 1996. ftp.cis.uab.edu/pub/hyatt.

[13] R. M. Hyatt. Book learning a methodology to tune an opening book automatically. *ICCA Journal*, 22(1):3–12, 1999.

[14] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[15] J. Kolodner. *Case-Based Reasoning.* Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[16] V. Koscis and J. Uiterwijk. Learning move ordering in chess. In *Proceedings of the CMG Sixth Computer Olympiad Computer Games Workshop*, July 2001.

[17] N. Krogius, A. Livsic, B. Parma, and M. Taimanov. *Encyclopedia of Chess Middlegames.* Publisher, 1980.

[18] S. Minton. *Learning Search Control Knowledge: An Explanation-based Approach.* Kluwer Academic Publishers, Boston, MA, 1988.

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

[20] T. M. Mitchell, R. Keller, and S. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1):47–80, 1986.

[21] D. E. Moriarty and R. Miikkulainen. Evolving neural networks to focus minimax search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, Seattle, WA, 1994.

[22] J. Schaeffer, M. Hlynka, and V. Jussila. Temporal difference learning applied to a high-performance game-playing program. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 529–534, Seattle, Washington, August 2001.

[23] G. J. Tesauro. TD-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215–219, 1994.

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

A Appendix

A.1 Gradient of Cost Model

Below we show how we derived the partial derivatives of the cost model used by the online learner:

$$
\begin{aligned}
&\frac{\partial C(p,S,\vec{w})}{\partial w_i} \\
&= \frac{\partial (B(p,\vec{w})^{D(p,S,\vec{w})})}{\partial w_i} \\
&= \frac{\partial (e^{\ln(B(p,\vec{w})^{D(p,S,\vec{w})})})}{\partial w_i} \\
&= \frac{\partial (e^{D(p,S,\vec{w})\ln B(p,\vec{w})})}{\partial w_i} \\
&= e^{D(p,S,\vec{w})\ln B(p,\vec{w})} \frac{\partial (D(p,S,\vec{w})\ln B(p,\vec{w}))}{\partial w_i} \\
&= C(p,S,\vec{w}) \frac{\partial (D(p,S,\vec{w})\ln B(p,\vec{w}))}{\partial w_i} \\
&= C(p,S,\vec{w}) \left(D(p,S,\vec{w}) \frac{\partial (\ln B(p,\vec{w}))}{\partial w_i} + \frac{\partial (D(p,S,\vec{w}))}{\partial w_i} \ln B(p,\vec{w}) \right) \\
&= C(p,S,\vec{w}) \left(D(p,S,\vec{w}) \frac{1}{B(p,\vec{w})} \frac{\partial B(p,\vec{w})}{\partial w_i} + \frac{\partial (D(p,S,\vec{w}))}{\partial w_i} \ln B(p,\vec{w}) \right) \\
&= C(p,S,\vec{w}) \left(\frac{D(p,S,\vec{w})}{B(p,\vec{w})} \frac{\partial B(p,\vec{w})}{\partial w_i} + \frac{\partial (D(p,S,\vec{w}))}{\partial w_i} \ln B(p,\vec{w}) \right)
\end{aligned}
$$

A.2 Estimating $B(p, \vec{w})$ - Example

In Section 5 the introduced a method for learning search control during actual game play. One of the challenges using this method is to estimate efficiently in real-time the effects changing each of the search control parameters has on the growth rate of the search. Here we illustrate the technique used to do the estimation.

Figure A.1 shows a search tree expanded using a depth threshold of 2.0. In this example there are only two move-classes; the first has fractional-ply weight of 0.4 (pictured using dotted lines) and the second a weight of 1.0 (pictured using solid lines). We use the vector $\vec{w} = \{0.4, 1.0\}$ to represent these weights. The

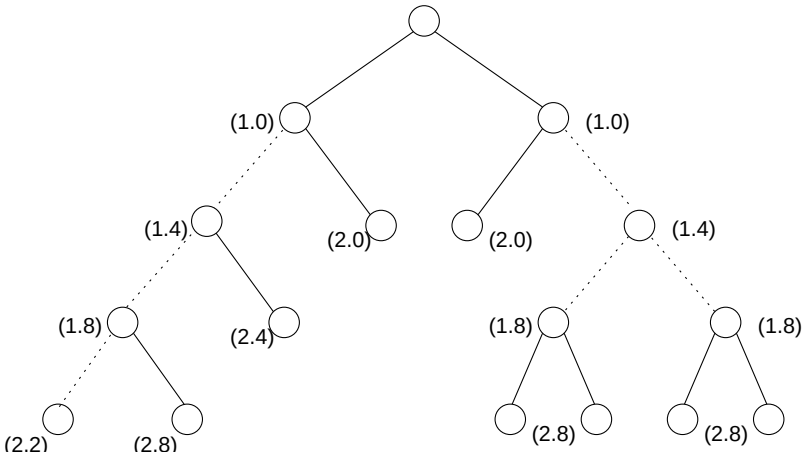

Fig. A.1. Depth of nodes in a game tree.

number besides each node shows the depth of the node. As soon as the depth equals (or exceeds) the depth threshold the node is evaluated and the search backtracks. A count of the total number of nodes expanded is also kept. In this example 17 nodes are expanded, thus the growth rate of the search is

$$B(p, \vec{w})^{2.0} = 17 \quad \Longrightarrow \quad B(p, \vec{w}) = 4.123$$

The problem we face is that we also need to simultaneously approximate the growth rate of the search, as if the parameter vectors $\vec{w}_1 = \{0.4 + \delta, 1.0\}$ and $\vec{w}_2 = \{0.4, 1.0 + \delta\}$ were instead used to expand the tree (needed for calculating the gradient). This is done by recording for each parameter vector a separate depth and count of number of nodes expanded. Figure A.2 demonstrates this process using $\delta = 0.1$.

Instead of only one depth, each node has now three depths associated with it. Each of the depths is recorded by a different weight vector, that is, $\vec{w} = \{0.4, 1.0\}$, $\vec{w}_1 = \{0.5, 1.0\}$ and $\vec{w}_2 = \{0.4, 1.1\}$, respectively. In the figure, the

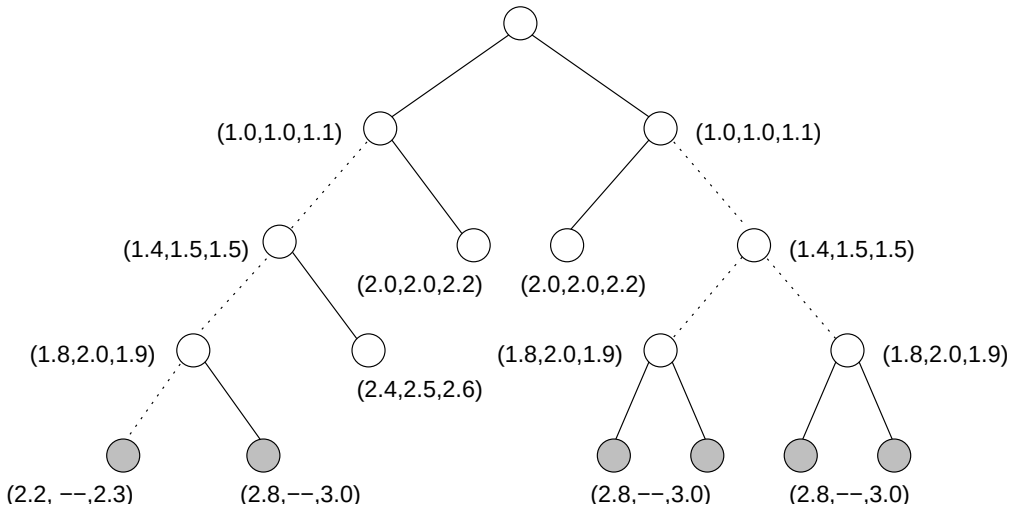

Fig. A.2. Multiple depths of nodes in a game tree.

leftmost depth of a triplet is the same as used in Figure A.1. As before it determines when to stop expanding the branches in the search tree. The two other auxiliary depths are used only for deciding whether or not to include a node in the total node count for the alternative weight vectors. That is, as soon as one of the depths reaches the threshold limit, the nodes in that subtree are not counted as being explored by the corresponding weight vector. For example, in the figure the shaded nodes are not included in the total node count for vector $\vec{w}_1$, because the $\vec{w}_1$ depth has already reached 2.0. The rationale is that if we were using that weight vector to expand the tree, this branch would not be explored this deeply. On the other hand, in this example the $\vec{w}_2$ weights expand exactly the same tree as $\vec{w}$. Thus,

$$B(p, \vec{w}_1)^{2.0} = 11 \implies B(p, \vec{w}_1) = 3.317$$

$$B(p, \vec{w})^{2.0} = 17 \implies B(p, \vec{w}_2) = 4.123$$

This approach is, of course, only an approximation of the size of the actual trees explored if the alternative weight vectors were used instead. The reason is that different evaluations would be backed up the tree, possibly causing different branches of the tree to be explored. However, this technique can be used with little overhead during the search, and works well in practice.

Finally, recall that the partial derivatives simply state how much the value of the growth-rate function is expected to change if each weight is increased by a small amount. They can be approximated as:

$$\frac{\partial B(p, \vec{w})}{\partial w_i} = \frac{(B(p, \vec{w}_i) - B(p, \vec{w}))}{\delta}$$

where δ is the small constant by which each w_i is perturbed.