Bjornsson, Y. and Marsland, T. A., *Learning Control of Search Extensions*, Procs. 6th Joint Conf. on Information Sciences, Research Triangle Park NC, (March 2002): 446-449.

Learning Control of Search Extensions

Yngvi Björnsson and Tony Marsland
Department of Computing Science
University of Alberta
Edmonton, Alberta
CANADA T6G 2E8
E-mail: {yngvi,tony}@cs.ualberta.ca

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. One of the main appeals of the method is that it is non-intrusive and domain independent. Therefore, with only minimal modifications, almost any search-based game-playing program can be "plugged" into the learning module. Experimental results are provided in the domain of chess.

1 Introduction

In the planning and scheduling domain, learning methods are applied successfully to improve search efficiency [7]. These methods work primarily by deriving and refining control rules. Unfortunately, such a rule-based approach is not feasible for learning search control 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. This calls for a different approach for learning search control.

In this paper we introduce a method for learning search-control in two-person games. We start by discussing search-control strategies in two-person games, and follow with a description of the learning system itself. Finally, learning results are presented for a strong chess program.

2 Search Control

The $\alpha\beta$-algorithm [5] is almost universally employed by programs 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. Here we are concerned only with the latter.

The number of nodes visited by the $\alpha\beta$-algorithm grows exponentially with search depth. This raises the question: how should one use the available time to find a good move? Although, the basic formulation of the 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 promising 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 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. The method we introduce here automatically parameterizes the search.

3 Architecture

The basic architecture of the learning system consists of three main parts: the learning module, the game-playing engine and, finally, a pre-generated database of game positions.

The learning module, which is also the main driver, first spawns off the game-playing program as a separate process. All communication between the learning module and the game-playing engine is done via Unix-domain sockets (pipes) that have been redirected to the game program's standard input/output. The database contains a collection of game positions, where each position is labeled with information about the correct move choice. The positions are read from the database and

used as training data. The learning algorithm repeatedly calls the game-playing engine, asking it to solve each game position in the database, but using different search-control parameters. Essentially, we are interested in finding the parameter set that solves the most positions, while searching as few nodes as possible.

One of the main design objectives is to allow minimal modification to any game-playing program "plugged" into the learning system. This implies that the learning module needs to be generic enough to learn the parameters without a detailed knowledge of the specific depth extension schemes employed by the game-playing program. We now describe the main components of the architecture in more detail.

3.1 Learning Module

Given a set of training samples, T, we are interested in finding the parameter vector $\vec{w}$ that minimizes the total number of nodes it takes to "solve" all the samples. In other words, we want to minimize the function:

$$F(\vec{w}) = \sum_{(p_t, m_t) \in T} Nodecount(p_t, m_t, \vec{w}, n),$$

where the $Nodecount(p_t, m_t, \vec{w}, n)$ function returns the number of nodes visited when the game-playing program searches position p_t using parameter vector $\vec{w}$. An upper limit of n nodes is allotted for each search. The program either finds the correct move m_t (that is, solves the problem), in which case the function returns the actual number of nodes searched, or the upper node-count limit (n) is reached, in which case n is returned. It is important to have an upper limit on the number of nodes each search can expand, otherwise a single difficult game position can dominate the entire test-suite.

A well known hill-climbing method, *gradient-descent*, 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 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.

Algorithm 1 outlines this procedure as adapted to our learning task. First the N search-control parameters are initialized to 1 (alternatively, random values could be assigned). The outermost loop iterates until the parameters converge to fixed values. At the beginning of each iteration the node-count information is reset. The total number of nodes visited when searching all the test positions in the database using the current weight vector $\vec{w}$ is called $nodes$. Similarly, the $nodes_i$ ($i = 1, 2, ..., N$) record the provisional node count used for estimating the gradient.

Algorithm 1 The Learning Algorithm

```
1: // Initialize the parameters.
2: for i = 1, N do
3:    w_i ← 1.0
4: end for
5: // Iterate until a sufficiently good w⃗ is found.
6: while not terminate do
7:    // Reset the node count to zero.
8:    nodes ← 0
9:    for i = 1, N do
10:      nodes_i ← 0
11:   end for
12:   // Search all the game positions in the test-suite.
13:   for all (p_t, m_t) ∈ T do
14:      nodes ← nodes + Nodecount(p_t, m_t, w⃗, n)
15:      for i = 1, N do
16:         w_i ← w_i + δ
17:         nodes_i ← nodes_i + Nodecount(p_t, m_t, w⃗, n)
18:         w_i ← w_i − δ
19:      end for
20:   end for
21:   // Adjust the parameter vector.
22:   for i = 1, N do
23:      w_i ← w_i − μ ((nodes_i − nodes)/nodes)
24:   end for
25:   μ ← Decrease(μ)
26: end while
```

Next the algorithm loops over all the training samples. Each sample consists of a game position (p_t) and the best move for that position (m_t). For each position the game-playing program is called $N + 1$ times: once using the current parameter vector $\vec{w}$, and then once for each of the N parameters. Each time the i-th item in the parameter vector $\vec{w}$ is increased by a small amount δ. These additional searches are used to measure how sensitive the game-playing program is to changes in each search-control parameter. The interaction with the game-playing program is abstracted away in the $Nodecount(p_t, m_t, \vec{w}, n)$ function. From the learning point of view the function simply returns the number of nodes the game-playing program expands when solving game position p_t using the supplied parameter vector.

At the end of each iteration the parameter vector $\vec{w}$ is modified in the direction opposite to the "gradient". The $Nodecount$ function cannot be expressed in a closed form, and the gradient can therefore not be derived analytically. Instead the node counts ($nodes_i$) are used to tell how each parameter change affects the $F(\vec{w})$ func-

tion, just as the gradient would. Furthermore, the learning rate μ is used to control the step size of the parameter changes. It is decreased in between iterations to guarantee eventual convergence.

3.2 Game-Playing Program

The only changes required to the game-playing program is to augment its command 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** $move\ n$
  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 n nodes is reached. The number of nodes actually searched must be returned, and also a flag indicating whether the suggested move was found by the search. The return string has the following format:

  $$\textbf{nodes}\ flag\ count$$

  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 n).

The $Nodecount(p_t, m_t, \vec{w}, n)$ function 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 commands.

4 Experimental Results

To obtain practical experience with the learning method we used it to learn search-control parameters for the chess program Crafty [4]. The program uses a fractional-ply-based extension scheme. Instead of all moves 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 1-ply deep search. The smaller the fraction, the more aggressive are the extensions.

We observed the performance improvement of the program as it learned using the extensive ECM test-suite [6]. 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 [0.1,2.0]. If the right move is not found after examining half a million nodes the search is stopped for that problem. The learning rate μ is fixed to 1.0 (no forced convergence) and the δ is set to 0.15.

In Figure 1 the two graphs show the program's improvement from one iteration to the next. On the upper graph, the dotted line shows the percentage of problems solved (out of the 879), and the solid line shows the total number of nodes searched relative to the first iteration (310 million nodes). The learning algorithm minimizes the total number of nodes searched, the increase in problems solved follows indirectly as a side effect! After only a few iterations the values have converged and the total node count is reduced to 73% (229M) of the original, at which level 57% (508) of the problems are solved correctly as opposed to only 39% (346) in the beginning.

The lower graph shows how the move-class parameters evolve. The four move-class parameters tuned were *check*, *forced-replies* (only one legal reply to check), *passed-pawn pushes*, and *re-captures*. They converge to fractional-ply values of 0.10, 0.77, 0.96, and 0.89, respectively. This shows clearly that checking moves are a particularly important category to extend on. The other parameters also decrease, although more gradually.

5 Related Work

Previous attempts to use machine-learning methods to improve search efficiency in two-person games have not been particularly successful. For example, *explanation-based learning* and *case-based reasoning* approaches, although interesting, have yet to demonstrate improved search efficiency. An overview of these approaches is given by Fürnkranz [3]. An attempt to use an evolving neural network to focus the search of an Othello program was at best only moderately successful [8]. 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 much lesser importance, moderate success has been achieved by learning search-control rules [2].

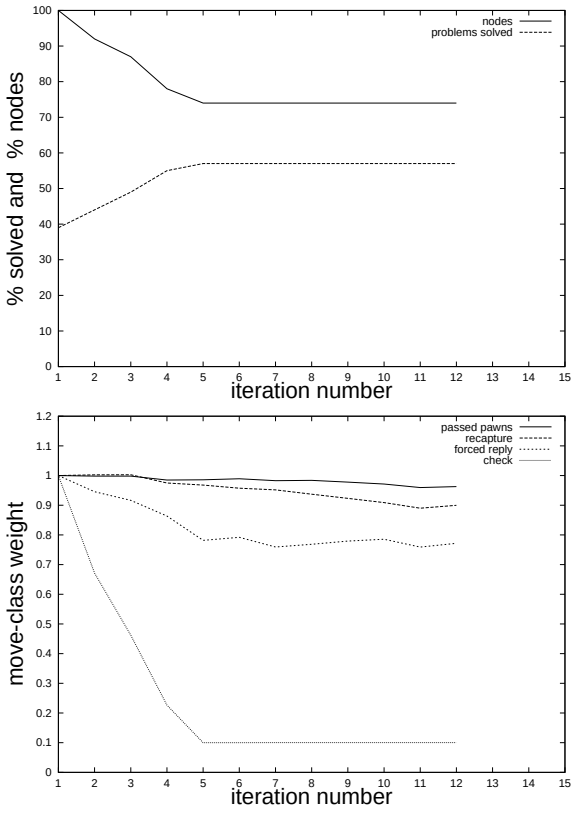

Figure 1: Learning results

Based on ideas similar to those presented here, we also developed a method for learning search extensions during on-line play (as opposed to analyzing game positions off-line) [1]. On-line learning is a more challenging task. For one thing, the absence of labeled training samples renders supervised learning approaches useless. Instead, the on-line learner needs to understand where things went wrong. Furthermore, during a game it is impossible to search the same game position multiple times using different parameter values. A different scheme is used to estimate in real-time how sensitive the search is to changes in each of the search parameters. Unfortunately, to make the on-line learning approach feasible it was necessary to impose certain constraints on the type of search extensions that can be learned, and elaborate code modifications were necessary to the game-playing engine.

On the other hand, the method introduced here, although restricted to off-line use only, does not impose such constraints and is straightforward to implement. Thus the two methods complement each other.

6 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 extensions 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 learning method introduced here tunes these search-control parameters by analyzing game positions off-line. The main appeal of the method is that almost any search-based game-playing program can be "plugged" into the learning module with only minimal modifications.

There are still many avenues for further research. For example, one of the outstanding challenges is how to automatically discover features to extend on.

References

[1] Y. Björnsson and T. Marsland. Learning search control in adversary games. In *Advances in Computer Games 9*, 2001.

[2] T. Cazenave. Generation of patterns with external conditions for the game of go. In *Advances in Computer Games 9*, 2001.

[3] J. Fürnkranz. Machine learning in computer chess: The next generation. *ICCA Journal*, 19(3):147–161, 1996.

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

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

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

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

[8] 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.