Bootstrap Learning of α-β-Evaluation Functions

Alois P. Heinz* Christoph Hense*

Abstract

We propose α-β-evaluation functions that can be used in game-playing programs as a substitute for the traditional static evaluation functions without loss of functionality. The main advantage of an α-β-evaluation function is that it can be implemented with a much lower time complexity than the traditional counterpart and so provides a significant speedup for the evaluation of any game position which eventually results in better play. We describe an implementation of the α-β-evaluation function using a modification of the classical classification and regression trees and show that a typical call to this function involves the computation of only a small subset of all features that may be used to describe a game position. We show that an iterative bootstrap process can be used to learn α-β-evaluation functions efficiently and describe some of the experience we made with this new approach applied to a game called malawi.

*Institut für Informatik, Universität Freiburg, Rheinstraße 10–12, D-79104 Freiburg, Germany. E-mail: {heinz|hense}@informatik.uni-freiburg.de

1 Introduction

Game playing programs especially those for two-person games with complete information and with alternating moves like chess and checkers make up a very important part of AI research [2, 11, 1, 4]. Programs for interesting, i.e. non-trivial games are both search-intensive and knowledge-intensive systems where the complexity of search can be traded off for complexity of knowledge and vice versa, because one can try to explore all possible paths of play from a given position towards the end of a game where the outcome is evident or one can try to find out some information about the values of next positions in a knowledge base. In practice, the search depth of the dynamic search in the game tree has to be restricted somehow and a heuristic static evaluation function is applied to the positions at the horizon of search from where they are backed up to the next positions.

Much research in developing general methods for search reduction [13] on the one hand and in finding game-specific evaluation functions on the other hand has been accomplished. The alpha-beta procedure [8, 12] is one of the best known algorithms that prunes off subtrees of the search tree through knowledge that they cannot have any further influence on the solution. There is a large record of investigations in machine learning procedures for evaluation functions, e.g. see [15, 16, 14, 10, 6].

Our new approach to evaluation function learning is three-fold: First, we define α-β-evaluation functions that allow us to extend the alpha-beta pruning scheme from the alpha-beta procedure to the static evaluation function, providing a more seamless integration of search-based dynamic and knowledge-based static evaluation. We secondly demonstrate how α-β-evaluation functions can be implemented as a modification of the classical classification and regression trees [3] which can be built from a set of n training examples in time $\mathrm{O}(n \log^2 n)$. We show that this implementation allows the α-β-evaluation function to be computed much faster than the traditional evaluation function, because only a subset of the features of a given game position needs to be computed. Finally we show that it is possible to learn an α-β-evaluation function for a game in a kind of bootstrap process from scratch, where at the beginning only the outcomes for final positions are known and at the end a good evaluation function has been generated.

The structure of the paper is as follows. The next section describes the basic background in the theory of games and in classification trees as they apply to game tree evaluation. In section 3 we define α-β-evaluation functions, show their applicability to the alpha-beta procedure and describe their efficient implementation. Section 4 contains the description of the overall bootstrap learning process and a short synopsis of our experience with this kind of learning brought into action for a definite game. Section 5 summarizes our results.

2 Preliminaries

For the sake of simplicity our following explications are restricted to the case of two-person zero-sum games with complete information and alternating moves. And further we assume that each played game has only a finite number of moves and one of two possible outcomes, win and loss. So no draws are allowed here.

Each game in this context can be described by its *game tree*. The root of this tree is the starting position, the direct descendants of an interior node are the positions that can be reached by a single move from the actual position by the player at turn, and the leaves are all positions where no more move is allowed and for which one of the outcomes win or loss for the actual player is defined. A win for one player is a loss for the other player and vice versa. We shall identify a win with the value 1 and a loss with the value -1.

In theory the *evaluation* of a game tree is a simple matter. If the value of any leaf node is given correctly by the *static evaluation function* the value of any node can be computed recursively by negating the minimum, i.e. a kind of generalized NAND function, of the set of values for the direct descendants of the node. But in practice the recursion depth of the procedure has to be restricted somehow and the static evaluation function has to be augmented to give heuristic values in the interval $[-1, 1]$ for interior nodes, where the game tree expansion has to be stopped for timing reasons.

The *alpha-beta procedure* evaluates a given game position by applying a kind of lazy evaluation scheme and so tries to minimize computation time by subtree pruning. The value it computes for a given game position is always inside the interval bounded by the provided parameters α and β, also called the *alpha-beta window*. During the computation the procedure tries to increase the return value from α towards β, and it returns when the limit β is reached or when all direct descendants of the position have been considered recursively. It returns immediately with the value of the static evaluation function if the limiting recursion depth has been reached. Initially the alpha-beta procedure is called with the window $[-1, 1]$. It should be noticed that the static evaluation function in the traditional form of the alpha-beta procedure is not called with α and β parameters and may return values outside of the alpha-beta window.

The static evaluation function has to give the real game theoretic values for leaf nodes and some heuristic values for other nodes. It is a really hard task to find a good heuristic evaluation function for interesting games that can be computed efficiently. Often the static evaluation function is computed in two separate steps. In the first step a number of features are extracted from a given game position. In the second step these features are composed by some magic function. The interesting features of game positions are often discovered and described by human experts of the game in question. In the following let us assume that a set of relevant features has been provided and is fixed.

The magic function can be given by human experts, too, or it can be learned or optimized by programs. It can have the form of a polynomial of some degree or it can be a set of tables or anything else. In most cases the computation of the magic function requires all features of a given game position to be known in advance. Thus the most inner loop of game tree evaluation includes the computation of all feature values which can be very time consumptive.

Classification trees provide one method for implementing functional evaluations for vector arguments. A classification tree is a binary tree where each inner node represents a decision of the kind "vector $[i] \leq$ constant?" and each leaf represents a classification in the form of a value. For a vector that is to be classified, first the decision at the root has to be made. If the answer is true, the algorithm repeats with the left node, otherwise with the right node of the classification tree. At last the classification value is found in a leaf of the tree. Classification trees are a nonparametric method for function representation and allow for high flexibility. Each leaf of the tree corresponds to a hyper-rectangle in D-dimensional space. It is clear that any function with a finite domain can be completely described by a classification tree and other functions may be approximated.

Classification trees can be constructed iteratively from a set of classified data vectors. Let a *learning set* L of training data vectors in D-dimensional space be given and let each vector $x \in L$ have class $\mathrm{c}(x) \in \{-1, 1\}$. To each node k of the classification tree T that is grown from L the value $\mathrm{v}(k) := |L \cap k|^{-1} \sum_{x \in L \cap k} \mathrm{c}(x)$ is assigned, that is the mean of all the class values of training vectors falling into k. The value of any vector x with respect to a classification tree T is the value of the leaf k that represents the hyper-rectangle to which x belongs, $\mathrm{v}_T(x) := \mathrm{v}(k(x))$. The *impurity* of a node k is defined to be $\mathrm{u}(k) := 1 - \mathrm{v}(k)^2$, so it is low or zero if the vast majority or all of the training vectors in it belong to the same class and it is maximal if the numbers of vectors of each class in it are equal. The impurity of the classification tree T is $\mathrm{u}(T) := |L|^{-1} \sum_{k \in \mathbf{leafs}(T)} |L \cap k| \, \mathrm{u}(k)$. If the impurity of a tree is zero, it classifies all training vectors correctly. It is always possible to find such a tree for a finite learning set. But such a tree may be too large and its ability to generalize the classification to other vectors than those in the learning set may be very bad. Therefore a two-phase procedure is applied, that uses a learning set L for growing a tree and a test set Q for subsequent pruning [3].

At the beginning of the tree building process the tree consists of only one node, the root, which is a leaf at the same time. As long as there is a leaf k of the tree, that contains more than a predefined number of training vectors a dimension $i \in \{1, \ldots, D\}$ and a decision value c is sought to split k into a left descendant k_l and right one k_r, such that each of them represents at least a certain factor of the vectors and the loss of impurity $\mathrm{u}(k) - \mathrm{u}(k_l) - \mathrm{u}(k_r)$ is maximized. The height of the whole tree is logarithmic in the number of training vectors $n := |L|$. The tree building process consists of sorting all training examples in each dimension as a preprocessing step and a D-fold scan over all remaining vectors for each inner node, where each vector is considered at most once in every tree level. So this phase needs time in $\mathrm{O}(D \times n \log n)$.

Let $\mathrm{r}_T(x) := 1 - \mathrm{v}_T(x)\mathrm{c}(x)$ be the *penalty* for the classification of vector x with tree T. This penalty is zero if T classifies x correctly and it is equal 2 if the classification is totally wrong. The classification penalty for a set Q is $\mathrm{R}_T(Q) := |Q|^{-1} \sum_{x \in Q} \mathrm{r}_T(x)$. It's easy to see that the penalty for the learning set L is equal to the impurity $\mathrm{u}(T)$.

The classification tree T gets pruned in a sequence of steps, where the main idea is to decrease the number of leafs $|T|$ in T as far as possible, while keeping the impurity low. In each step for a non-leaf node k of T the subtree T_k of T with root k is merged into a new leaf node, if the value

$a_T(k) := (|T_k| - 1)^{-1}\{|L \cap k|\, \mathrm{u}(k) - \sum_{k' \in \mathbf{leafs}(T_k)} |L \cap k'|\, \mathrm{u}(k')\}$ is minimal for k among all inner nodes of T. The trees constructed by iterative application of this procedure minimize for an increasing sequence of parameters a_i the number of leafs in a subtree of T with minimal costs $\mathcal{R}_{a_i} := \mathrm{u}(T) + a_i\,|T|$. These costs reward the purity and punish the size of a tree T. In each of the above-mentioned steps the classification penalty $\mathrm{R}_{T'}(Q)$ for the newly constructed tree T' is evaluated. The final tree T^* is that tree in this sequence for which the penalty is minimal. The time complexity for this second phase is in $\mathrm{O}(n \log^2 n)$, if the test set Q has about the same size as the learning set. The implementation has to maintain a heap structure for the $a_T(k)$ values that has to be updated in each step for all nodes on the path from the merged node k to the root.

3 α-β-Evaluation functions and their efficient implementation

We have seen that in the traditional form of the alpha-beta procedure the static evaluation function may return any value even if it is outside of the alpha-beta window, although this violates the constraints of the window evaluation scheme and one of the values α or β would suffice in this case and eventually would lead to the same result of the dynamic evaluation procedure. Beyond that we can show that a static evaluator that obeys the alpha-beta window can be implemented more efficiently than a classical evaluator.

An *α-β-evaluation function* is called with three arguments, a game position p and the boundaries α and β of an interval $[\alpha, \beta] \subseteq [-1, 1]$ and it returns a value from the interval $[\alpha, \beta]$. If the function returns the value v when called with the parameters p, -1 and 1, then it returns v as well, when called with p, α and β if $v \in [\alpha, \beta]$. If $v \leq \alpha$ or $v \geq \beta$, then α or β is returned respectively.

The classical evaluation function can easily be substituted by a corresponding α-β-evaluation function without changing the semantic of the alpha-beta procedure anyhow. But at least two advantages are obtained. The first is a neat integration of the static evaluation function into the dynamic search procedure, which are now both called with an alpha-beta window that determines the range of possible return values in advance. The second, more serious advantage concerns the speedup that can be achieved by alternative implementations of the altered function.

A classification tree based evaluation function can be adapted very easily to the new scheme. Each node k including the leafs has to represent additionally an interval $[\alpha_k, \beta_k]$ where α_k is the minimum and β_k is the maximum of all values of the leafs in the subtree with root k. The preprocessing time for changing the tree with a simple recursive procedure is only linear in the number of nodes. The new implementation of the static evaluation function is now given in Figure 1 in pseudo code. It is easy to see that this algorithm

```
function static_evaluation (p: position; α, β: real;
           k: evaluation_node): real;
begin
  for i := 1 to D do unknown [i] := true;
  while true do begin
    if k.β ≤ α then return (α);
    if k.α ≥ β then return (β);
    if leaf (k) then return (k.α);
    if unknown [k.feature] then begin
      vector [k.feature] := get_feature (p, k.feature);
      unknown [k.feature] := false
    end;
    if vector [k.feature] ≤ k.split_value
      then k := k.left
      else k := k.right
  end
end;
```

Figure 1: The classification tree based static α-β-evaluation function

implements the α-β-evaluation function. The function returns prematurely with one of the values α or β as soon as it is detected that all values in the leafs of a relevant subtree of the classification tree are outside of the alpha-beta window. So there may be many features which need not to be computed in this case.

Each feature of the given game position is computed at most once because they are stored in a vector for repeated later reference. On the other hand there is no need for all features to be computed during the static evaluation each time even if the computed value is inside of the alpha-beta window. This stems from the fact that not every feature needs to appear on the path from the root to a certain leaf of the classification tree.

The fact that the computation time required to evaluate the α-β-evaluation function is mainly determinated by the set of different features that have to be computed on a path from the root to a leaf may give rise to a new impurity function that is used to find the next split feature and split value for some node when growing the classification tree. So from a set of features that are equal likely to be chosen those should be preferred to the others that already appear on the path from the root to the node in question.

4 Bootstrap learning of evaluation functions

The learning of classification trees is very efficient, when a representative set of training examples for the function to be learned is given in advance. But often exactly this is impossible when the function is the static evaluation function for an interesting game, because the game theoretic value of most positions is unknown.

If only the rules of a game are given, the game theoretic values are obvious without reasoning only for positions at the end of the game via the so-called *trivial evaluation function*. When game tree evaluation procedures are applicable in reasonable time up to a search depth of d, the exact value can be obtained (in reasonable time) for all positions that are root of a winning strategy, i.e. a part of the game tree, for one of the players with height not exceeding d. So, if we start from scratch, we are only able to provide a learning set with feature vectors of game positions labeled with 1 or -1, if win or loss comes out within d moves, or labeled with 0, if the value is not known. This set then can be used to construct a heuristic evaluator that is sufficiently accurate for the aforesaid positions but needs less time than the dynamic evaluator.

Another problem is, to find relevant game positions, i.e. game positions that happen to appear in real games played by knowledgeable players. Randomly chosen positions may be too far off the conventional paths of play. An approximate solution to this problem takes game positions from computer simulated games. But these simulations mostly exhibit very poor play, because in the beginning only the trivial evaluation function can be used. The first heuristic evaluator may be a good evaluator with respect to the learning set and the test set, but it may be bad for game positions resulting from real games. Nevertheless there is real hope that it may be a large bit better than the trivial evaluator, that is unable to estimate any other positions than end positions.

The first step that involves playing a number of games and evaluating all occurring positions provides us with learning and test sets that are suitable for learning an advanced evaluation function when compared with the trivial one. And this gives us the chance to start over with the same procedure once more. So again a better evaluation function can be derived from the previous one. The *bootstrap learning algorithm* consists of the following steps:

1. Let f be the trivial evaluation function.

2. Play m games using alpha-beta search up to depth d_1 and function f with an added grain of chance (to derive different games), while collecting the set P of feature vectors of the positions.

3. Assign classification values to the elements of P, using alpha-beta search up to depth d_2 and function f.

4. Divide P randomly into sets L and Q. Build a classification tree T^* using L to grow and Q to prune it. Let f be the evaluation function stemming from T^*.

5. If f improved in step 4. and the limit for iterations is not exceeded then go to step 2. else stop.

In theory, if we had enough played games and if the evaluation function could be learned exactly, each step of the above-mentioned procedure would deliver an evaluation function that is correct for positions with a winning strategy that needs up to d more moves than the correctly evaluated positions of the function it is derived from. In practice, iterative application of the upgrade step eventually yields a good evaluation function that is bootstrapped from the trivial one in a kind of *heuristic retrograde analysis* [5].

Each step of the bootstrap procedure may construct a completely new classification tree from the classified feature vectors and throw the old tree away. Another possibility is the reuse of the old tree. The learning vectors of the new step can be fed into the old structure that contains a lot of already assembled knowledge. And after that the expanded tree is pruned again with a test set.

We have implemented the bootstrap learning procedure on a Sun workstation in C. We applied it in a number of different experiments with varying conditions to the not widely known game named malawi [7]. Malawi [9] is played on a 6×6 board. Each player owns six movable tiles that are initially positioned at the own baseline and twelve balls, two of them are initially on each of the own tiles. There are three possible kind of moves: The balls from one tile can be redistributed, at most one to each of the other tiles. A tile can be moved horizontally or vertically a number of steps according to the number of balls on it. If a tile cannot be moved to a position because this is occupied by a tile of the opponent, then all balls of this tile can be captured. The game ends with a win when one player can move *from* a position of the opponent's baseline. It ends with a loss when no balls are left or after 500 moves. Typical game positions permit about 40 different moves and the length of a played game is about 50 moves. We consider malawi to have about the same complexity as chess.

We used 24 different integer valued features to describe a malawi game position. The evaluation functions that resulted from our experiments had to demonstrate their strengths in a final contest. The winning function was derived by 467 iterations of the bootstrap procedure that played 100 games per step. The game positions were evaluated with the evaluation function of the last step in depth $d = 2$. The trees of the last step were expanded and then pruned. The resulting tree consisted of 66695 nodes. This evaluation function was able to classify 93.8 % of the positions before the end of the game correctly. The game program using this function was considerably stronger than the human experts in our team.

5 Conclusion

We have shown that the alpha-beta evaluation scheme in game playing programs can be extended from the dynamic search procedure to the static evaluation function without changing the semantic of the calling dynamic procedure. And we have presented an implementation of this generalized evaluation function with the help of modified classification trees that enormously accelerates the most inner loop computation of the algorithm that decides upon which move to do next. This speedup results in stronger play because the search depth of the dynamic procedure can be increased.

We have shown how the trivial evaluation function that results directly from the rules of a game can be improved step by step. The bootstrap procedure starts from scratch and iteratively generates a knowledgeable evaluation function by hoisting up the knowledge level by level of the game tree. We have reported on our experiments with this kind of evaluation function learning for the game malawi. Our experience is that bootstrap learning of α-β-evaluation functions is a fast method to bring forth knowledgeable and efficient evaluation functions.

Acknowledgements

We would like to thank Sven Schuierer for helpful discussions and an anonymous referee for suggested improvements on an earlier draft.

References

[1] S. G. Akl. Checkers-playing programs. In S. C. Shapiro and D. Eckroth, editors, *Encyclopedia of Artificial Intelligence*, pages 88–93. John Wiley & Sons, New York, 1987.

[2] R. Banerji. Game playing. In S. C. Shapiro and D. Eckroth, editors, *Encyclopedia of Artificial Intelligence*, pages 312–319. John Wiley & Sons, New York, 1987.

[3] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, Belmont, California, 1984.

[4] S. E. Fahlman. Final report on information processing research — July 1987 to July 1990. Technical Report CMU-CS-92-156, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1992.

[5] R. Gasser. Applying retrograde analysis to nine mens morris. In D. Levy and D. Beal, editors, *Heuristic Programming in Artificial Intelligence 2*, pages 161–173. Ellis Horwood, 1991.

[6] G. M. Gupton. Genetic learning algorithm applied to the game of othello. In D. Levy and D. Beal, editors, *Heuristic Programming in Artificial Intelligence*, pages 241–254. Ellis Horwood, 1989.

[7] C. Hense. Lernen von Klassifikationsbäumen für Spielsituationen. Diplomarbeit, Albert-Ludwigs-Universität Freiburg, June 1992.

[8] D. Knuth and R. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326, 1975.

[9] G. Kody. Malawi. Piatnik-Verlag, Wien.

[10] K.-F. Lee and S. Mahajan. A pattern classification approach to evaluation function learning. *Artificial Intelligence*, 36:1–25, 1988.

[11] T. Marsland. Computer chess methods. In S. C. Shapiro and D. Eckroth, editors, *Encyclopedia of Artificial Intelligence*, pages 159–171. John Wiley & Sons, New York, 1987.

[12] J. Pearl. The solution for the branching factor of the alpha-beta pruning algorithm and its optimality. *Commun. ACM*, 25(8):559–564, August 1982.

[13] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, Reading, MA, 1984.

[14] J. R. Quinlan. Learning efficient classification procedures and their application to chess end games. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning, An Artificial Intelligence Approach*, chapter 15, pages 463–482. Springer-Verlag, Berlin, 1984.

[15] A. Samuel. Some studies in machine learning using the game of checkers. In E. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 71–105. McGraw-Hill, New York, 1963.

[16] A. Samuel. Some studies in machine learning using the game of checkers. ii — recent progress. *IBM J. Res. Develop.*, 11(6):601–617, Nov. 1967.