6th International Conference on Applied Informatics
Eger, Hungary, January 27–31, 2004.

A solution for building evaluation functions in two player games

Gábor Balázsfalvi

University of Debrecen, Institute of Informatics
e-mail: bg0005@stud.unideb.hu

Abstract

Two player games often use some heuristic in decision-making. In the Minimax or Alphabeta methods [3, 4], however, sometimes the game can build the whole tree of the game, it's often impossible. Other games like Nim may use perfect evaluation functions while playing. We have developed a method for building heuristical evaluation functions for games; which can not use these methods (the game tree or perfect evaluation). Our method can be generally applied, if some important features can be extracted from every state of the game. This is an iterative method, a genetic algorithm; which guarantees a convergence to the optimal evaluation function [2].

Key Words: Genetic algorithm, two player games, evaluation function, heuristic

1. Introduction

When an artificial intelligence driven player makes its decision, it often uses a part of the game tree. There are other methods than Minimax[1], but Minimax with some acceleration[2] is better in some cases. It is a keypoint of the Minimax to find a good evaluation function. Weighted linear functions are good because they can be quickly computed. The Canonical Genetic Algorithm (CGA) [1, 2] is used to solve optimization problems of type: $\max\left\{f(b)|b\in\{0,1\}^l\right\}$. But the weights of our functions are real numbers, so CGA cannot be used directly. Our genetic algorithm is introduced in section 5. In section 2 we describe the Minimax algorithm.

[1] e.g. SSS^*
[2] e.g. NegaScout [5]

2. The Minimax algorithm and the Alphabeta pruning

Two player games with perfect information have an initial state. There are operators, which can modify a state. There are two players; their names are Max and Min. There is a utility function, which gives a score at the end of a play; this is when a terminal state is reached. The utility function gives positive value when Max wins, negative when Min wins, and zero when the match is a draw.

The function *minimax-decision* gives back an operator from a game.

```
function minimax-decision(game) returns operator

for each op in operators[game] do

value[op]← minimax-value(op(game),game,−∞,∞);

end

return op where value[op] is maximal;
```

The original minimax algorithm goes down in the game-tree to the leaves. When a leaf node is found, a utility function tells its utility value for the player MAX. When the number of the states of a game is too big, we can not always use a utility function. Then we use an evaluation function, when a depth in the recursion is reached. This can be perfect, or heuristical. A perfect function gives the value on a state which were be given by the minimax-value on the same state.[3] There are states that we do not need to evaluate. This is called pruning the game tree. One of these pruning technics is the Alphabeta pruning. There are a lot of methods that makes minimax faster. One of them is NegaScout [5], see code. The function minimax-value gets a state, a game, and gives back a utility value:

```
function minimax-value(state,game,α,β) returns goodness
 if terminal-test(state) then return utility[state];
 a ← alpha;
 b ← beta;
 for each s in children[state] do
  t ← -minimax-value( s, -b, -a );
  if (t > a) AND (t < β) AND (s is not the first child)
     AND (not at the maximal depth) then
     a ← −minimax-value( s, −β, −t );
  a ← max( a, t );
  if a ≥ β then return a;
  b ← a + 1;
```

[3]For example in NIM. In this case, we don't need search the game-tree, the function tells us which move is the best.

```
end
return a;
```

3. What are the features of a game?

We are working with two player games with perfect information [3, 4]. In the studies of Artificial Intelligence, there are correct mathematics on how these games can be described by sets and functions. For example, in chess, we can describe a state of a game with the chessmen on the board, with their position, and the fact that the actual turn belongs to the White player or to the Black one. In a game like chess these features may be the number of chessmen, their value in chess, their possibilities for attack, etc. All in all, the features of a game are some kinds of abstract things, while when they are extracted, they become real numbers. These real numbers are also denoted by 'features'.

4. Feature extraction

Let the set of the states in a game notated by Γ. The features of a $\gamma \in \Gamma$ state are $t_1^\gamma, ...t_n^\gamma$, where n is determined by the game, and is constant while playing. This is the main point of the algorithm. If n is big, then the convergence will be slow, but if n is small, then the evaluation function will be weak. A h evaluation function then will evaluate a heuristic value from a γ state by these features. h has n pieces of weights, which are real numbers denoted by $s_1^h, ..., s_n^h$. These are constants through the game. Then, the value of $h(\gamma)$ is:

$$h(\gamma) = \sum_{i=1}^{n} s_i^h * t_i^\gamma.$$

Playing with Minimax, this should be multiplied by -1 at Min's turn. When Negamax is applied, this multiplication is included. From now on, the building method is modifying the weights. For this, we have used a slightly extended genetic algorithm.

5. Modifying the weights

A set of initial population of evaluation functions (denoted by Δ_1) contains M elements with randomly selected weights relatively far from zero (their absolute value is larger then 1). We should give a higher weight to the more important features.[4]

Let us take two players with modifiable evaluation function. Set their evaluation functions to the elements of Δ_1, so that two evaluation functions can be compared.

[4]This is not important, but the convergence will be faster.

The evaluation function of the winner then takes J scores, or both take $L < J$ when the match was a draw. One can get negative scores when notable mistakes are made (for example long playing, but this is an advanced method and slows down the convergence). When every pair of evaluation functions are compared, we get the average scores of them. Now, we make an inter-population (this will be a multiset, so one evaluation function can be in it more than once): let us get a uniformly distributed random variable x from the interval $[0, 1[$, and an evaluation function h with score z. If $x \leq z$, then we put h to the inter-population (again, if it is already there), and the new score of h is $z - x$. This is done while the number of evaluation functions in the inter-population reaches a threshold (approximately the number of evaluation functions in Δ_1), or no more functions can be put in.

When the inter-population is ready, we get M random pairs from it. From a pair of evaluation functions, we make a new evaluation function by averaging their weights. These new weights should be modified by multiplying them by random variables from a Gaussian distribution with expected value of 1, and with a small dispersion (this variable must not be a negative value, and must not be larger than two, and it should be very close to one). This may help us out from a local maximum, but because of this, we will never reach the global maximum (if it exists at all). When a very good evaluation function is found, we should save it to a separate set. We can modify this dispersion while better and better solutions are found.

For a faster convergence, we may take the advantage of when a set of weights w_i yields states γ_1,γ_2,..., then multiplying them with a positive x $x * w_i$ yields the same states. So, the weight of one feature can be standard one. This should be the smallest one (the weight of the less important feature).

5.1. Why does this method give good results?

This is a modified genetic algorithm. The canonical genetic algorithm (CGA) is described in [2]. The CGA is the discrete model of a nonlinear optimization problem. We can find that the CGA does not converge, but reaches the optimal solution, if one exists. This is mainly because the number of the states is finite. In our algorithm, the number of all available weight-vectors is infinite. And because of we modify the weights when we multiply them by random variables, the actual solution can be worse than another; which was dropped by the method. So, we should save the winner solutions.

But globally, the solutions are getting better and better, so it should be called "convergence". In practice, a quasi-optimal solution can be reached.

6. Examples, results

We used this method in making an amoba game, and a chess game. In amoba, we used only three features, the number of twos, threes and fours. When a five is got, the game is over. The weight of twos is set to constant one, so it was a

two-dimensional problem. The method converged very fast, but the game made a lot of mistakes. Using some more features, and a Minimax algorithm with depth of three, the computer became a professional player. It won games opposing other computer players, as far as humans.

In chess, much more features were used. First, we used only the material-values of the chessmen. This converged slightly fast, and gave the standard result (about the material-values in chess: 1, 3, 3, 5, 9). But it cannot play well with human, nor with other computer players. Using the number and importance of attacks, we got a good chess player; which won most games opposing other computer players that did not use any kind of databases, and worked much faster. We used a Minimax player with depth of three and four in the recursion, while these players think six-seven steps forward. Neither human players could overcome this (they were not professional players, they were our friends).

A. Example amoba play

In this example play the human player(me) has began. In this case, we used a three-step forward looking Minimax algorithm:

Human 10 11; Computer 10 10;

Human 11 11; Computer 9 11;

Human 11 9; Computer 11 10;

Human 12 10; Computer 9 10;

Human 13 11; Computer 8 10;

Human 7 10; Computer 10 8;

Human 12 11; Computer 14 11;

Human 9 13; Computer 10 12;

Human 7 9; Computer 11 13;

Human 12 14; Computer 9 9;

Human 14 12; Computer 15 13;

Human 7 11; Computer 9 8;

Human 9 7; Computer 9 12 – Computer has won.

B. Chess player results

In this case, we used material values, and one–one constant for multiplying material values for attack and be attacked. Results are about:

material values:

Pawn – 1; Knight – 3.1; Bishop – 3.5; Rook – 5.5; Queen – 8.9; King – 10

multiplier constants: for attacking – $\frac{1}{5}$; for being attacked – $\frac{1}{6}$

Acknowledgements

Many thanks to Dr. Magda Várterész, who helped me with suggests, and finding useful papers.

References

[1] Whitley, D.: *A Genetic Algorithm Tutorial, Statistics and Computing 4(2), 65-85p, 1994*

[2] Rudolph, G.: *Convergence Analysis of Canonical Genetic Algorithms, IEEE Transactions on Neural Networks 5(1), 96-101p, 1994*

[3] Futó, I. (ed.): *Mesterséges Intelligencia*, Aula, Budapest, 1999.

[4] Russel, S. J. - Norvig, P.: *Artificial Intelligence. A Modern Approach.*, Prentice Hall, 1995 (Hungarian language edition published Panem Könyvkiadó, Budapest, 2000)

[5] A. Reinefeld: *Spielbaum-Suchverfahren. Informatik-Fachbericht 200, Springer-Verlag, Berlin (1989), ISBN 3-540-50742-6 (http://www.zib.de/reinefeld/bib/83icca.pdf)*