The Alpha-Beta Procedure

Can we estimate the efficiency benefit of the alpha-beta method?

Suppose that there is a game that always allows a player to choose among \( b \) different moves, and we want to look \( d \) moves ahead.

Then our search tree has \( b^d \) leaves.

Therefore, if we do not use alpha-beta pruning, we would have to apply the static evaluation function \( N_d = b^d \) times.

Of course, the efficiency gain by the alpha-beta method always depends on the rules and the current configuration of the game.

However, if we assume that somehow new children of a node are explored in a particular order - those nodes \( p \) are explored first that will yield maximum values \( e(p) \) at depth \( d \) for MAX and minimum values for MIN - the number of nodes to be evaluated is:

\[
N_d = \begin{cases} 
2b^{d/2} - 1 & \text{for even } d \\
2b^{(d+1)/2} + b^{(d-1)/2} - 1 & \text{for odd } d 
\end{cases}
\]

Therefore, the actual number \( N_d \) can range from about \( 2b^{d/2} \) (best case) to \( b^d \) (worst case).

This means that in the best case the alpha-beta technique enables us to look ahead almost twice as far as without it in the same amount of time.

In order to get close to the best case, we can compute \( e(p) \) immediately for every new node that we expand and use this value as an estimate for the Minimax value that the node will receive after expanding its successors until depth \( d \).

We can then use these estimates to expand the most likely candidates first (greatest \( e(p) \) for MAX, smallest for MIN).

Of course, this pre-sorting of nodes requires us to compute the static evaluation function \( e(p) \) not only for the leaves of our search tree, but also for all of its inner nodes that we create.

However, in most cases, pre-sorting will substantially increase the algorithm’s efficiency.

The better our function \( e(p) \) captures the actual standing of the game in configuration \( p \), the greater will be the efficiency gain achieved by the pre-sorting method.

Even if you do not want to apply \( e(p) \) to inner nodes, you should at least do a simple check whether in configuration \( p \) one of the players has already won or no more moves are possible.

If one of the players has won, this simplified version of \( e(p) \) returns the value \( \infty \) or \( -\infty \) if in configuration \( p \) the player MAX or MIN, respectively, has won.

It returns 0 (draw) if no more moves are possible.

This way, no unnecessary - and likely misleading - analysis of impossible future configurations can occur.

Timing Issues

It is very difficult to predict for a given game situation how many operations a depth \( d \) look-ahead will require.

Since we want the computer to respond within a certain amount of time, it is a good idea to apply the idea of iterative deepening.

First, the computer finds the best move according to a one-move look-ahead search.

Then, the computer determines the best move for a two-move look-ahead, and remembers it as the new best move.

This is continued until the time runs out. Then the currently remembered best move is executed.
How to Find Static Evaluation Functions

Often, a static evaluation function $e(p)$ first computes an appropriate feature vector $f(p)$ that contains information about features of the current game configuration that are important for its evaluation.

There is also a weight vector $w(p)$ that indicates the weight (= importance) of each feature for the assessment of the current situation.

Then $e(p)$ is simply computed as the dot product of $f(p)$ and $w(p)$.

Both the identification of the most relevant features and the correct estimation of their relative importance are crucial for the strength of a game-playing program.

Once we have found suitable features, the weights can be adapted algorithmically.

This can be achieved, for example, with an artificial neural network.

So the biggest problem consists in extracting the most informative features from a game configuration.

Let us look at an example: Chinese Checkers.

Chinese Checkers

Idea for important feature:

- assign positional values
- sum values for all pieces of each player
- feature “progress” is difference of sum between players

Chinese Checkers

Another important feature:

- For successful play, no piece should be “left behind”
- Therefore add another feature “coherence”: Difference between the players in terms of the smallest positional value for any of their pieces.

Weights used in sample program:

- 1 for progress
- 2 for coherence
Isola

Your biggest programming assignment in this course will be the development of a program playing the game Isola.

In order to win the tournament and receive an incredibly valuable prize, you will have to write a static evaluation function that
• assesses a game configuration accurately and
• can be computed efficiently.

Rules of Isola:
• Each of the two players has one piece.
• The board has 7x7 positions which initially contain squares, except for the initial positions of the pieces.
• A move consists of two subsequent actions:
  – moving one’s piece to a neighboring (horizontally, vertically, or diagonally) field that contains a square but not the opponent’s piece,
  – removing any square with no piece on it.
• If a player cannot move any more, he/she loses the game.

Initial Configuration:

If in this situation O is to move, then X is the winner:

If X is to move, he/she can just move left and remove the square between X and O, and also wins the game.

You can start thinking about an appropriate evaluation function for this game.

You may even consider revising the Minimax and alpha-beta search algorithm to reduce the enormous branching factor in the search tree for Isola.

We will further discuss the game and the Haskell interface for the tournament in the following weeks.