The Minimax Algorithm

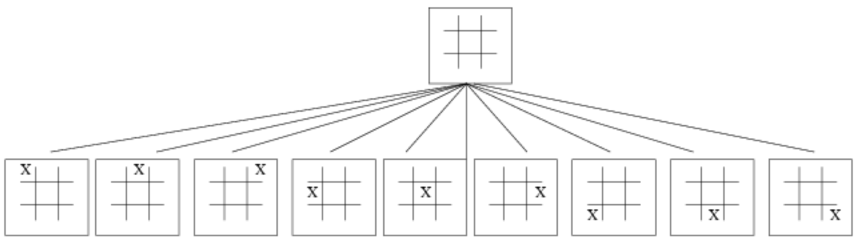
Glenn Strong
Glenn.Strong@cs.tcd.ie

February 10, 2011

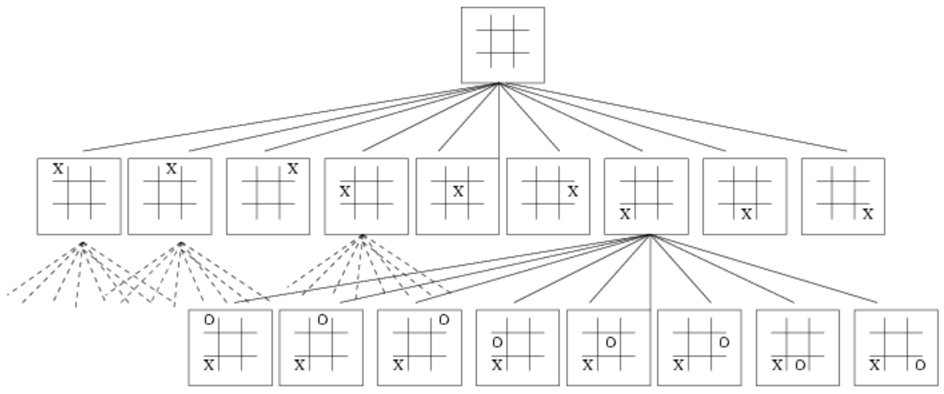
1 Motivation

Here is a simple algorithm that will not work to calculate the winning move in a game:

- From the current position, calculate all the possible moves that could be made from that position. For a game of noughts and crosses the result might look like:



- Expand each of these new positions so that they include the possible moves for the *other* player:



- Continue this expansion until a winning position for the required player is found.

This algorithm *will* work, in the sense that it will locate a series of winning moves for the game, but the cost of performing that calculation is enormous. At the first level of the graph there will be one board. At the next 9, at the next there will be $9 * 8$ at the level past that there will be $9 * 8 * 7$. In total:

$$\sum_{0}^{n=8} \frac{9!}{(n+1)!} = 986410$$

This is not so big that we cannot calculate it, but it is alarming since noughts and crosses is such a simple game.

2 Minimax

For a game which has a more complex tree than noughts and crosses (technically, where the *branching factor* is higher) this simplistic approach cannot work. The tree we are building will simply be too large. We need some mechanism to cut down the number of subtrees that are built – ideally we will be able to eliminate moves which do not look promising and concentrate on those which seem likely to lead to a positive outcome (a win).

We introduce the idea of a function (which we will call the *cost function* which assigns a number to a game position representing how likely that position is to lead to a win. This score can be calculated for each playable position.

If we knew that the scoring function was perfect then we could simply proceed to a solution by taking the move with the highest score and making that move. We might then expect the opponent to play the move which has their highest score (which will also be the move with the lowest score for us), and so on. However, the scoring function is unlikely to be perfect. It is therefore advisable to explore the possible moves further — for example, during a piece exchange in chess a superficially unattractive move may turn out to be advantageous if pursued further. The depth of the search is known as the *ply* of the search (so a 4-ply search examines the tree to a depth of 4). In order to pursue the tree further we will have to make guesses as to how the opponent will play. The cost function can be used again to evaluate how the opponent is likely to play. After evaluating some number of moves ahead we examine the total value of the cost to each player. The goal is to find a move which will *maximise* the value of our move and will *minimise* the value of the opponents moves. The algorithm used is the *minimax search procedure.*

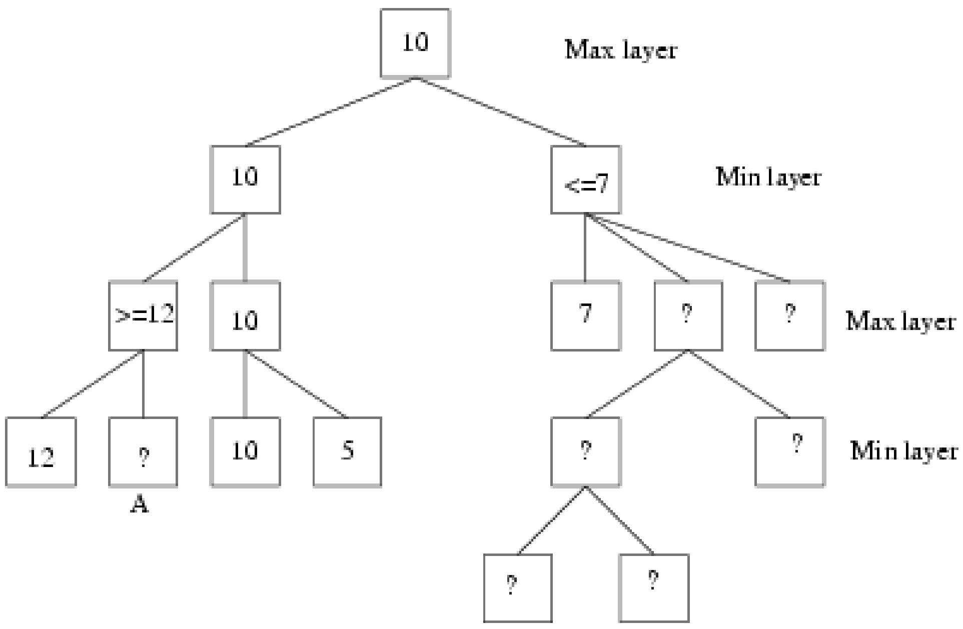
3 Smarter Minimax

The minimax search as described still requires the tree to be searched to a greater extent than is practical in order to find a good move. We need some way of avoiding evaluations on parts of the tree that seem unlikely to lead to success and concentrating efforts on the moves that look hopeful.

Consider this tree, where the numbers indicate scores and the question marks indicate moves which have not yet had scores evaluated:

```
function MINIMAX(N) is
begin
if N is deep enough then
        return the estimated score of this leaf
else
        Let N1, N2, .., Nm be the successors of N;
        if N is a Min node then
                return min{MINIMAX(N1), .., MINIMAX(Nm)}
        else
                return max{MINIMAX(N1), .., MINIMAX(Nm)}
end MINIMAX;
```

Figure 1: Minimax algorithm. Two player, perfect information zero-sum game



The key to this diagram is that the unevaluated locations will not affect the score of the top node under minimax. Consider the node marked “?” at the bottom left of the tree (labelled “A”). It does not matter what this node evaluates to because the min player will have already taken the decision to play towards the node valued 10, and will have decided this on the basis that playing the other node would reveal a more valuable move for the max player. This is called a *beta-cutoff*.

Now consider the nodes labelled “?” on the right-hand subtree. It does not matter what they evaluate to since the max player will always want to play the move scored 10 in order to keep the min player from having the option of playing the “7” move. This is termed an *alpha cutoff*. Making this decision cuts off a complete subtree from consideration on the right-hand side of the tree.

By applying α and β cutoffs we have managed to avoid some needless evaluations that simple minimax would have performed.

The algorithm for minimax with α-β pruning is similar to the original minimax algorithm but avoids some recursive steps by handing around values which represent the best and worst moves that will be made so far (which correspond to the tree nodes marked with inequalities in the diagram).

```
function MINIMAX-AB(N, A, B) is
  if N is deep enough then
    return the estimated score of this leaf
  else
    alpha = a; beta = b;
    if N is a Min node then
       For each successor Ni of N
         beta = min{beta, MINIMAX-AB(Ni, alpha, beta)}
         if alpha >= beta then return alpha
       end for
       return beta
    else
       For each successor Ni of N
         alpha = max{alpha, MINIMAX-AB(Ni, alpha, beta)}
         if alpha >= beta then return beta
       end for
       return alpha
    end if
  end if
end function
```

Figure 2: Minimax algorithm with α-β pruning.

4 Improvements

A number of further improvements to the minimax algorithm are possible to deal with certain undesirable situations.

- Wait for quiescent states. It is possible that when a tree node is expanded slightly the state could change drastically. Such information may have a major effect on the choice of which move should be played. To ensure that short-term considerations do not influence our choice of move we might choose to expand the tree until no such major changes occur. This helps to avoid the so-called *horizon effect* where a very bad move for a player can be temporarily delayed so that it does not appear in the section of the tree that minimax explores (it does not eliminate this possibility, only ameliorate it).

- Secondary search. Another way to tackle the horizon effect is to double-check moves before making them. Once minimax has proposed a move it may be a good idea to expand the search tree a few levels under the node at the bottom of the tree that caused the move to be selected (in order to verify that no nasty surprises are waiting once the move is played).