Introduction to Artificial Intelligence

Unit # 4

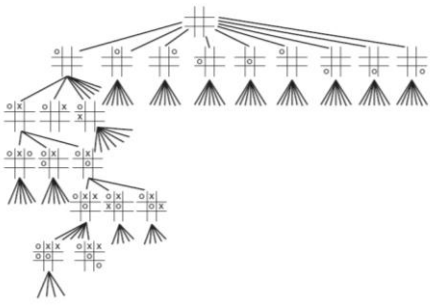
Acknowledgement

- The slides of this lecture have been taken from the lecture slides of CS307 – "Introduction to Artificial Intelligence" by Dr. Sajjad Haider.

Game Tree

- Many two-player games can be efficiently represented using trees, called **game trees**.
- A game tree is an instance of a tree in which the root node represents the state before any moves have been made, the nodes in the tree represent possible states of the game (or **positions**), and arcs in the tree represent moves.
- It is usual to represent the two players' moves on alternate levels of the game tree, so that all edges leading from the root node to the first level represent possible moves for the first player, and edges from the first level to the second represent moves for the second player, and so on.

Game Tree of Tic-Tac-Toe



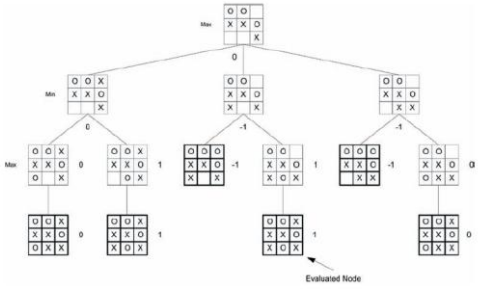
Search in Game Trees

- One approach to playing a game might be for the computer to use a tree search algorithm such as depth-first or breadth-first search, looking for a goal state (i.e., a final state of the game where the computer has won).
- For Tic-Tac-Toe, the branching factor of the root node is 9 because there are nine squares in which the computer can place its first nought.
- It can be inefficient for a computer to exhaustively search the tree because it has a maximum depth of 9 and a maximum branching factor of 9, meaning there are approximately 9 x 8 x 7 . . . 2 x 1 nodes in the tree, which means more than 350,000 nodes to examine.
- Chess has an average branching factor of about 35 and games often go to 50 moves by each player, so the search tree has about 35^{100} or 10^{154} nodes.
- The Deep Blue chess computer which defeated Kasparov in 1997 would typically search to a depth of between six and sixteen plies to a maximum of forty plies in some situations.

Evaluation Function

- For a computer to use this tree to make decisions about moves in a game of tic-tac-toe, it needs to use an **evaluation function**, which enables it to decide whether a given position in the game is good or bad.
- If we use exhaustive search, then we only need a function that can recognize a win, a loss, and a draw.
- Then, the computer can treat "win" states as goal nodes and carry out search in the normal way.

Minimax Algorithm for Tic-Tac-Toe

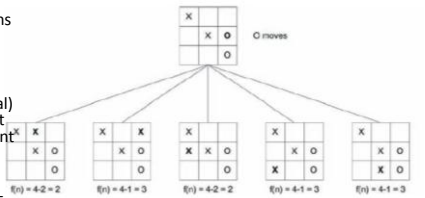


Ply and Move

- In discussing game trees, we use the concept of **ply**, which refers to the depth of the tree.
- When a computer evaluates a game tree to ply 5, it is examining the tree to a depth of 5. The 4th ply in a game tree is the level at depth 4 below the root node.
- Because the games we are talking about involve two players, sequential plies in the tree will alternately represent the two players. Hence, a game tree with a ply of 8 will represent a total of eight choices in the game, which corresponds to four moves for each player.
- It is usual to use the word *ply* to represent a single level of choice in the game tree, but for the word *move* to represent two such choices—one for each player (Source: wikipedia)

Static Evaluation Function for Tic-Tac-Toe

- The static evaluation function is defined as the number of possible win positions not blocked by the opponent minus the number of possible win positions (row, column, and diagonal) for the opponent not blocked by the current player:
- *f* (*n*) = *win_positions-lose_positions*



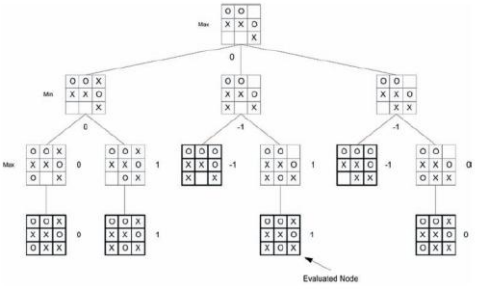
Minimax Algorithm

- The minimax algorithm is a useful method for simple two-player games. It is a method for selecting the best move given an alternating game where each player opposes the other working toward a mutually exclusive goal.
- In a two-person game, you must assume that your opponent has the same knowledge that you do and applies it as well as you do. So at each stage of the game you must assume your opponent makes the **best available move**. This is the basis of the minimax procedure.

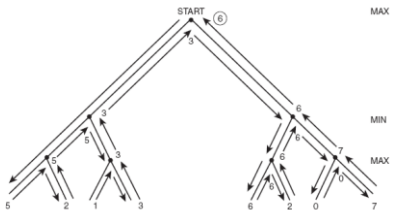
MiniMax Algorithm

- It is assumed that a suitable static evaluation function is available, which is able to give an overall score to a given position.
- In applying Minimax, the static evaluator will only be used on leaf nodes, and the values of the leaf nodes will be filtered up through the tree, to pick out the best path that the computer can achieve.
- The principle behind Minimax is that a path through the tree is chosen by assuming that at its turn (a **max node**), the computer will choose the move that will give the highest eventual static evaluation, and that at the human opponent's turn (a **min node**), he or she will choose the move that will give the lowest static evaluation.
- In other words, we assume that each player makes the next move that benefits them the most.

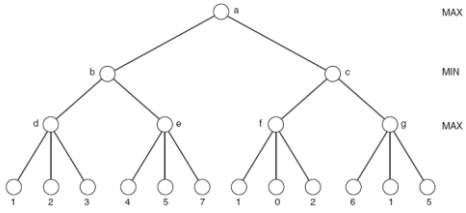
Minimax Algorithm for Tic-Tac-Toe



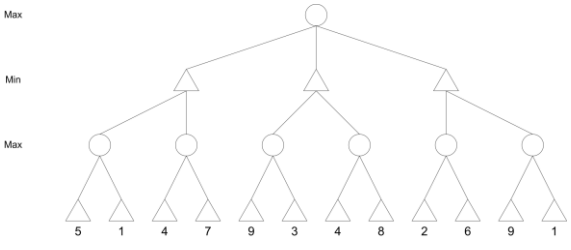
Example of Minimax Algorithm



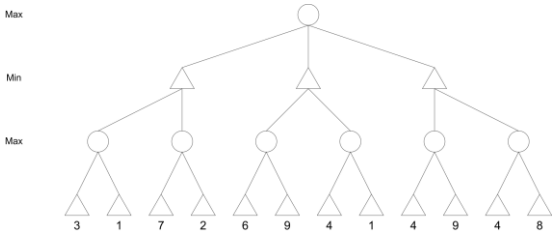
Example I



Example II



Example III

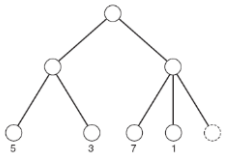


Lookahead and Horizon Effect

- Minimax is a very simple algorithm and is unsuitable for use in many games, such as chess, where the game tree is extremely large.
- In such cases, **bounded lookahead** is very commonly used and can be combined with Minimax.
- The idea of bounded lookahead is that the search tree is only examined to a particular depth. All nodes at this depth are considered to be leaf nodes and are evaluated using a static evaluation function.
- When we employ lookahead strategy, we suffer from what is called the **horizon effect**.
- When we can't see beyond the horizon, it becomes easier to make a move that looks good now, but leads to problems later as we move further into this subtree.

Alpha-Beta Pruning

- Using alpha–beta pruning, it is possible to remove sections of the game tree that are not worth examining, to make searching for a good move more efficient.
- The principle behind alpha–beta pruning is that if a move is determined to be worse than another move that has already been examined, then further examining the possible consequences of that worse move is pointless.



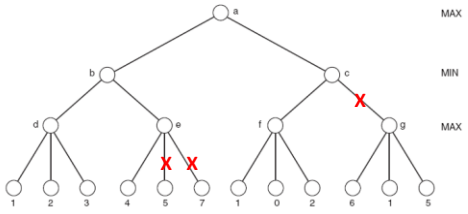
Working of Alpha-Beta Pruning

- During the depth-first search of the game tree, we calculate and maintain two variables *alpha* and *beta*.
- The alpha variable defines the best move that can be made to maximize (our best move) and the beta variable defines the best move that can be made to minimize (the opposing best move).
- While we traverse the tree, if alpha is ever greater than or equal to beta, then the opponent's move forces us into a worse position.
- In this case, we avoid evaluating this branch any further.

Working of Alpha-Beta Pruning

- The algorithm maintains two values, alpha and beta, which represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of respectively.
- Initially alpha is negative infinity and beta is positive infinity.
- Together alpha and beta provides a window of possible scores. We will never choose to make moves that score less than alpha and our opponent will never let us make moves scoring more than beta. The score we finally achieve must lie between the two.
- As the recursion progresses the "window" becomes smaller. When beta becomes less than alpha, it means that the current position cannot be the result of best play by both players and hence need not be explored further.

Example I (Revisited)



Only 7 nodes (out of 12) explored with alpha-beta pruning.

Working of Alpha-Beta for Example I

Step	Node	Alpha	Beta	Notes
1	a	$-\infty$	∞	Alpha starts as $-\infty$ and beta starts as ∞.
2	b	$-\infty$	∞	
3	d	$-\infty$	∞	
4	d	1	∞	
5	d	2	∞	
6	d	3	∞	At this stage, we have examined the three children of d and have obtained an alpha value of 3, which is passed back up to node b.
7	b	$-\infty$	3	At this min node, we can clearly achieve a score of 3 or better (lower). Now we need to examine the children of e to see if we can get a lower score.
8	e	$-\infty$	3	

Working of Alpha-Beta for Example I (Cont'd)

8	e	$-\infty$	3	
9	e	4	3	CUT-OFF. A score of 4 can be obtained from the first child of e. Min clearly will do better to choose d rather than e because if he chooses e, max can get at least 4, which is worse for min than 3. Hence, we can now ignore the other children of e.
10	a	3	∞	The value of 3 has been passed back up to the root node, a. Hence, max now knows that he can score at least 3. He now needs to see if he can do better.
11	c	3	∞	
12	f	3	∞	We now examine the three children of f and find that none of them is better than 3. So, we pass back a value of 3 to c.
13	c	3	3	CUT-OFF. Max has already found that by taking the left-hand branch, he can achieve a score of 3. Now it seems that if he chooses the right-hand branch, min can choose f, which will mean he can only achieve a score of 2. So cut-off can now occur because there is no need to examine g or its children.

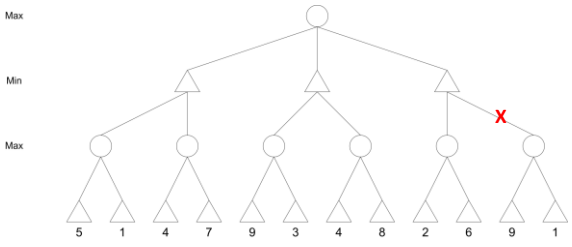
Alpha-beta pruning

```
alpha-beta(player,board,alpha,beta)
    if(game over in current board position)
        return winner
    children = all legal moves for player from this board
    if(max's turn)
        for each child
          score = alpha-beta(other player,child,alpha,beta)
          if score > alpha then
              alpha = score (we have found a better best move)
          if alpha >= beta then return alpha (cut off)
              return alpha (this is our best move)
    else (min's turn)
        for each child
          score = alpha-beta(other player,child,alpha,beta)
          if score < beta then
              beta = score (opponent has found a better worse move)
          if alpha >= beta then
              return beta (cut off)
          return beta (this is the opponent's best move)
```

Alpha-beta pruning - Pseudocode
(Source: Wikipedia)

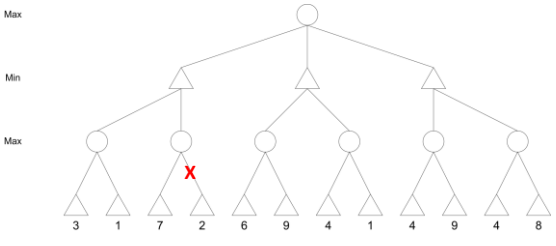
```
function alphabeta(node, depth, α, β, Player)
    if  depth = 0 or node is a terminal node
        return the heuristic value of node
    if  Player = MaxPlayer
        for each child of node
            α := max(α, alphabeta(child, depth-1, α, β, not(Player) ))
            if β≤α
                break                             (* Beta cut-off *)
        return α
    else
        for each child of node
            β := min(β, alphabeta(child, depth-1, α, β, not(Player) ))
            if β≤α
                break                             (* Alpha cut-off *)
        return β
(* Initial call *)
alphabeta(origin, depth, -infinity, +infinity, MaxPlayer)
```


Example II



10 nodes (out of 12) explored with alpha-beta pruning.

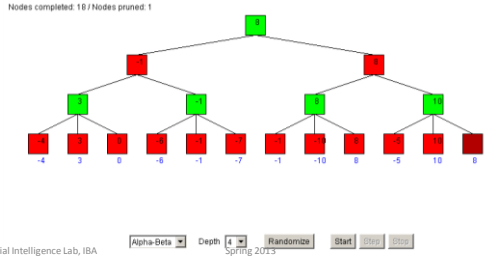
Example III



11 nodes (out of 12) explored with alpha-beta pruning.

Java Applet

- http://www.ocf.berkeley.edu/~yosenl/extras/alphabeta/alphabeta.html



Advantages of Alpha-Beta Pruning

- The alpha–beta pruning method provides its best performance when the game tree is arranged such that the best choice at each level is the first one (i.e., the left-most choice) to be examined by the algorithm.
- With such a game tree, a Minimax algorithm using alpha–beta cut-off will examine a game tree to double the depth that a Minimax algorithm without alpha–beta pruning would examine in the same number of steps.