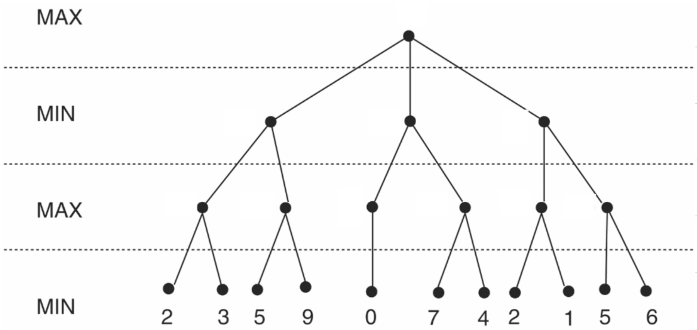
COMP 6721 Applied Artificial Intelligence (Fall 2023)

Worksheet #2: Adversarial Search

Game of Nim. Play a game of Nim against your team mate, starting with 7 tokens: circle the tokens that you split into the two new piles at each move (piles must be non-empty and differently-sized). Player “MIN” starts:

(MIN) • • • • • • • (MAX)

MiniMax. Let’s apply the MiniMax algorithm discussed in the lecture on an example (fixed ply depth of 3):



1. build complete game tree
2. label each level according to player’s turn (MAX or MIN)
3. label leaves with a utility function to determine the outcome of the game
   - e.g., (0, 1) or (-1, 0, 1)
4. propagate this value up:
   - if parent=MAX, give it max value of children
   - if parent=MIN, give it min value of children
5. select best next move for player at root as the move leading to the child with the highest value (for MAX) or lowest values (for MIN)

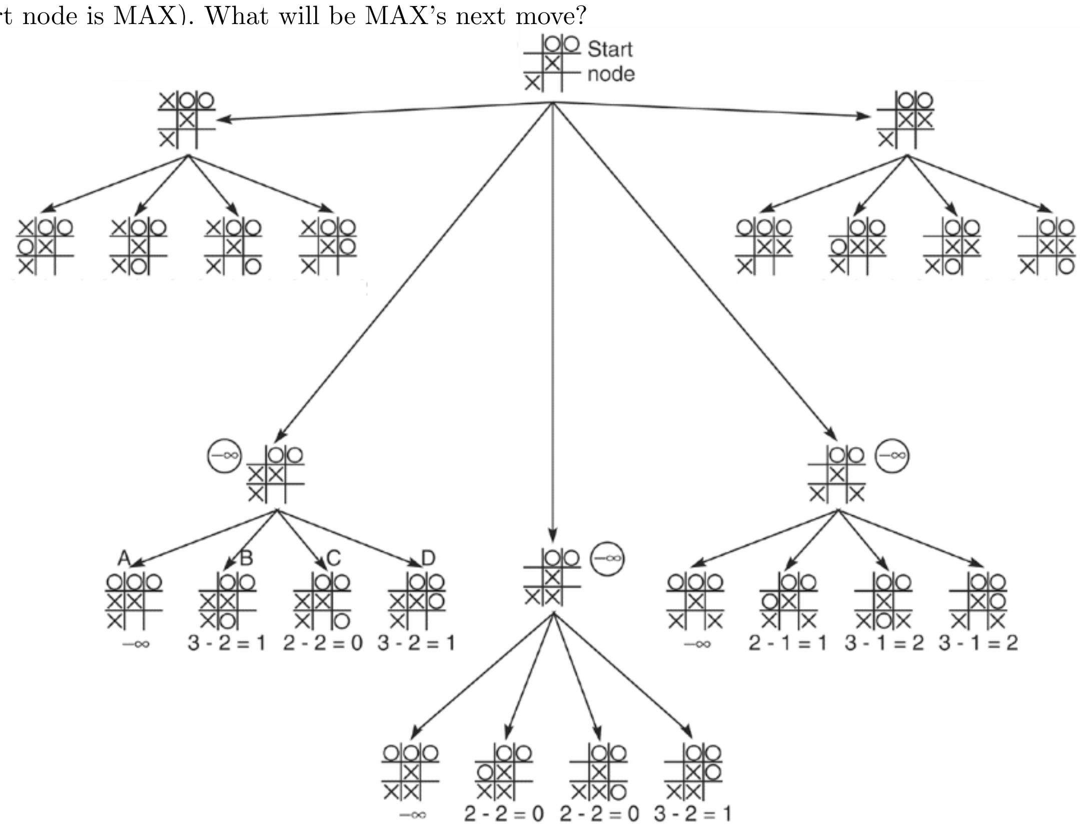
Leaf nodes show the actual heuristic value *e(n)*

1. Apply Step 4: Add the *back-up* heuristic values of all non-leaf (internal) nodes, up to the root
2. Apply Step 5: Highlight the *best next move* for MAX (from root, one move only)

MiniMax Heuristic for Tic-Tac-Toe. Using the heuristic shown below, compute the values of $e(n)$ for the three game states (MAX plays X):

$$e(n) = \begin{cases} \text{number of rows, columns, and diagonals open for MAX} \\ \quad \text{- number of rows, columns, and diagonals open for MIN} \\ +\infty, \text{ if n is a forced win for MAX} \\ -\infty, \text{ if n is a forced win for MIN} \end{cases}$$

Two-ply MiniMax. Compute the missing values using MiniMax in the game tree shown below (same heuristic as above, start node is MAX). What will be MAX’s next move?

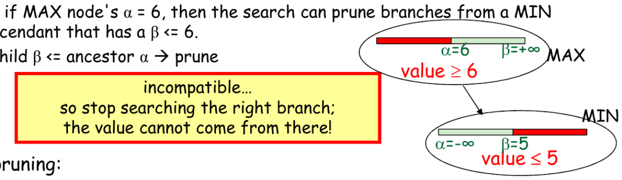


Alpha-Beta Pruning. Apply the Alpha-Beta Pruning algorithm:

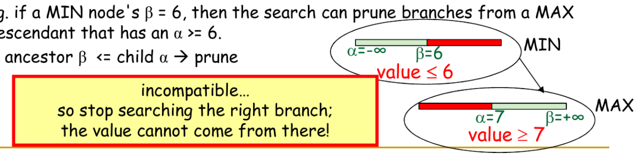
```
function alphabeta(node, depth, α, β, maximizingPlayer)
     if depth = 0 or node is a terminal node
         return the heuristic value of node
     if maximizingPlayer
         v := -∞
         for each child of node
             v := max(v, alphabeta(child, depth - 1, α, β, FALSE))
             α := max(α, v)
             if β ≤ α
                 break (* β cut-off *)
         return v
     else
         v := ∞
         for each child of node
             v := min(v, alphabeta(child, depth - 1, α, β, TRUE))
             β := min(β, v)
             if β ≤ α
                 break (* α cut-off *)
         return v
```

Initial call:
`alphabeta(origin, depth, -∞, +∞, TRUE)`

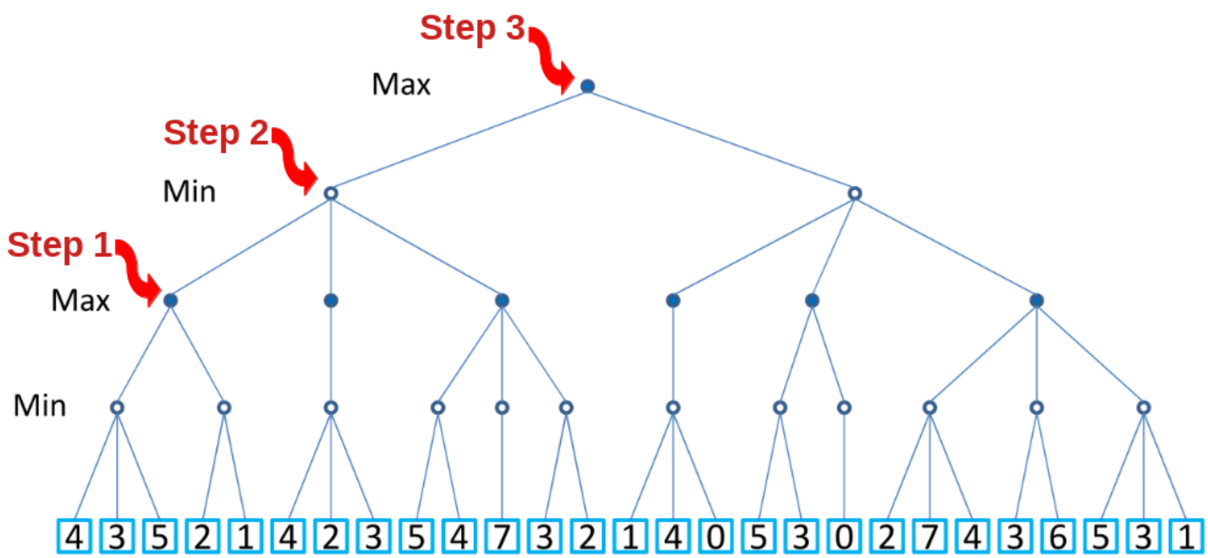
- α : lower bound on the final backed-up value.
- β : upper bound on the final backed-up value.
- Alpha pruning:
  - eg. if MAX node's α = 6, then the search can prune branches from a MIN descendant that has a β <= 6.
  - if child β <= ancestor α → prune



- Beta pruning:
  - eg. if a MIN node's β = 6, then the search can prune branches from a MAX descendant that has an α >= 6.
  - if ancestor β <= child α → prune



on the following search tree:



We will compare and discuss the results after each of the three steps:

Step 1: Perform the Alpha-Beta procedure (left-to-right) until you reached the node marked with "Step 1".

- Call `alphabeta( root, 4,` $-\infty$, $+\infty$, `TRUE)`
- Circle each node that you explored and show which subtrees are cut off by the algorithm (if any).

Step 2: Now continue with the algorithm until you reached the node marked "Step 2", marking explored nodes and cut subtrees as before.

Step 3: Complete the algorithm until you calculated the value for the root node in the same fashion.

How many nodes did the algorithm explore (out of 27 possible): .... ?