Artificial Intelligence

Lab 6

Gaming Algorithms

Agenda

Introduction to Gaming Algorithms

Games vs. Search problems

Gaming Algorithms

- Minimax
- Alpha-Beta
- Tic Tac Toe Hands on

Gaming Algorithms

- Games are a form of multi-agent deterministic environment (2 players).
- What do other agents do and how do they affect our success?
- Cooperative vs. Competitive multi-agent environments.
- Competitive multi-agent environments give rise to gaming search.

Games vs. Search Problems

- Why can not we use traditional search algorithms like BFS, DFS, UCS, A* ?
- Game problems includes two player, both players try to win the game, so, both of them try to make the best move possible at each turn.
- Searching algorithms like BFS, DFS, UCS or A* are not accurate for this.
- So, we need another search procedures that improve to:
  - ☐ **Generate procedure:** It generates only good moves that can be taken from current state.
  - ☐ **Test procedure:** that choose the best move to be explored first.

Minimax Algorithm

- Minimax is a kind of backtracking algorithm that is used in game theory to find the optimal move for a player, assuming that your opponent also plays optimally.

- It is widely used in two player turn-based games such as Tic-Tac-Toe, Chess, etc.

- In Minimax the two players are called maximizer and minimizer.
  - The maximizer tries to get the highest score possible.
  - The minimizer tries to do the opposite and get the lowest score possible.

Mini-Max Terminology

A game can be defined a search problem with the following components:

- **Initial state:** It comprises the position of the board and showing whose move it is.
- **Successor function:** It defines what the legal moves a player can make are.
- **Terminal state:** It is the position of the board when the game gets over.
- **Utility function:** It is a function which assigns a numeric value for the outcome of a game.

For instance, in chess or tic-tac-toe, the outcome is either a win, a loss, or a draw, and these can be represented by the values +1, -1, or 0, respectively.

Game Tree (2-player, De

The computer is **Max (X)**.
The opponent is **Min (O)**.

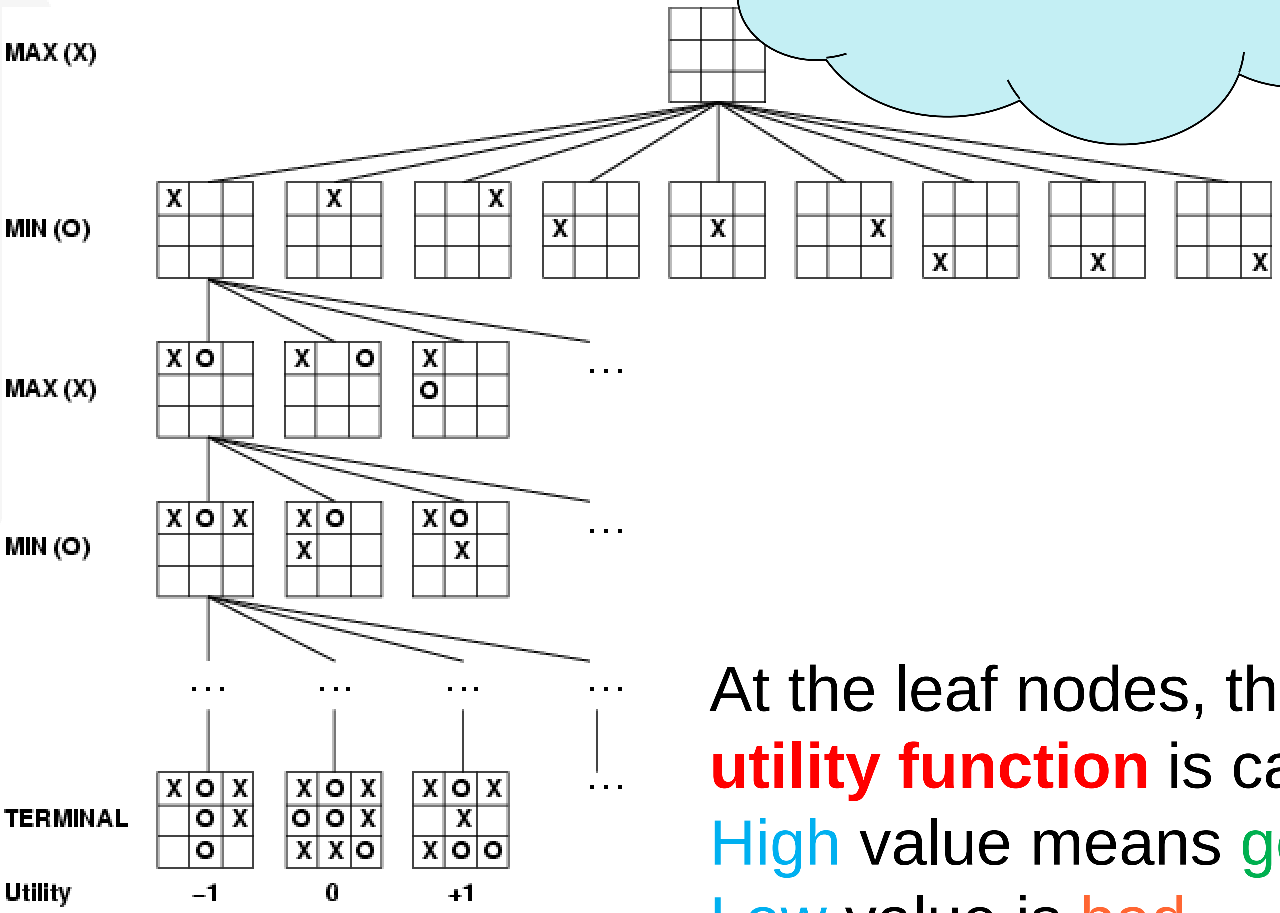
computer's turn

opponent's turn

computer's turn

opponent's turn

leaf nodes are evaluated



At the leaf nodes, the **utility function** is called. High value means good, Low value is bad.

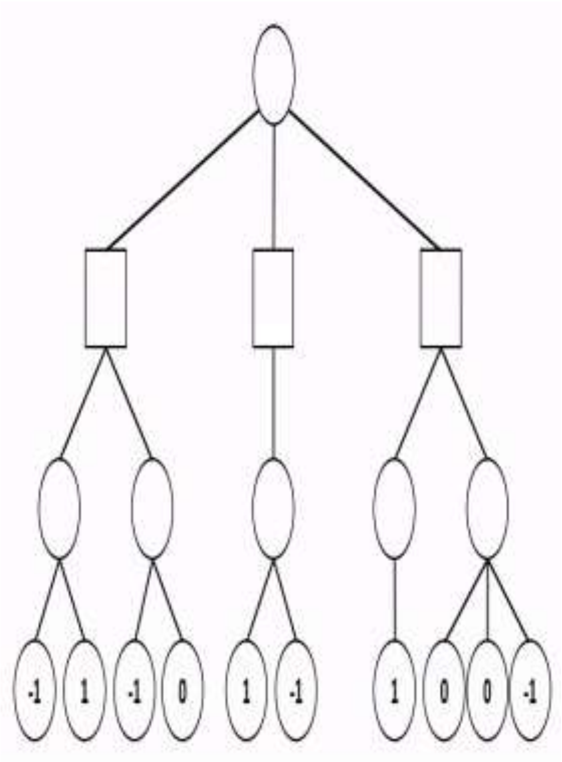
How does the algorithm work?

Step 1: First, generate the entire game tree starting with the current position of the game all the way up to the terminal states.

Step 2: Apply the utility function to get the utility values for all the terminal states.

Step 3: Determine the utilities of the higher nodes with the help of the utilities of the terminal nodes.

- From bottom to top
- For a max level, select the maximum value of its successors
- For a min level, select the minimum value of its successors

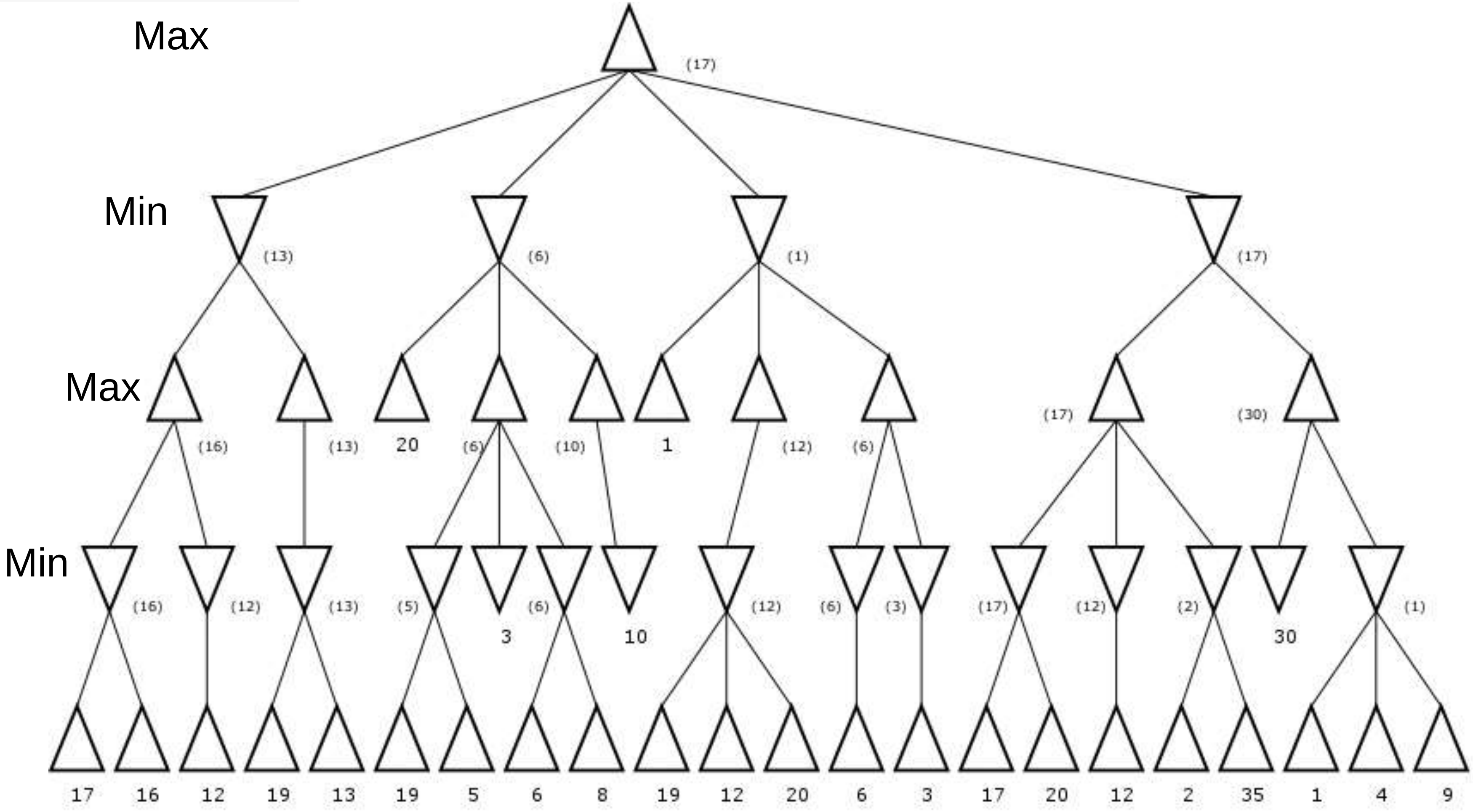


Step 4: From root node select the move which leads to highest value

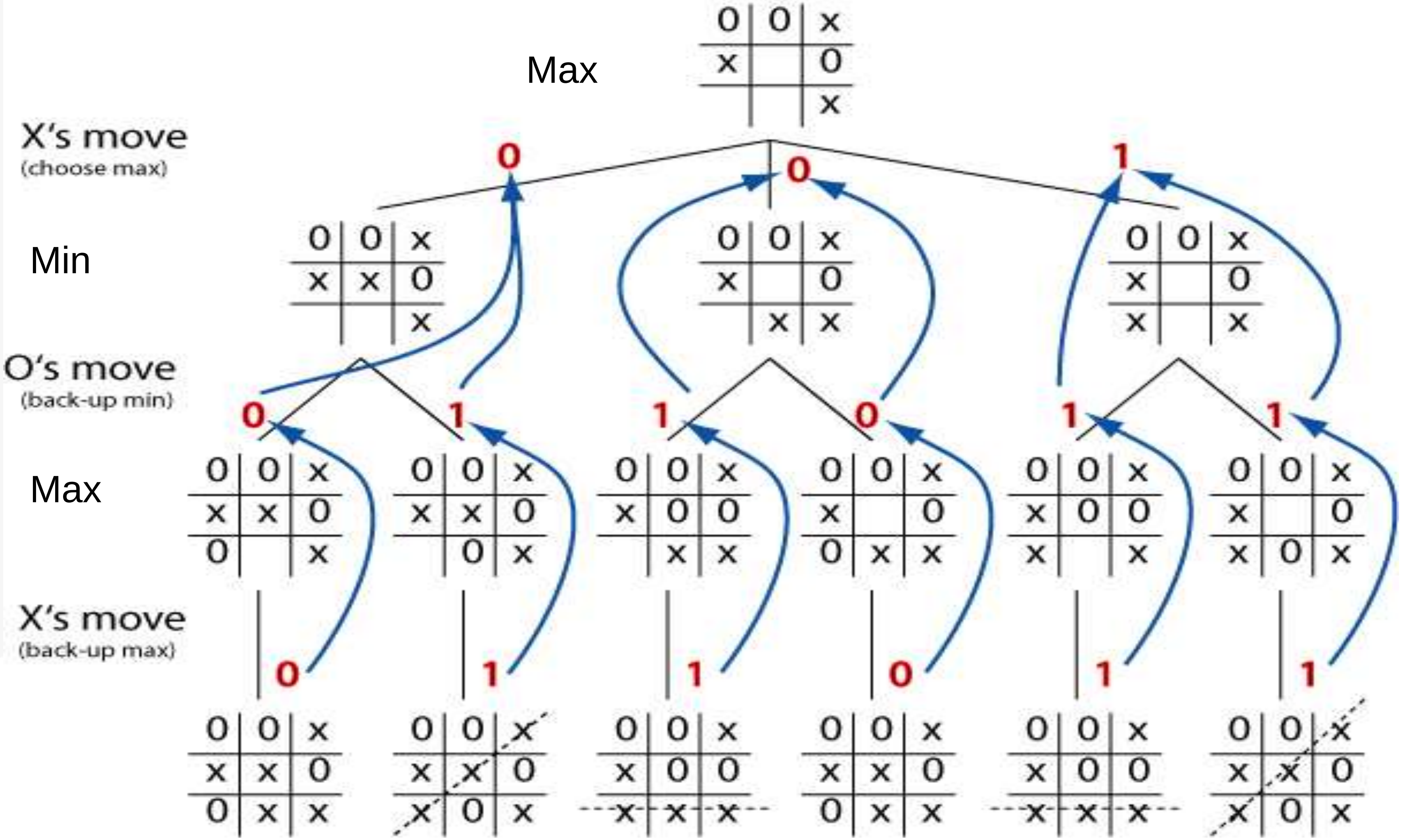
Utility Evaluation Function

- Utility Functions are very game-specific
- The simplest utility function can be evaluated as Sum Zero:
  - 1 if player X wins
  - -1 if player O wins
  - 0 if tie

Example


Max
(17)
Min
(13)
(6)
(1)
(17)
Max
(16)
(13)
20
(6)
(10)
1
(12)
(6)
(17)
(30)
Min
(16)
(12)
(13)
(5)
(6)
3
10
(12)
(6)
(3)
(17)
(12)
(2)
30
(1)
17
16
12
19
13
19
5
6
8
19
12
20
6
3
17
20
12
2
35
1
4
9

Another Example



Minimax Algorithm

```
function minimax(node, depth, maximizingPlayer) is
 if node is a terminal node then
    return the utility value of node
 if maximizingPlayer then
       value := −∞
       for each child of node do
           value := max(value, minimax(child, depth − 1, FALSE))
       return value
 else (* minimizing player *)
     value := +∞
     for each child of node do
       value := min(value, minimax(child, depth − 1, TRUE))
     return value
```

Making our Minimax smarter :

Assume that there are 2 possible ways for X to win the game from a give board state.

- Move **A** : X can win in 2 move
- Move **B** : X can win in 4 moves

Our evaluation function will return a value of +10 for both moves **A** and **B**. Even though the move **A** is better because it ensures a faster victory, our AI may choose **B** sometimes.

To overcome this problem we subtract the depth value from the evaluated score.

This means that in case of a victory it will choose a the victory which takes least number of moves.

Making our Minimax smarter :

So the new evaluated value will be:

- Move **A** will have a value of +10 – 2 = 8
- Move **B** will have a value of +10 – 4 = 6

Now since move **A** has a higher score compared to move **B** our AI will choose move **A** over move **B**.

The same thing must be applied to the minimizer. Instead of subtracting the depth we add the depth value.

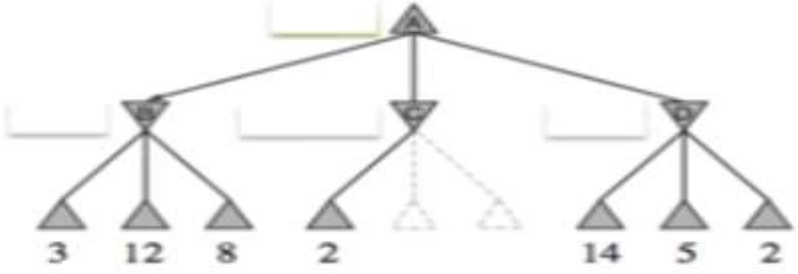
Properties of Minimax

- Minimax algorithm requires expanding the entire tree.

- How deeply should the tree be searched? Each increase in depth multiplies the total search time by about the number of moves available at each level.

Alpha-Beta Pruning

The full minimax search explores some parts of the tree it doesn't have to. For example, Do we need to calculate Z value ?.



Do we need to expand all nodes?

$$\begin{aligned} minimax(root) &= max(min(3,12,8), min(2,x,y), min(14,5,2)) \\ &= max(3, min(2,x,y), 2) \\ &= max(3, z, 2) \\ &= 3 \end{aligned}$$

Do we need z?

Alpha-Beta Strategy

Instead of calculating value of utility Only.

Calculate Two Extra Values:

- Alpha (α): a value of the best choice so far for Max (Highest value)
- Beta (β): a value of the best choice so far for Min (lowest value)

Search, maintaining α and β Whenever

$\alpha \geq \beta_{higher}$, or $\beta \leq \alpha_{higher}$ further search at this node is irrelevant

How to Prune the Unnecessary Path

- **If** beta value of any MIN node below a MAX node is less than or equal to its alpha value, **then** prune the path below the MIN node.

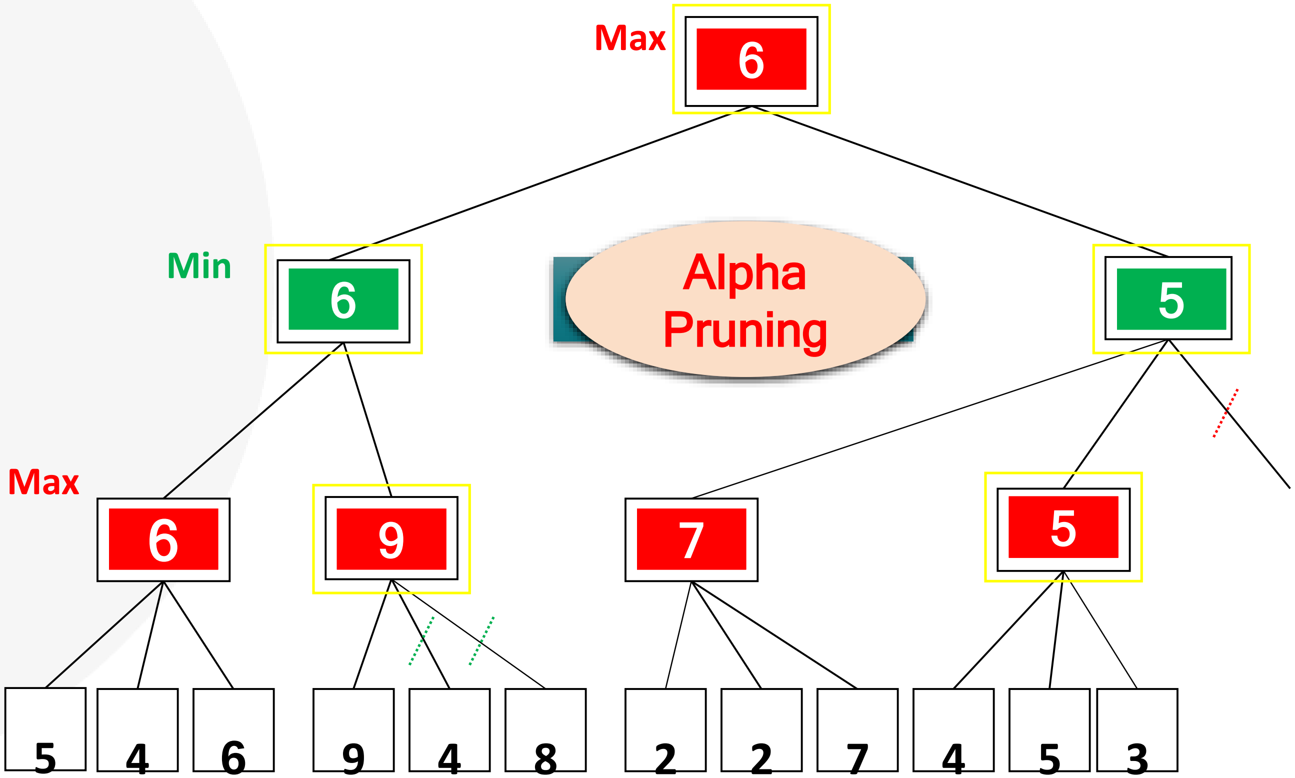
- **If** alpha value of any MAX node below a MIN node exceeds the beta value of the MIN node, **then** prune the nodes below the MAX node.

Example

Alpha

Beta

Alpha



Alpha Pruning

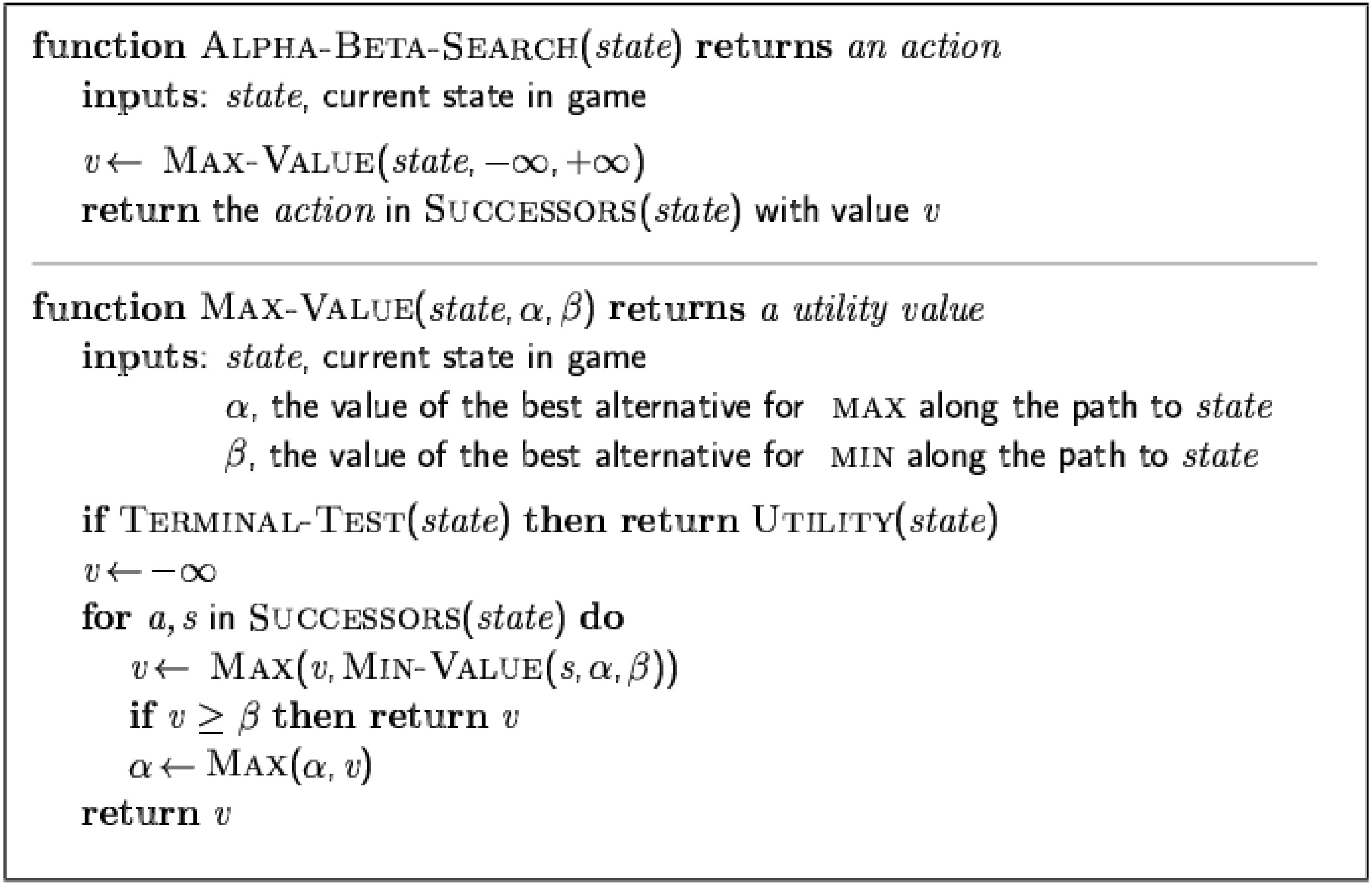
Max asks: 5 > 6

An answer: No

An action: Keep Alpha’s value

Node 9

The α-β algorithm



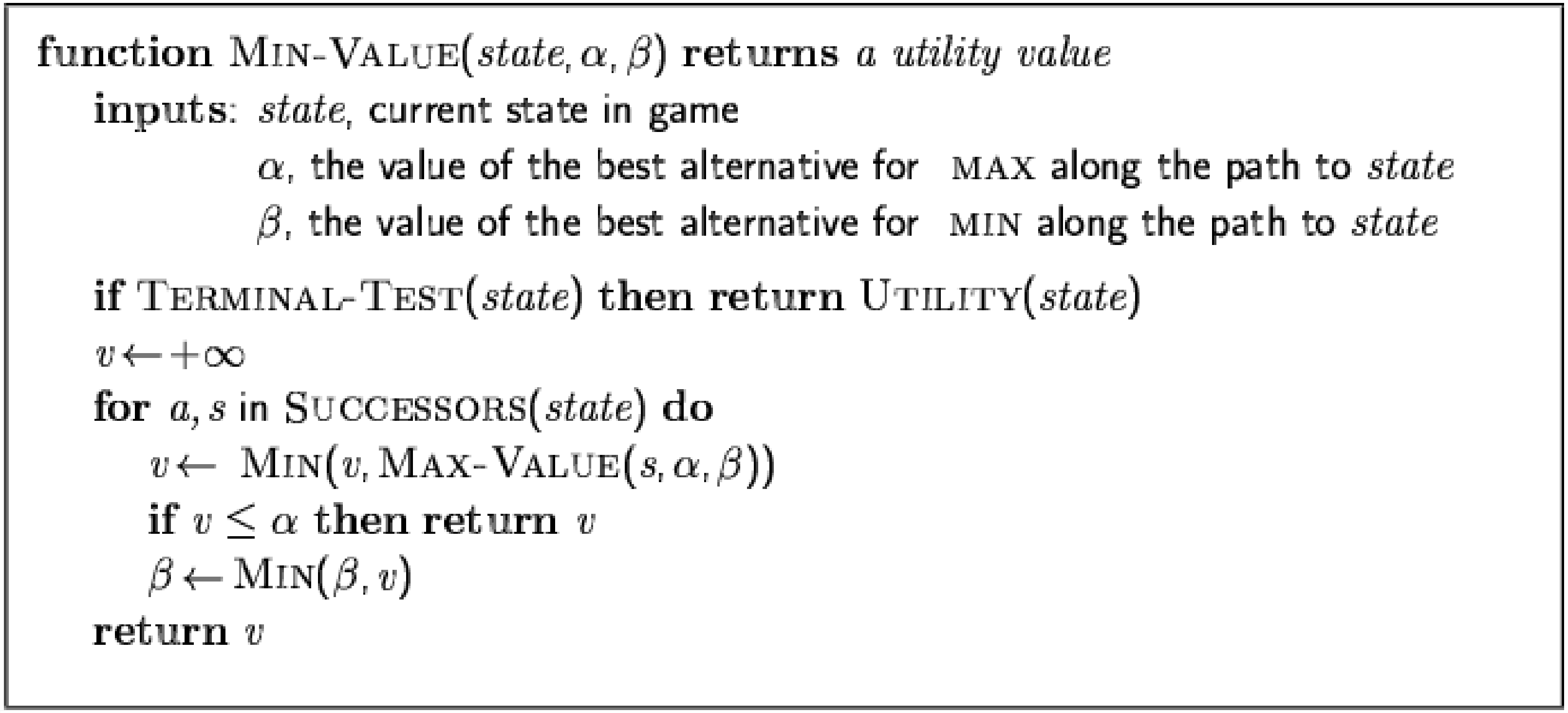
```
function ALPHA-BETA-SEARCH(state) returns an action
    inputs: state, current state in game

    $v \leftarrow$ MAX-VALUE(state, $-\infty$, $+\infty$)
    return the action in SUCCESSORS(state) with value $v$
```

```
function MAX-VALUE(state, $\alpha$, $\beta$) returns a utility value
    inputs: state, current state in game
            $\alpha$, the value of the best alternative for MAX along the path to state
            $\beta$, the value of the best alternative for MIN along the path to state

    if TERMINAL-TEST(state) then return UTILITY(state)
    $v \leftarrow -\infty$
    for $a, s$ in SUCCESSORS(state) do
        $v \leftarrow$ MAX($v$, MIN-VALUE($s, \alpha, \beta$))
        if $v \geq \beta$ then return $v$
        $\alpha \leftarrow$ MAX($\alpha$, $v$)
    return $v$
```

The α-β algorithm

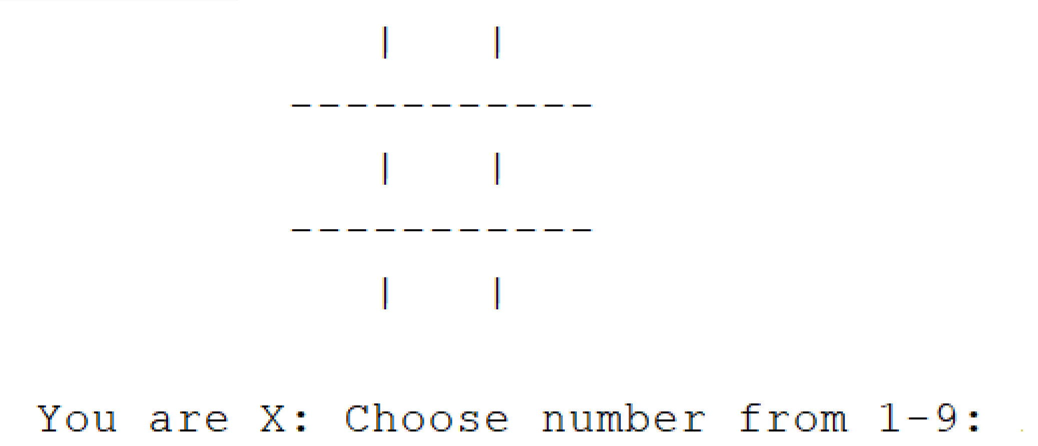


```
function MIN-VALUE($state, \alpha, \beta$) returns a utility value
    inputs: $state$, current state in game
            $\alpha$, the value of the best alternative for MAX along the path to $state$
            $\beta$, the value of the best alternative for MIN along the path to $state$

    if TERMINAL-TEST($state$) then return UTILITY($state$)
    $v \leftarrow +\infty$
    for $a, s$ in SUCCESSORS($state$) do
        $v \leftarrow$ MIN($v$, MAX-VALUE($s, \alpha, \beta$))
        if $v \leq \alpha$ then return $v$
        $\beta \leftarrow$ MIN($\beta, v$)
    return $v$
```

Hands on – Tic Tac Toe

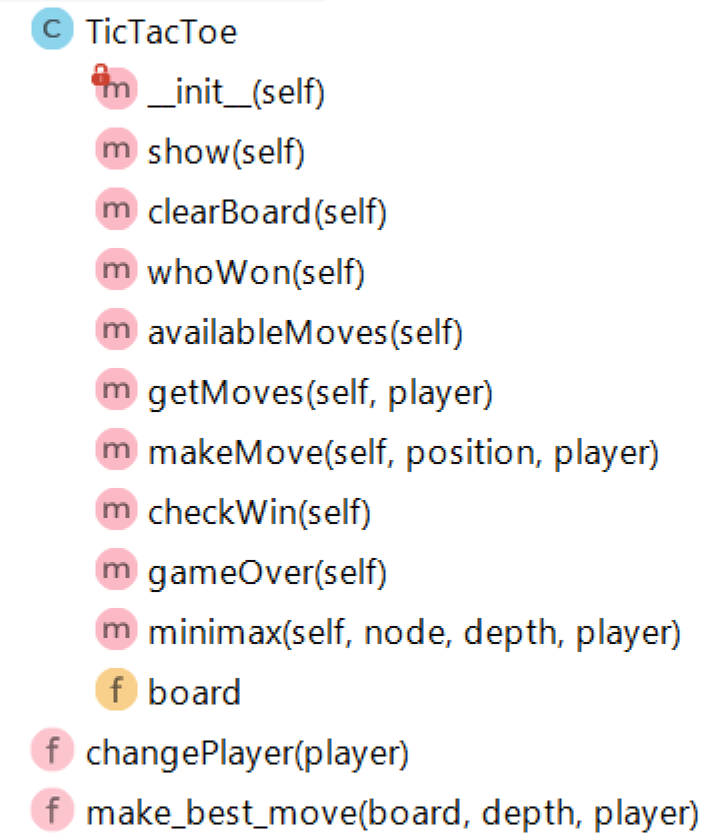
- Human is ‘X’ and Machine ‘O’.
- Board is 1 based index.
- X is maximizer and O is minimizer.



```
             |   |
          ------------
             |   |
          ------------
             |   |

You are X: Choose number from 1-9:
```

Hands on – Tic Tac Toe



- c TicTacToe
  - m __init__(self)
  - m show(self)
  - m clearBoard(self)
  - m whoWon(self)
  - m availableMoves(self)
  - m getMoves(self, player)
  - m makeMove(self, position, player)
  - m checkWin(self)
  - m gameOver(self)
  - m minimax(self, node, depth, player)
  - f board
- f changePlayer(player)
- f make_best_move(board, depth, player)

Minimax Algorithm

```
function minimax(node, depth, maximizingPlayer) is
 if node is a terminal node
       then return the utility value of node
 if maximizingPlayer
      then value := −∞
       for each child of node do
           value := max(value, minimax(child, depth − 1, FALSE))
       return value
 else (* minimizing player *)
     value := +∞
     for each child of node do
       value := min(value, minimax(child, depth − 1, TRUE))
     return value
```

Milestone 3

Gaming Algorithms milestone deadline: 12 April 2019.

It will be published on course-sites : 4 April 2019.

General instructions:

- Regarding your AI-Package:
- Add a new folder named ‘GamingAlgorithms’.
- Add only one new ‘.PY’ file for writing your code.

Regarding your submission file:

- Submit **only running** code that you have tested before.
- Your assignment should be written in **ONE** “.py” file, this file should include the solution of **ALL** the problems and a **main** function that calls them.
- Compressed files (.zip/.rar) are **not allowed.**
- The Submission of team work package is only through **your shared folder on google drive.**
- **Don’t delete** any previous milestones.

Questions?