IART - Artificial Intelligence

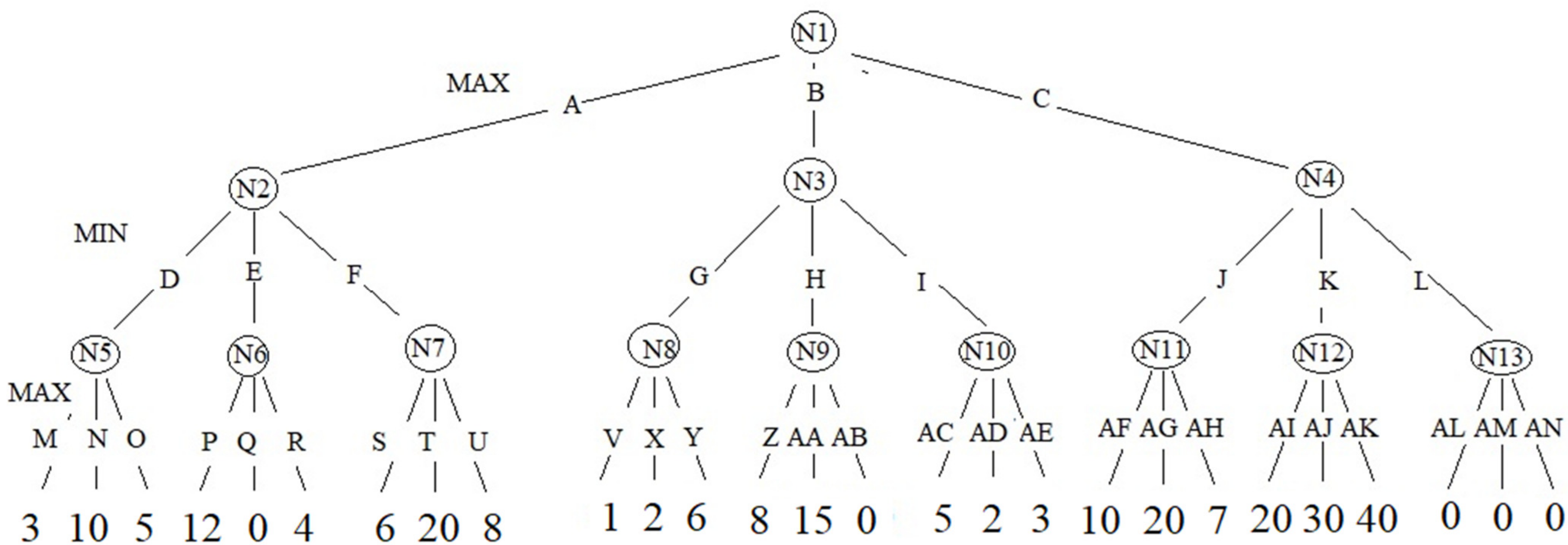
Exercise 3: Adversarial Search Problems

Luís Paulo Reis

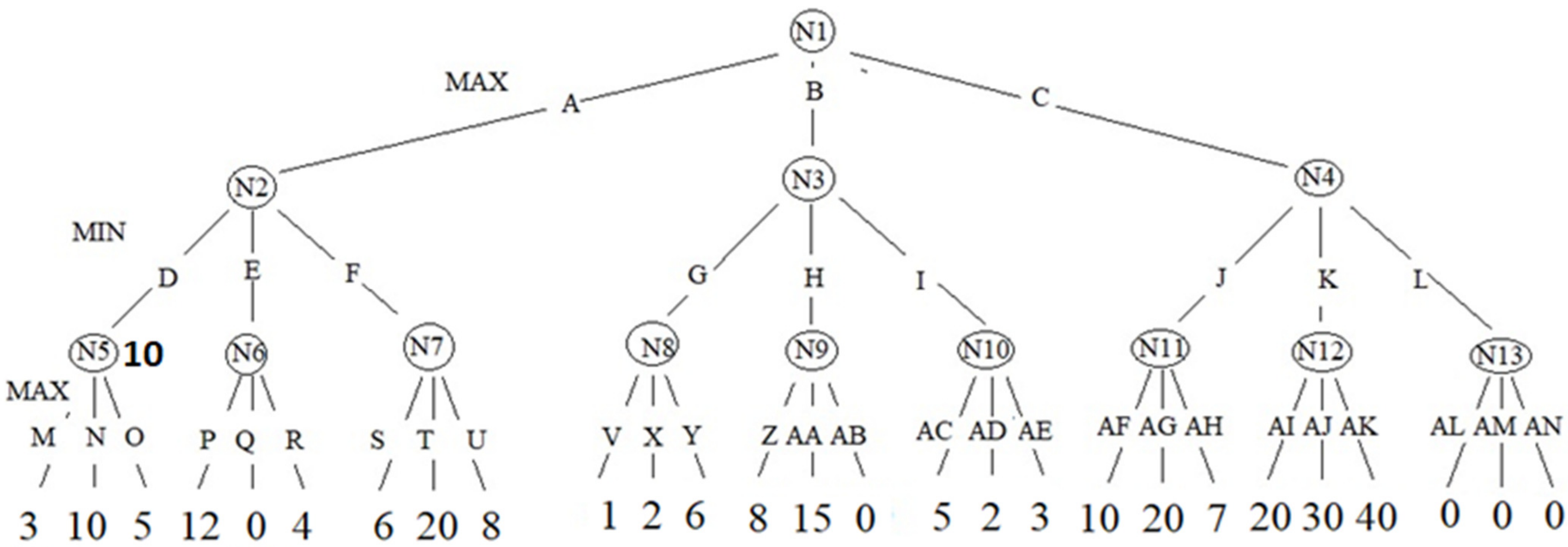
LIACC – Artificial Intelligence and Computer Science Lab.
DEI/FEUP – Informatics Engineering Department, Faculty of Engineering of the University of Porto, Portugal
APPIA – Portuguese Association for Artificial Intelligence

Exercise 3.1: Minimax with Alpha-Beta Cuts

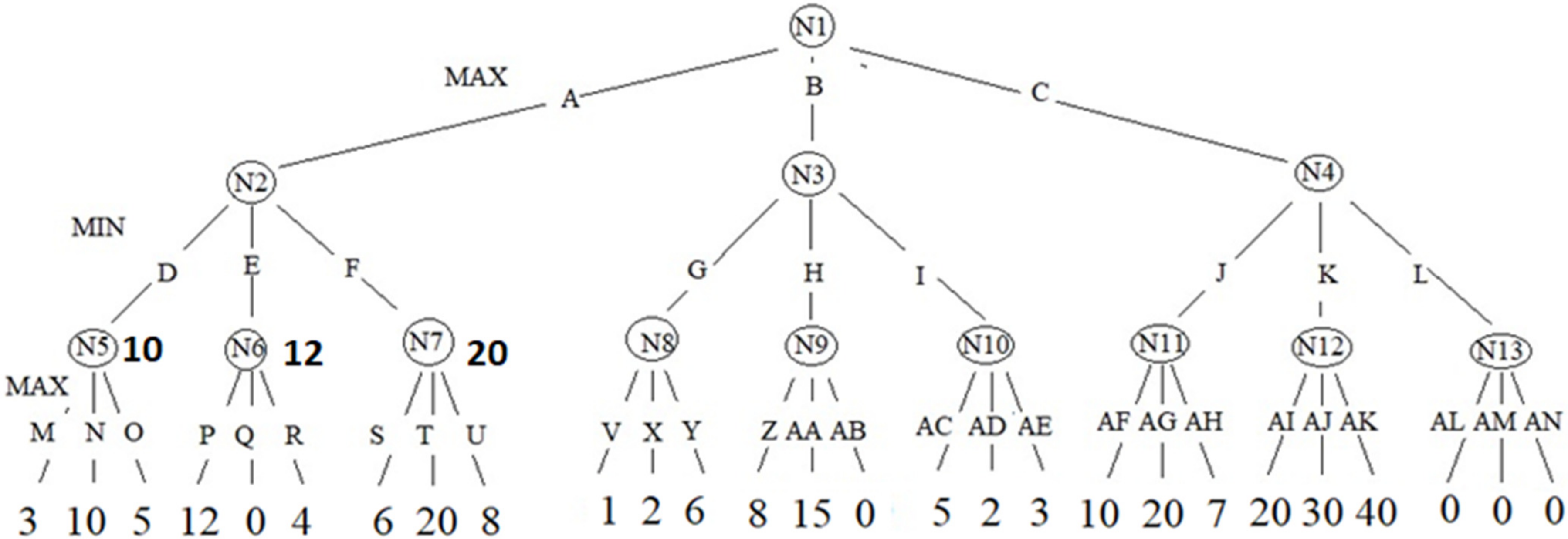
Apply the Minimax algorithm with alpha-beta cuts to the following tree that has a branch factor of 3 at the top level, 3 at the second level, and also 3 at the final level, and with the evaluation function values indicated for the final line. Indicate the final value of each node and which branches are cut by the Alpha-Beta cuts.



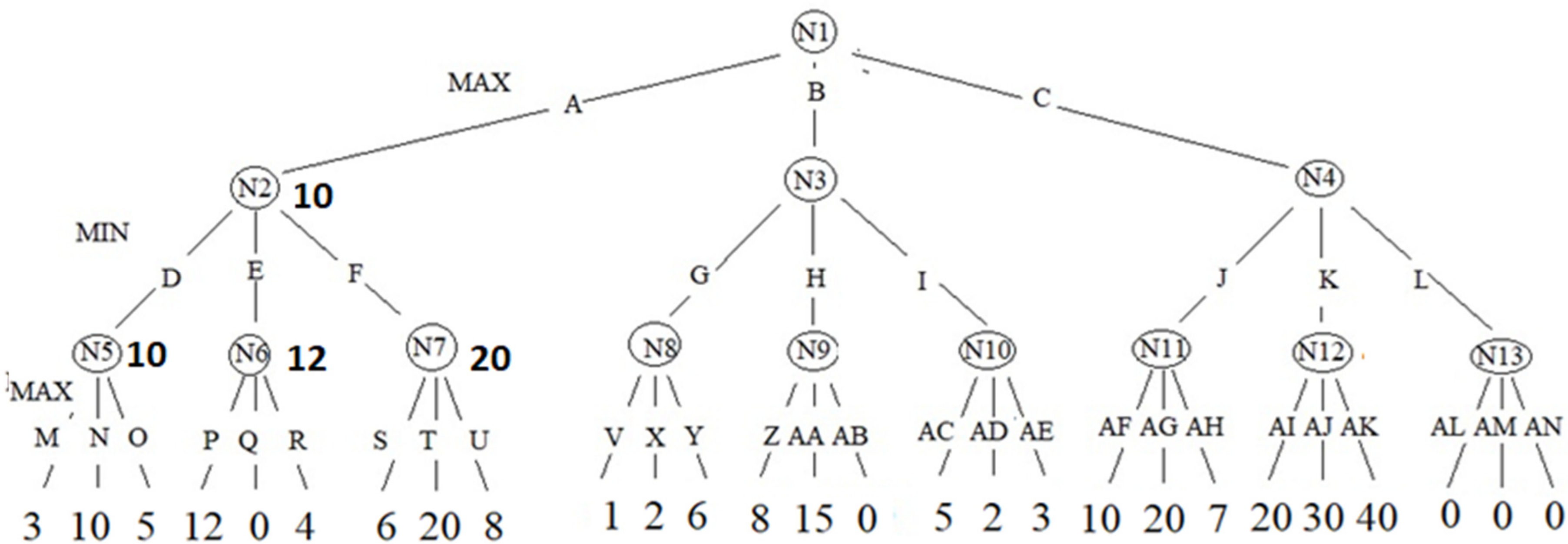
Exercise 3.1: Minimax



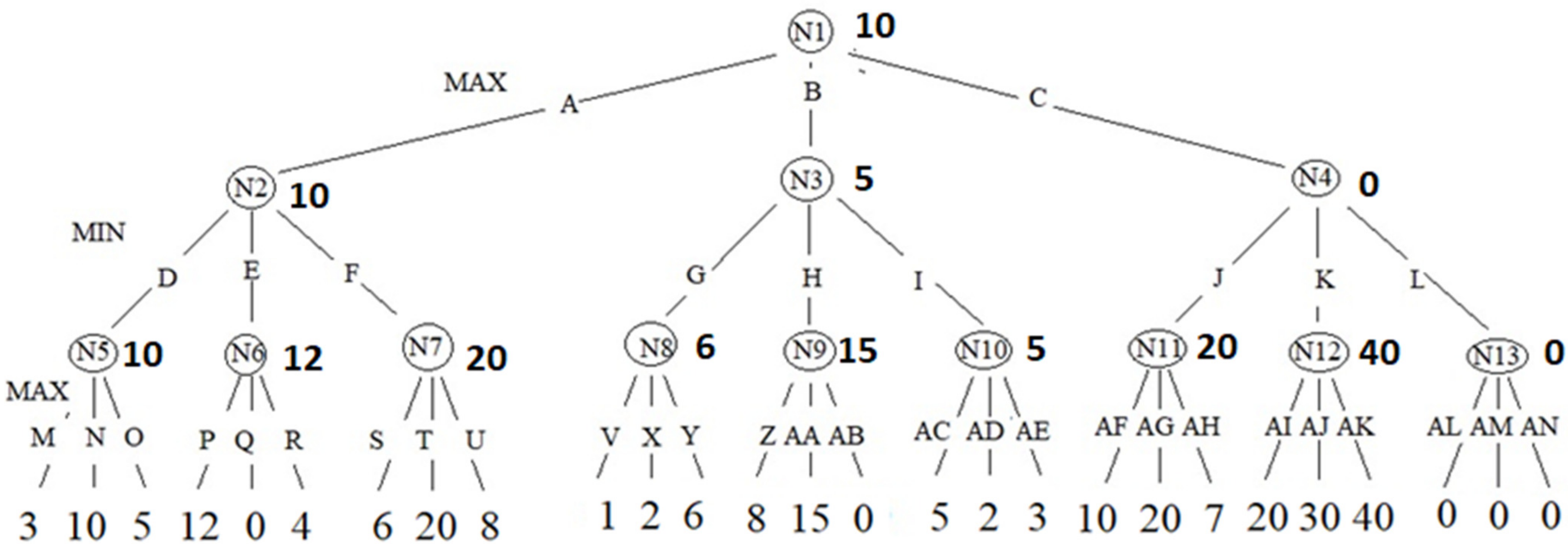
Exercise 3.1: Minimax



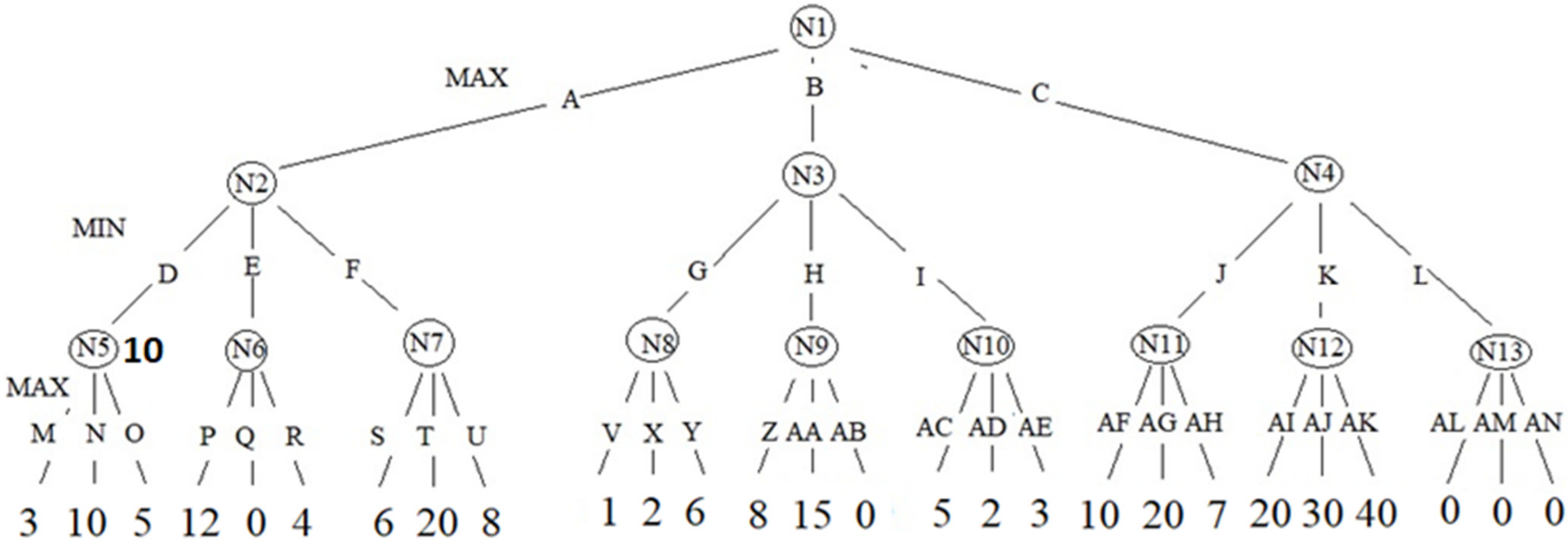
Exercise 3.1: Minimax



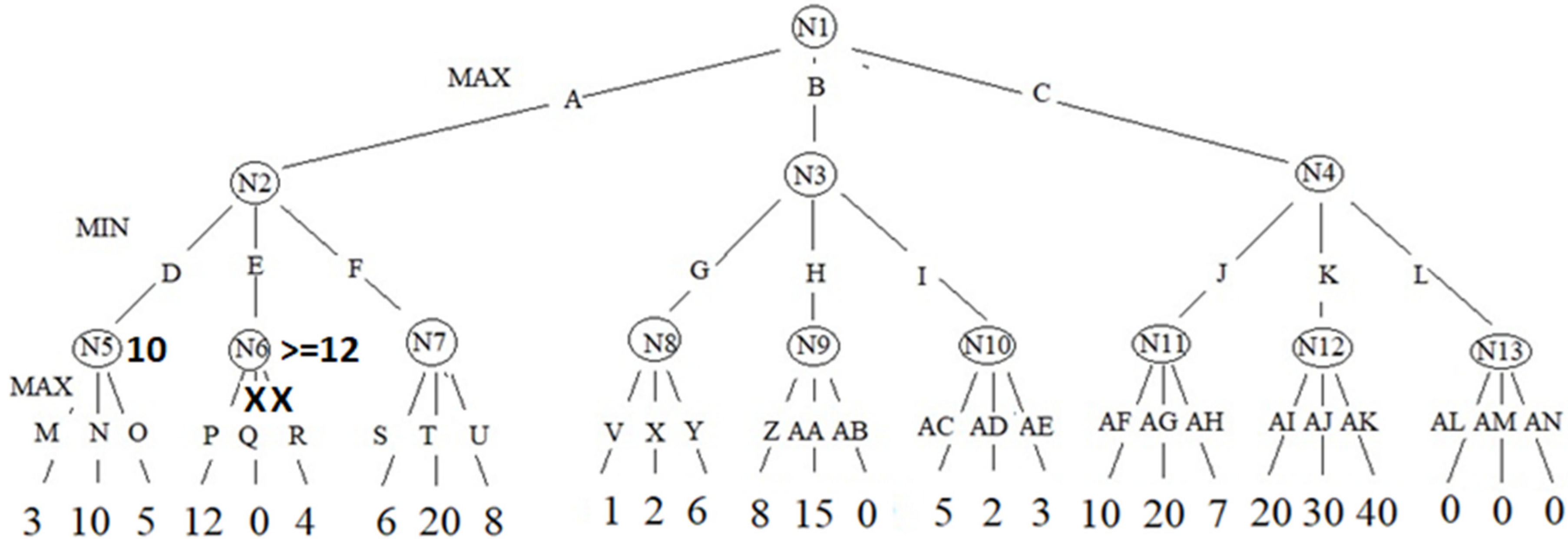
Exercise 3.1: Minimax



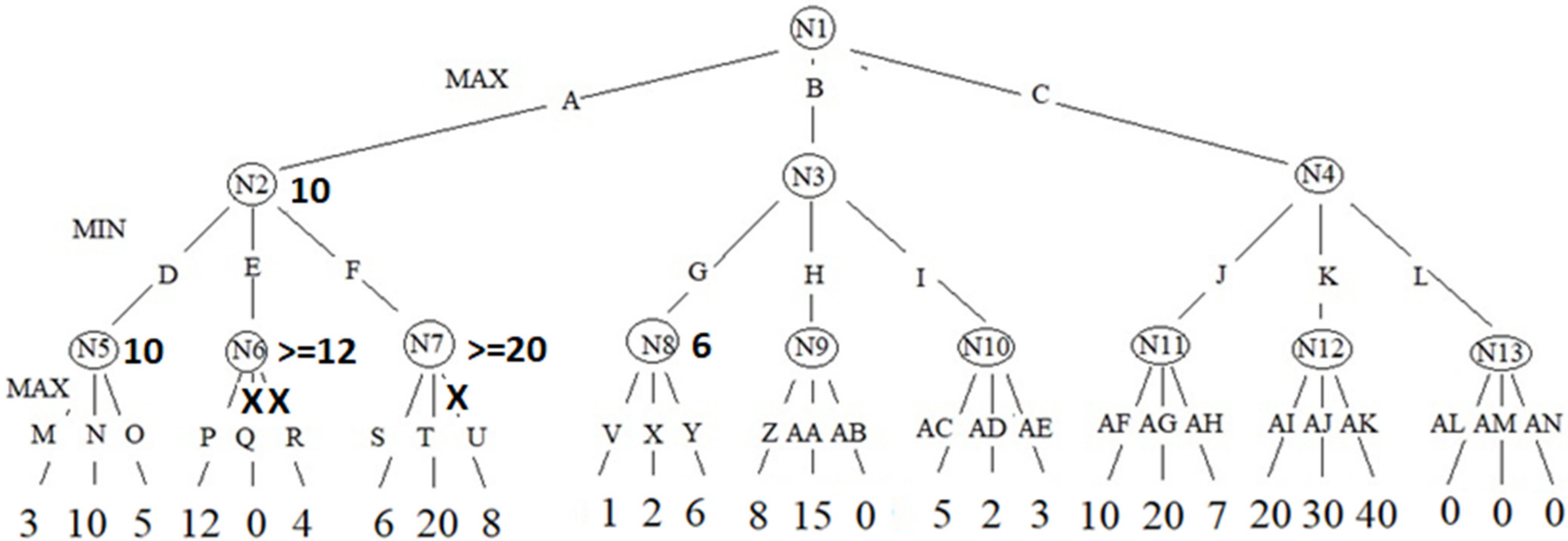
Exercise 3.1: Minimax with Alpha-Beta Cuts



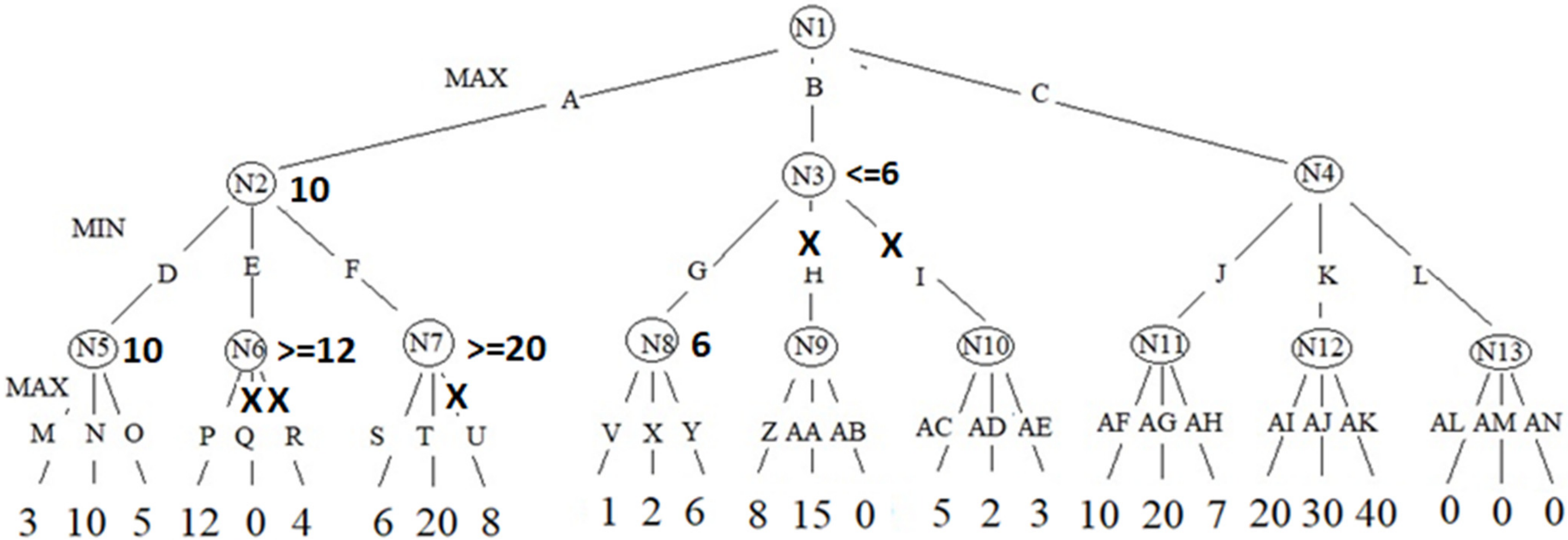
Exercise 3.1: Minimax with Alpha-Beta Cuts



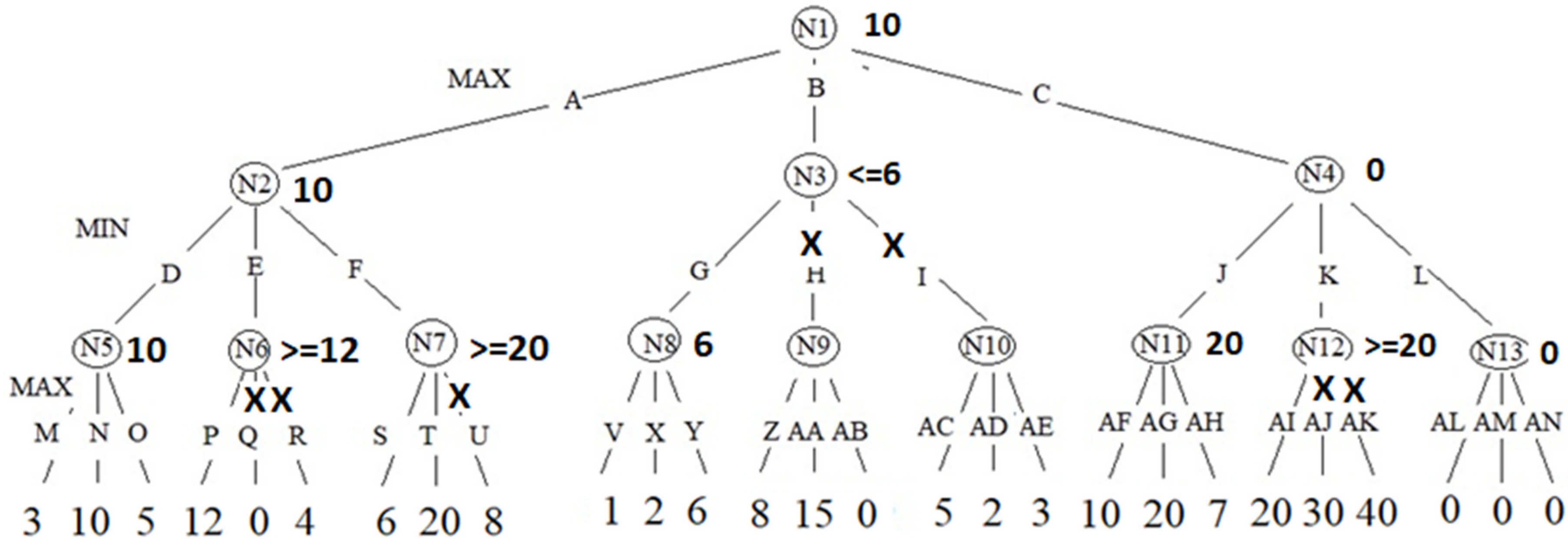
Exercise 3.1: Minimax with Alpha-Beta Cuts



Exercise 3.1: Minimax with Alpha-Beta Cuts

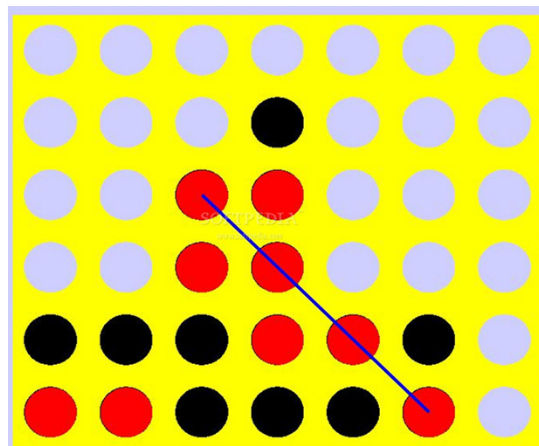


Exercise 3.1: Minimax with Alpha-Beta Cuts



Exercise 3.2: Connect-4 Game

The game called "Connect Four" in the English language version (“4 em Linha” in the Portuguese version - https://en.wikipedia.org/wiki/Connect_Four) is played on a vertical board of 7x6 squares (i.e., 7 squares wide and 6 squares high), by two players, to which are initially assigned 21 pieces to each.



The two players play alternately one of their pieces. The piece to be played is placed on the top of the board and slides either to the base of the board, or in a cell immediately above another one already occupied (see previous figure). The winner will be the player who manages to obtain a line of 4 pieces of its color/symbol horizontally, vertically, or diagonally. If the 42 pieces are played without any player getting a line, the final result will be a draw.

Connect-4 Game Implementation

a) Formulate this game as a search problem with opponents, indicating the state representation, moves/operators (and respective names, preconditions, and effects), and the objective test.

b) Implement a simple version of the “Connect-Four” game in a programming language of your choice.

c) Implement the following functions:

> c1) *int nlines4 (int Player)* that given the state of the board calculates the number of lines with 4 pieces (horizontal, vertical, diagonal) of a given player.
>
> c2) *int nlines3 (int Player)*, similar to the previous function, but which calculates the number of sets of 4 consecutive spots that have three pieces of the player followed by an empty spot, i.e., that are possibilities to win the game.
>
> c3) *int central (int Player)*, that assigns 2 points to each player piece in the center column of the board (column 4) and 1 point to each piece in the columns around it (columns 3 and 5).

Connect-4 Game Implementation

d) Implement an agent to play the game using the minimax algorithm with alpha-beta cuts.

e) Compare the results of the implemented agents, playing 10 matches of this game with each other, using the minimax algorithm with alpha-beta cuts, with levels (2, 4, 6 and 8), and the following evaluation functions:

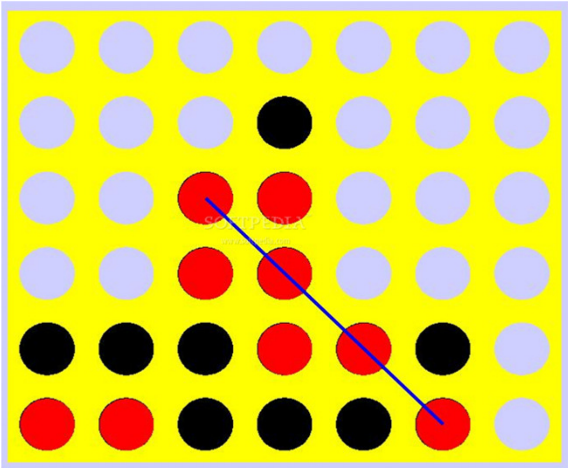
- *Agent1: EvalF1 = nlines4(1) - nlines4(2)*
- *Agent2: EvalF2 = 100* EvalF1 + nlines3(1) - nlines3(2)*
- *Agent3: EvalF3 = 100* EvalF1 + central(1) - central(2)*
- *Agent4: EvalF4 = 5* EvalF2 + EvalF3*

f) Conclude about the effectiveness of each of the evaluation functions/agents and the effect of the depth used in the Minimax Algorithm.

g) How could you improve the evaluation function for this type of agent?

Solving the Connect-4 Game

- **State Representation:**
- **Initial State:**
- **Objective State:**
- **Operators:**



Solving the Connect-4 Game

- **State Representation:**

  Matrix with Board: *B[6,7], or* in the general case *B[N,M],* filled with values 0..2 *# 0 represents empty square, 1 and 2 pieces from player 1 or 2*

  Also the Player to move (*Pla*). Also it is a good idea to add to the last square played *(Yl, Xl)* for efficiency

- **Initial State:**

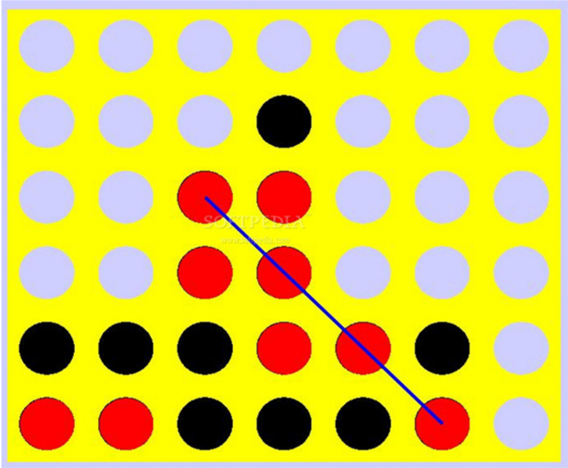
  ???

- **Objective State:**

  ???

- **Operators:**

  ???

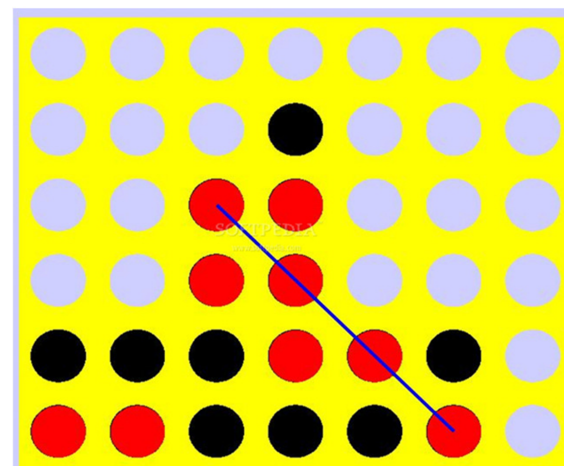


Solving the Connect-4 Game

- **State Representation:**

  Matrix with Board: *B[6,7], or* in the general case *B[N,M],* filled with values 0..2 *# 0 represents empty square, 1 and 2 pieces from player 1 or 2*

  Also the Player to move (*Pla*). Also it is a good idea to add to the last square played *(Yl, Xl)* for efficiency



- **Initial State:**

  B[6,7]={0} //Matriz B all with zeros (0)

  Pla = 1

- **Objective State:**

  ???

- **Operators:**

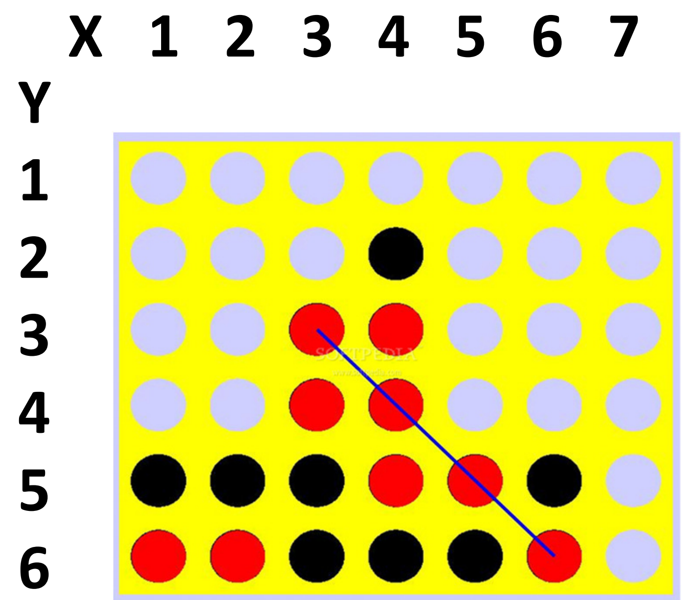
  ???

Solving the Connect-4 Game

- **State Representation:**

  Matrix with Board: *B[6,7], or* in the general case *B[N,M],* filled with values 0..2
  # 0 represents empty square, 1 and 2 pieces from player 1 or 2

  Player to move: *Pla*.
  Also it is a good idea to add the last square played *(Yl, Xl)* for efficiency



- **Initial State:**

  B[6,7]={0} # Matriz B filled with zeros (0)

  Pla = 1 # Player 1 is the first to move

- **Objective State 0> Objective Test**

  // returns 0- draw, 1-Win for player 1, 2-Win for player 2, -1 – game not finished

```
def objective_test(B|Pla|Yl|Xl: State):
    … #Test lines in all directions from (Yl,Xl)
```

- **Operators:**

  ???

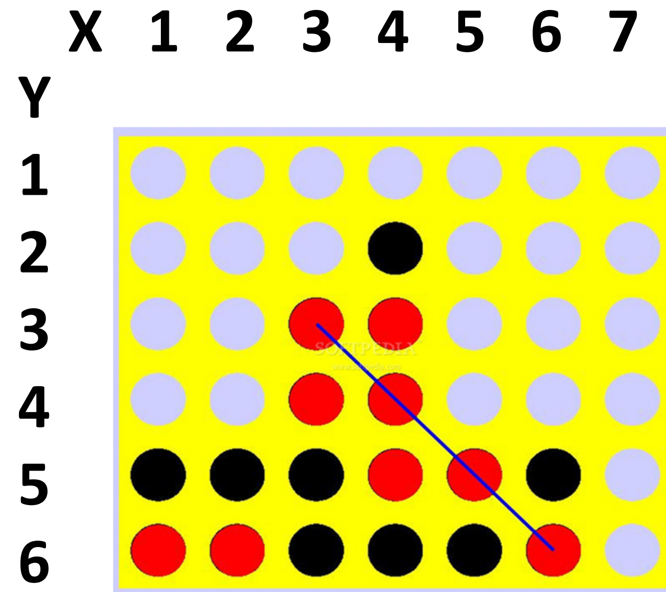
Solving the Connect-4 Game

Operators:

Name:

exec_move(Col: int)

PreConditions:



Solving the Connect-4 Game

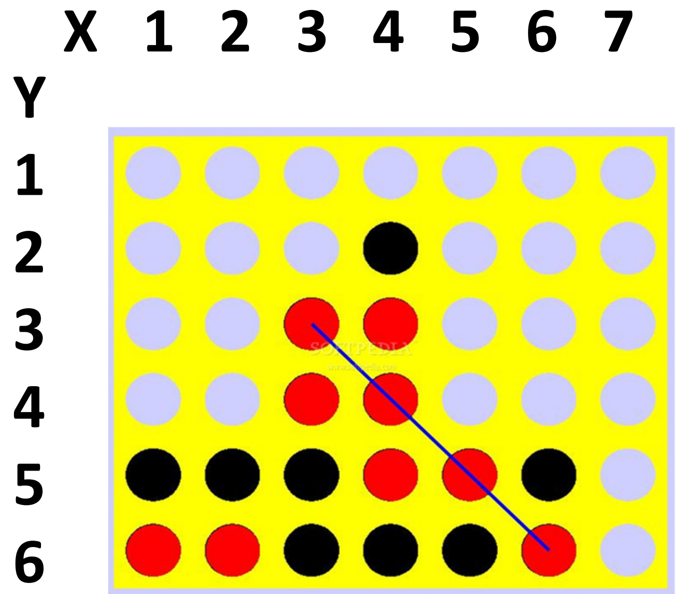
Operators:

Name:

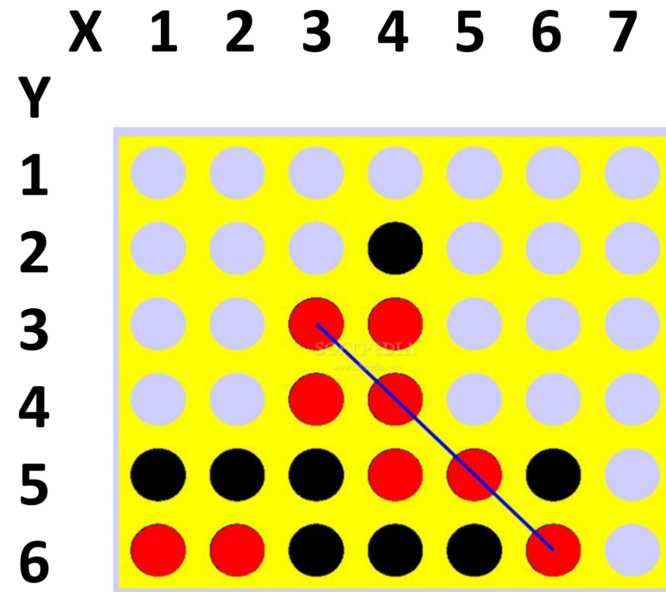
exec_move(Col: int)

PreConditions:

B[1, Col] == 0 *#for playing at Column Col*
#the top position of the
#Column must be empty



Solving the Connect-4 Game



Operators:

Name:

exec_move(Col: int)

PreConditions:

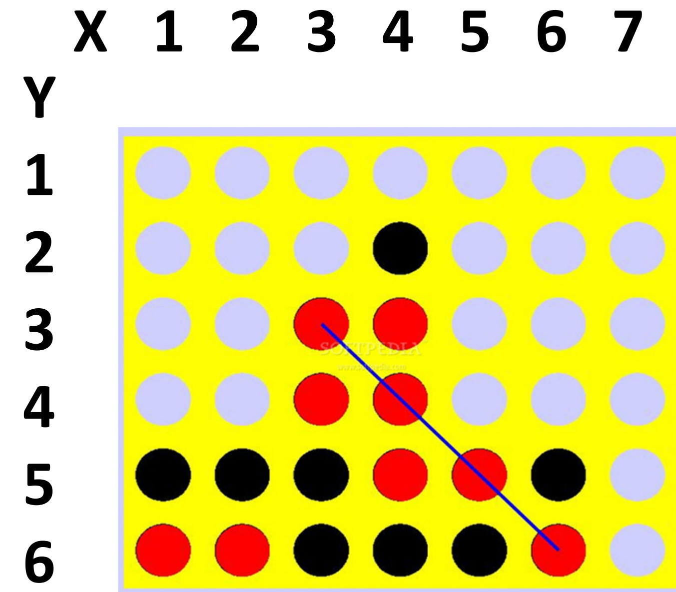
B[1, Col] == 0 *# for playing at Column Col*

 # the top position of the

 # Column must be empty

```
def valid_move(B|Pla|X1|Y1: State, Col: Operator):
   return B[1,Col]==0
```

Solving the Connect-4 Game



Operators:

Name:

exec_move(Col: int)

PreConditions:

B[1, Col] == 0 *# for playing at Column Col*

 # the top position of the

 # Column must be empty

```
def valid_move(B|Pla|X1|Y1: State, Col: Operator):
   return B[1,Col]==0
```

Effects:

```
def exec_move(B|Pla|X1|Y1: State, Col: Operator):
   …
```

Solving the Connect-4 Game

Operators:

Name:

exec_move(Col: int)

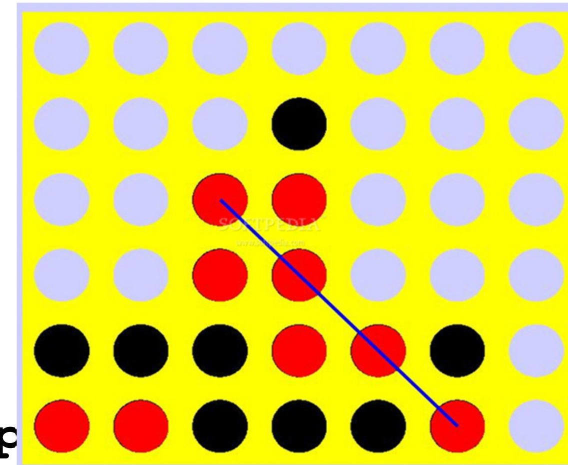
PreConditions:

B[1, Col] == 0 *# for playing at Column Col*

 # the top position of the

 # Column must be empty

```
def valid_move(B|Pla|X1|Y1: State, Col: Op
    return B[1,Col]==0
```



Effects:

```
def exec_move(B|Pla|X1|Y1: State, Col: Operator):
    i = 6
    while(B[i,Col]!=0)
      i-=1
    B[i,Col] = Pla
    Pla = 3-Pla
    Y1 = i
    X1 = Col
    return B|Pla|X1|Y1;
```

State Space

- **What is the State Space Size for the Connect4 Game:**
  - 6x7 Game?
  - Generic Case: NxM Game?
- **What is the Maximum Branching Factor for the connect 4 Game:**
  - 6x7 Game?
  - Generic Case: NxM Game?

State Space

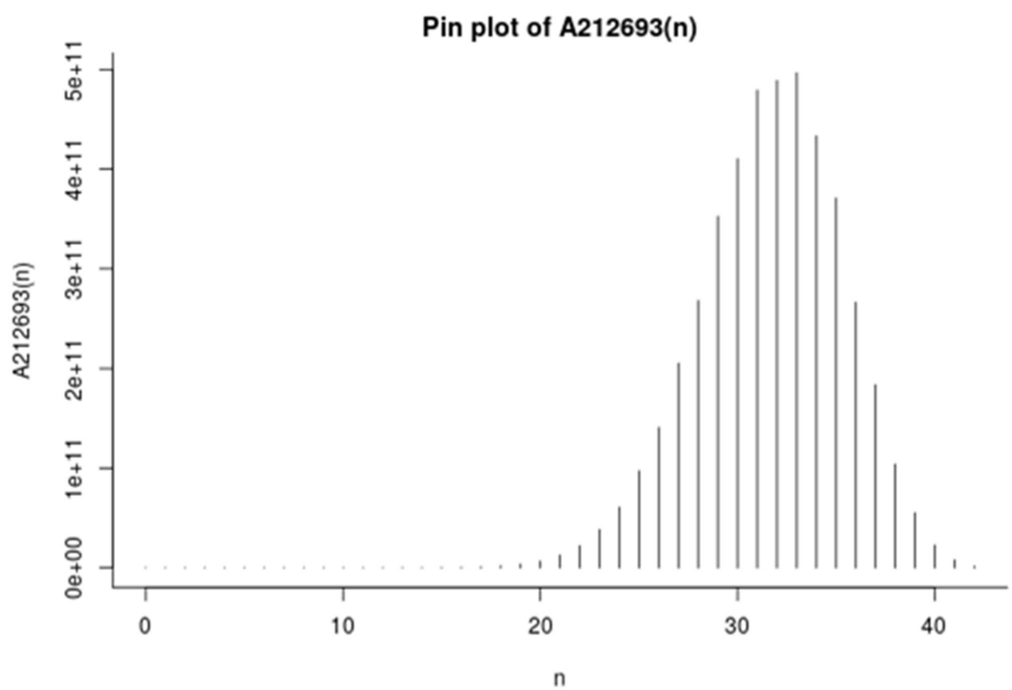
- **What is the State Space Size for the Connect4 Game:**
  - 6x7 Game?
    - $3^{(6x7)} = 3^{42} = 1.09 \times 10^{20}$?
    - Not completely since it includes a set of invalid states!
    - We need a better analysis!
- **What is the Maximum Branching Factor for the connect 4 Game:**
  - 6x7 Game?
    - Maximum Branching factor = 7 #7 columns
  - Generic Case: NxM Game?
    - Maximum Branching factor = M #Number of columns

State Space

- **What is the State Space Size for the Connect4 Game:**
  - 6x7 Game?
    - $3^{(6x7)} = 3^{42} = 1.09x10^{20}$?
    - Not really since it includes a set of invalid states:
      - States with floating pieces on top of empty spaces
      - States that may only be successors of states with already a line of 4 on the board (i.e. situations where the game was already ended)
      - States with difference between number of pieces 1 vs pieces 2 different from 0 and different from 1
- **What is the Maximum Branching Factor for the connect 4 Game:**
  - 6x7 Game?
    - Maximum Branching factor = 7 #7 columns
  - Generic Case: NxM Game?
    - Maximum Branching factor = M #Number of columns

State Space

- **What is the State Space Size for the Connect4 Game:**
  - 6x7 Game?
    - $3^{(6x7)} = 3^{42} = 1.09x10^{20}$?
    - Number of legal 6X7 Connect4 positions after n pieces:
      - 1 + 7 + 49 + 238 + 1120 + 4263 + 16422, … = 4531985219092 $= 4.53x10^{13}$



Connect-4 Game Implementation

a) Formulate this game as a search problem with opponents, indicating the state representation, moves/operators (and respective names, preconditions, and effects), and the objective test.

b) Implement a simple version of the "Connect-Four" game in a programming language of your choice.

c) Implement the following functions:

c1) *int nlines4 (int Player)* that given the state of the board calculates the number of lines with 4 pieces (horizontal, vertical, diagonal) of a given player.

c2) *int nlines3 (int Player)*, similar to the previous function, but which calculates the number of sets of 4 consecutive spots that have three pieces of the player followed by an empty spot, i.e., that are possibilities to win the game.

c3) *int central (int Player)*, that assigns 2 points to each player piece in the center column of the board (column 4) and 1 point to each piece in the columns around it (columns 3 and 5).

Connect-4 Game Implementation

d) Implement an agent to play the game using the minimax algorithm with alpha-beta cuts.

e) Compare the results of the implemented agents, playing 10 matches of this game with each other, using the minimax algorithm with alpha-beta cuts, with levels (2, 4, 6 and 8), and the following evaluation functions:

– *Agent1: EvalF1 = nlines4(1) - nlines4(2)*

– *Agent2: EvalF2 = 100* EvalF1 + nlines3(1) - nlines3(2)*

– *Agent3: EvalF3 = 100* EvalF1 + central(1) - central(2)*

– *Agent4: EvalF4 = 5* EvalF2 + EvalF3*

f) Conclude about the effectiveness of each of the evaluation functions/agents and the effect of the depth used in the Minimax Algorithm.

g) How could you improve the evaluation function for this type of agent?

State and Operator/Move Representation

```
HEI = 6
WID = 7

class State:
      board: List[List[int]] # with [HEI][WID]
      player: int
      lastMoveX: int   #optional
      lastMoveY: int   #optional
      nmoves: int    #optional

      def __init__(hei = HEI, wid = WID):
     self.board = [[0] * hei] * wid
     nmoves = 0
     player = 1 # st.player=2;

class Movem:
      Col: int
```

Functions Needed for a Simple Game

```
def draw_state(st: State) -> None

def get_human_mov(st: State) -> Movem   #optional
def get_pc_rand_mov(st: State) -> Movem  #optional
def get_pc_minimax_mov(st: State, depth: int) -> Movem

def valid_movement(st: State, mov: Movem) -> bool
def execute_movement(st: State, mov: Movem) -> State

def check_winner(st: State) -> int       #Objective
Test
def evaluate(st: State, pl: int) -> int //For minimax
```

Simple Game Engine

```
def main():
 state = State()
 draw_state(state)

 while check_winner(state) != -1:
   if (state.player == 1):
     mov = get_pc_minimax_mov(state)
   else:
     mov = get_human_mov(state)
   state = execute_movement(state, mov)
   draw_state(state)

 print("Winner:", check_winner(st))
```

Inititialization and Drawing

```
def draw_state(st: State):
   print("| 1 | 2 | 3 | 4 | 5 | 6 | 7 |")
   print("-----------------------------")
   for i in range(HEI-1, -1, -1):
      for x in range(0, WID):
         if st.board[y][x] == 0:print("|   ")
         elif (st.board[y][x] == 1):print("| X ")
         elif (st.board[y][x] == 2):print("| O ")
         if (x==WID-1): print("|\n")
   print("-----------------------------")
```

Operators: Preconditions and Effects

```
def valid_movement(st: state , mov: Movem):
   return (mov.col>=1 && mov.col<= WID and
          st.board[HEI-1][mov.col-1]==0)

def execute_movement(st: state , mov: Movem):
   int i=0
   while (st.board[i][mov.col-1] != 0): i += 1
   st.board[i][mov.col-1]=st.player;
   st.lastMoveY = i
   st.lastMoveX = mov.col-1
   st.player= 3-st.player
   St.nmoves += 1
   return st
```

Human and Random Moves

```
def get_human_mov(st: State):
  mov = Movem()
  while not valid_movement(st, mov):
    mov.col = input("\nPlayer " + str(st.player)
                  + " Please Select Move (1-7):")
  return mov

def get_pc_rand_mov(st: State):
  mov = Movem()
  while not valid_movement(st, mov):
    mov.col=random.randint(0,7)
  return mov
```

Minimax with Alpha-Beta in Python

```
def minimax(state, depth, playerMax, alpha, beta):
  if depth==0 or state.isEndState(): return evalFunct(state)

  if playerMax:
    maxEval = -math.inf
    for move in state.validMoves():
      evaluation = minimax(move, depth-1, False, alpha, beta)
      maxEval = max(maxEval, evaluation)
      alpha = max(alpha, evaluation)
      if beta <= alpha:    break
    return maxEval

    minEval = math.inf
    for move in state.validMoves():
      evaluation = minimax(move, depth-1, True, alpha, beta)
      minEval = min(minEval, evaluation)
      beta = min(beta, evaluation)
      if beta <= alpha:   break
    return minEval
```

IART - Artificial Intelligence

Exercise 3: Adversarial Search Problems

Luís Paulo Reis

LIACC – Artificial Intelligence and Computer Science Lab.
DEI/FEUP – Informatics Engineering Department, Faculty of Engineering of the University of Porto, Portugal
APPIA – Portuguese Association for Artificial Intelligence