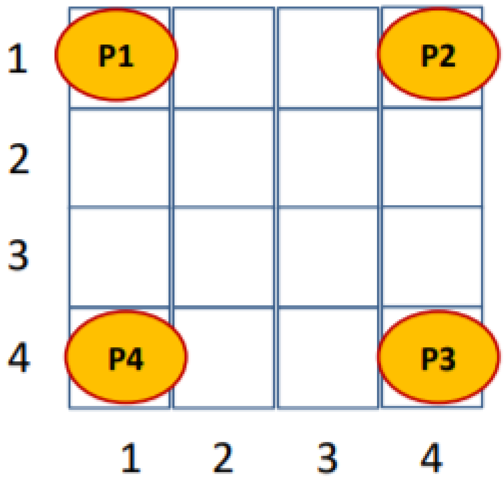
CS 6364 HW 2

Rohith Peddi, RXP190007

October 8, 2020

1 MULTI PLAYER GAMES

Four players are playing a game which is illustrated in the following Figure:



Player 1 is initially positioned in square [1,1], Player 2 is positioned in square [1,4], Player 3 is positioned in square [4,4] and Player 4 is positioned in square [1,4]. The player that will reach first the diagonally opposite position will win the game. Player 1 will start the game, followed by Player 2, Player 3 and Player 4.

All Players can move either up, down, left or right, if the square in that direction is available and not occupied by any other player.

For example, Player 1 initially can move only to the right or down.

1.1 Question 1

How do you represent each node of the game? (3 points) How is the initial node of the game represented? (1 point)

SOLUTION

NODE

Each **node** of the game tree consists of the following:

State	Parent	Action	Depth	Children

Game state can be represented with the following:

Current player	Board configuration	Utility	Moves

Every state can be uniquely identified by **current player, board configuration**. Compactly represented as follows:

to-move	P1	P2	P3	P4
[P1 or P2 or P3 or P4]	$(P1_x, P1_y)$	$(P2_x, P2_y)$	$(P3_x, P3_y)$	$(P4_x, P4_y)$

Following are the **actions/moves** available for each player

Actions	MOVE_LEFT	MOVE_RIGHT	MOVE_TOP	MOVE_BOTTOM

Utility of a state can be expressed as a vector with corresponding utility values for each player:

P1	P2	P3	P4

<u>INITIAL NODE</u>

Initial game state is represented as follows :

to-move	P1	P2	P3	P4
P1	(1,1)	(4,1)	(4,4)	(1,4)

1.2 Programming Assignment

A. Write code that generates the game tree for this multi-player game. (15 points)

The output should use the following format: [Current player =???— Father node (if not initial node) =???— Action= ????— Current game node =???—

if the game node is repeated, write REPEATED; if the current game node corresponds to a winning situation for one of the players, write WINS[PLAYER???]]

Hint: Successors of repeated game nodes should not be considered!

SOLUTION

Code consists of three segments :
Game Representation [Class], Node [Class], Function to compute game tree

Few functions are not included in this document, they are part of the file P1.py, attached with this submission

GAME STATE

```
"""
Class GameState is used to identify the current state of the game.
State is uniquely identified by two parameters : [TO_MOVE, BOARD CONFIGURATION]
"""
class GameState:

    def __init__(self, to_move, board, utility, moves):
        self.to_move = to_move
        self.utility = utility
        self.board = board
        self.moves = moves

    # Two states are equal if they have same to_move and board configurations
    def __eq__(self, other):
        if isinstance(other, GameState):
            return self.to_move == other.to_move and self.board == other.board
        return False
```

GAME REPRESENTATION

```
"""
Class that represents a game
"""
class MultiPlayerGame:

    def __init__(self, to_move, board):
        utility = self.compute_utility(board)
        moves = self.compute_moves(to_move, board)
        self.initial = GameState(to_move=to_move, board=board, utility=utility, moves=moves)

    def actions(self, state):
        return state.moves

    def result(self, state, move):
        if move not in state.moves:
            return state
        # Finding the next player
        current_player = state.to_move
        if current_player == 'P1':
            next_player = 'P2'
        elif current_player == 'P2':
            next_player = 'P3'
        elif current_player == 'P3':
            next_player = 'P4'
        elif current_player == 'P4':
            next_player = 'P1'
        new_board = self.new_board_config(state, move)
        new_moves = self.compute_moves(next_player, new_board)
        # If a new state has no moves associated with it then it results in ZERO utility for
    each player
        new_utility = dict(P1=0, P2=0, P3=0, P4=0) if len(new_moves) == 0 else self.
    compute_utility(new_board)
```

```
        return GameState(to_move=next_player, board=new_board, utility=new_utility, moves=new_moves)

    def terminal_test(self, state):
        current_board = state.board
        return current_board['P1'] == (4, 4) or current_board['P2'] == (1, 4) \
              or current_board['P3'] == (1, 1) or current_board['P4'] == (4, 1) or len(state.moves) == 0

    def compute_utility(self, board):
        players = ['P1', 'P2', 'P3', 'P4']
        for player in players:
            if player == 'P1' and board[player] == (4, 4):
                return dict(P1=200, P2=10, P3=30, P4=10)
            elif player == 'P2' and board[player] == (1, 4):
                return dict(P1=100, P2=300, P3=150, P4=200)
            elif player == 'P3' and board[player] == (1, 1):
                return dict(P1=150, P2=200, P3=400, P4=300)
            elif player == 'P4' and board[player] == (4, 1):
                return dict(P1=220, P2=330, P3=440, P4=500)
        return dict(P1='?', P2='?', P3='?', P4='?')

    def compute_moves(self, to_move, board):
        players = ['P1', 'P2', 'P3', 'P4']
        x, y = board[to_move]
        blocked_positions = [board[other_player] for other_player in players if other_player is not to_move]
        actions = ['MOVE_LEFT', 'MOVE_RIGHT', 'MOVE_TOP', 'MOVE_BOTTOM']
        # Imposing boundary restrictions
        if x >= 4:
            actions.remove('MOVE_RIGHT')
        elif x <= 1:
            actions.remove('MOVE_LEFT')
        if y >= 4:
            actions.remove('MOVE_BOTTOM')
        elif y <= 1:
            actions.remove('MOVE_TOP')

        # Imposing blocking restrictions by other players
        possible_actions = actions.copy()
        for action in possible_actions:
            xn = x
            yn = y
            if action == 'MOVE_LEFT':
                xn = x - 1
            elif action == 'MOVE_RIGHT':
                xn = x + 1
            elif action == 'MOVE_TOP':
                yn = y - 1
            elif action == 'MOVE_BOTTOM':
                yn = y + 1
            if (xn, yn) in blocked_positions:
                actions.remove(action)
        return actions
```

NODE

```
    """
Class that represents a node in the game tree
"""

class Node:

    def __init__(self, state, parent=None, action=None):
        """Create a search tree Node, derived from a parent by an action."""
        self.state = state
        self.parent = parent
        self.action = action
        self.depth = 0
        self.children = []
        self.best_action = None
        if parent:
            self.depth = parent.depth + 1

    def __repr__(self):
        node_string = "to_move : " + str(self.state.to_move)
```

```python
        node_string += ", board : " + str(self.state.board)
        return node_string

    def expand(self, game):
        """List the nodes reachable in one step from this node."""
        for action in game.actions(self.state):
            child = self.child_node(game, action)
            self.children.append(child)
        return self.children

    def child_node(self, game, action):
        next_state = game.result(self.state, action)
        next_node = Node(next_state, self, action)
        return next_node

    def __hash__(self):
        return hash(self.state)

    def __eq__(self, other):
        return isinstance(other, Node) and self.state == other.state
```

BUILD GAME TREE

```python
"""
Segment that finds out the game tree.
Here inorder to find the game tree we use a breadth first simulation.
We consider all the possible actions for P1 from each state of the frontier and
then all possible actions for P2 from new nodes reached by P1 performing an action.
"""
def breadth_first_game_tree(game):
    initial_node = Node(game.initial)
    iteration = 0
    print_game_tree(initial_node, False, game, False, 0, iteration)
    bread_first_frontier = deque([initial_node])
    explored = []
    while bread_first_frontier:
        node = bread_first_frontier.popleft()
        explored.append(node.state)
        is_terminal = game.terminal_test(node.state)
        if is_terminal:
            continue
        for child in node.expand(game):
            iteration += 1
            is_terminal = game.terminal_test(child.state)
            if child.state in explored:
                print_game_tree(child, True, game, is_terminal, len(bread_first_frontier), iteration)
            else:
                if child not in bread_first_frontier:
                    print_game_tree(child, False, game, is_terminal, len(bread_first_frontier), iteration)
                    bread_first_frontier.append(child)
    return initial_node
```

SAMPLE OUPUT

Complete output is part of the file game_tree.txt, It is included as part of submission. It can also be generated by running the code P1.py [Runtime : 10 min]

```

[Current Player : P1 | Father Node : to_move : P4, board : {'P1': (1, 3), 'P2': (3, 4), 'P3': (2, 4), 'P4': (4, 4)} | Action : MOVE_TOP | Current node : to_move : P1, board : {'P1': (1, 3), 'P2': (3, 4), 'P3': (2, 4), 'P4': (4, 3)} | REPEATED ]
[Current Player : P1 | Father Node : to_move : P4, board : {'P1': (2, 2), 'P2': (1, 2), 'P3': (2, 4), 'P4': (1, 1)} | Action : MOVE_RIGHT | Current node : to_move : P1, board : {'P1': (2, 2), 'P2': (1, 2), 'P3': (2, 4), 'P4': (2, 1)} ]
[Current Player : P1 | Father Node : to_move : P4, board : {'P1': (2, 2), 'P2': (1, 2), 'P3': (1, 3), 'P4': (1, 1)} | Action : MOVE_RIGHT | Current node : to_move : P1, board : {'P1': (2, 2), 'P2': (1, 2), 'P3': (1, 3), 'P4': (2, 1)} ]
[Current Player : P1 | Father Node : to_move : P4, board : {'P1': (2, 2), 'P2': (1, 2), 'P3': (4, 4), 'P4': (1, 1)} | Action : MOVE_RIGHT | Current node : to_move : P1, board : {'P1': (2, 2), 'P2': (1, 2), 'P3': (4, 4), 'P4': (2, 1)} ]
```

```
[Current Player : P1 | Father Node : to_move : P4, board : {’P1’: (2, 2), ’P2’: (1, 2), ’P3’:
    (3, 3), ’P4’: (1, 1)} | Action : MOVE_RIGHT | Current node : to_move : P1, board : {’P1’:
    (2, 2), ’P2’: (1, 2), ’P3’: (3, 3), ’P4’: (2, 1)} ]
[Current Player : P1 | Father Node : to_move : P4, board : {’P1’: (2, 2), ’P2’: (1, 2), ’P3’:
    (4, 2), ’P4’: (1, 1)} | Action : MOVE_RIGHT | Current node : to_move : P1, board : {’P1’:
    (2, 2), ’P2’: (1, 2), ’P3’: (4, 2), ’P4’: (2, 1)} ]
[Current Player : P1 | Father Node : to_move : P4, board : {’P1’: (2, 2), ’P2’: (1, 2), ’P3’:
    (3, 1), ’P4’: (1, 1)} | Action : MOVE_RIGHT | Current node : to_move : P1, board : {’P1’:
    (2, 2), ’P2’: (1, 2), ’P3’: (3, 1), ’P4’: (2, 1)} ]
[Current Player : P1 | Father Node : to_move : P4, board : {’P1’: (2, 2), ’P2’: (2, 1), ’P3’:
    (1, 3), ’P4’: (4, 2)} | Action : MOVE_LEFT | Current node : to_move : P1, board : {’P1’:
    (2, 2), ’P2’: (2, 1), ’P3’: (1, 3), ’P4’: (3, 2)} | REPEATED ]
[Current Player : P1 | Father Node : to_move : P4, board : {’P1’: (2, 2), ’P2’: (2, 1), ’P3’:
    (1, 3), ’P4’: (4, 2)} | Action : MOVE_TOP | Current node : to_move : P1, board : {’P1’:
    (2, 2), ’P2’: (2, 1), ’P3’: (1, 3), ’P4’: (4, 1)} | WINS [ P4 ]
[Current Player : P1 | Father Node : to_move : P4, board : {’P1’: (2, 2), ’P2’: (2, 1), ’P3’:
    (1, 3), ’P4’: (4, 2)} | Action : MOVE_BOTTOM | Current node : to_move : P1, board : {’P1’:
     (2, 2), ’P2’: (2, 1), ’P3’: (1, 3), ’P4’: (4, 3)} | REPEATED ]
[Current Player : P1 | Father Node : to_move : P4, board : {’P1’: (2, 2), ’P2’: (2, 1), ’P3’:
    (3, 3), ’P4’: (4, 2)} | Action : MOVE_LEFT | Current node : to_move : P1, board : {’P1’:
    (2, 2), ’P2’: (2, 1), ’P3’: (3, 3), ’P4’: (3, 2)} ]
[Current Player : P1 | Father Node : to_move : P4, board : {’P1’: (2, 2), ’P2’: (2, 1), ’P3’:
    (3, 3), ’P4’: (4, 2)} | Action : MOVE_TOP | Current node : to_move : P1, board : {’P1’:
    (2, 2), ’P2’: (2, 1), ’P3’: (3, 3), ’P4’: (4, 1)} | WINS [ P4 ]
[Current Player : P1 | Father Node : to_move : P4, board : {’P1’: (2, 2), ’P2’: (2, 1), ’P3’:
    (3, 3), ’P4’: (4, 2)} | Action : MOVE_BOTTOM | Current node : to_move : P1, board : {’P1’:
     (2, 2), ’P2’: (2, 1), ’P3’: (3, 3), ’P4’: (4, 3)} | REPEATED ]
[Current Player : P1 | Father Node : to_move : P4, board : {’P1’: (2, 2), ’P2’: (2, 1), ’P3’:
    (2, 4), ’P4’: (4, 2)} | Action : MOVE_LEFT | Current node : to_move : P1, board : {’P1’:
    (2, 2), ’P2’: (2, 1), ’P3’: (2, 4), ’P4’: (3, 2)} | REPEATED ]
[Current Player : P1 | Father Node : to_move : P4, board : {’P1’: (2, 2), ’P2’: (2, 1), ’P3’:
    (2, 4), ’P4’: (4, 2)} | Action : MOVE_TOP | Current node : to_move : P1, board : {’P1’:
    (2, 2), ’P2’: (2, 1), ’P3’: (2, 4), ’P4’: (4, 1)} | WINS [ P4 ]
[Current Player : P1 | Father Node : to_move : P4, board : {’P1’: (2, 2), ’P2’: (2, 1), ’P3’:
    (2, 4), ’P4’: (4, 2)} | Action : MOVE_BOTTOM | Current node : to_move : P1, board : {’P1’:
     (2, 2), ’P2’: (2, 1), ’P3’: (2, 4), ’P4’: (4, 3)} | REPEATED ]
[Current Player : P1 | Father Node : to_move : P4, board : {’P1’: (2, 2), ’P2’: (1, 2), ’P3’:
    (1, 3), ’P4’: (4, 2)} | Action : MOVE_LEFT | Current node : to_move : P1, board : {’P1’:
    (2, 2), ’P2’: (1, 2), ’P3’: (1, 3), ’P4’: (3, 2)} | REPEATED ]
```

1.3 Question 2

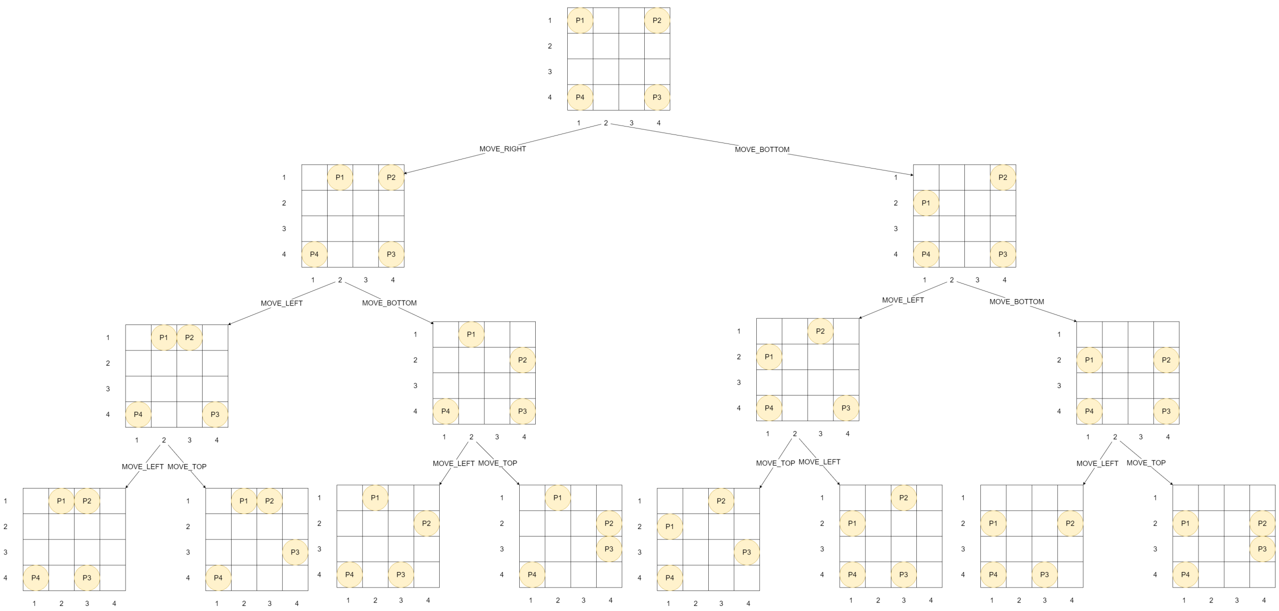
Draw the game tree based on the output of your code. (11 points) Note: You can draw the game tree on a piece of paper, take a picture and attach it to your answers, which will be returned in a PDF file.

SOLUTION

As conveyed by the professor, this answer consists of three segments:

1. Game tree generated by code upto 3 levels
2. Terminal states along with parent values
3. Complete game tree is present in game_tree.txt file attached with this submission

1. Game tree upto 3 levels - PICTORIAL REPRESENTATION



1. Game tree upto 3 levels - Output by code

```
[Current Player : P1 | Father Node : None | Action : None | Current node : to_move : P1, board : {'P1': (1, 1), 'P2': (4, 1), 'P3': (4, 4), 'P4': (1, 4)} ]
----------------------------------------------------------------------------
[Current Player : P2 | Father Node : to_move : P1, board : {'P1': (1, 1), 'P2': (4, 1), 'P3': (4, 4), 'P4': (1, 4)} | Action : MOVE_RIGHT | Current node : to_move : P2, board : {'P1': (2, 1), 'P2': (4, 1), 'P3': (4, 4), 'P4': (1, 4)} ]
[Current Player : P2 | Father Node : to_move : P1, board : {'P1': (1, 1), 'P2': (4, 1), 'P3': (4, 4), 'P4': (1, 4)} | Action : MOVE_BOTTOM | Current node : to_move : P2, board : {'P1': (1, 2), 'P2': (4, 1), 'P3': (4, 4), 'P4': (1, 4)} ]
----------------------------------------------------------------------------
[Current Player : P3 | Father Node : to_move : P2, board : {'P1': (2, 1), 'P2': (4, 1), 'P3': (4, 4), 'P4': (1, 4)} | Action : MOVE_LEFT | Current node : to_move : P3, board : {'P1': (2, 1), 'P2': (3, 1), 'P3': (4, 4), 'P4': (1, 4)} ]
[Current Player : P3 | Father Node : to_move : P2, board : {'P1': (2, 1), 'P2': (4, 1), 'P3': (4, 4), 'P4': (1, 4)} | Action : MOVE_BOTTOM | Current node : to_move : P3, board : {'P1': (2, 1), 'P2': (4, 2), 'P3': (4, 4), 'P4': (1, 4)} ]
[Current Player : P3 | Father Node : to_move : P2, board : {'P1': (1, 2), 'P2': (4, 1), 'P3': (4, 4), 'P4': (1, 4)} | Action : MOVE_LEFT | Current node : to_move : P3, board : {'P1': (1, 2), 'P2': (3, 1), 'P3': (4, 4), 'P4': (1, 4)} ]
[Current Player : P3 | Father Node : to_move : P2, board : {'P1': (1, 2), 'P2': (4, 1), 'P3': (4, 4), 'P4': (1, 4)} | Action : MOVE_BOTTOM | Current node : to_move : P3, board : {'P1': (1, 2), 'P2': (4, 2), 'P3': (4, 4), 'P4': (1, 4)} ]
----------------------------------------------------------------------------
[Current Player : P4 | Father Node : to_move : P3, board : {'P1': (2, 1), 'P2': (3, 1), 'P3': (4, 4), 'P4': (1, 4)} | Action : MOVE_LEFT | Current node : to_move : P4, board : {'P1': (2, 1), 'P2': (3, 1), 'P3': (3, 4), 'P4': (1, 4)} ]
```

```
[Current Player : P4 | Father Node : to_move : P3, board : {’P1’: (2, 1), ’P2’: (3, 1), ’P3’:
    (4, 4), ’P4’: (1, 4)} | Action : MOVE_TOP | Current node : to_move : P4, board : {’P1’:
    (2, 1), ’P2’: (3, 1), ’P3’: (4, 3), ’P4’: (1, 4)} ]
[Current Player : P4 | Father Node : to_move : P3, board : {’P1’: (2, 1), ’P2’: (4, 2), ’P3’:
    (4, 4), ’P4’: (1, 4)} | Action : MOVE_LEFT | Current node : to_move : P4, board : {’P1’:
    (2, 1), ’P2’: (4, 2), ’P3’: (3, 4), ’P4’: (1, 4)} ]
[Current Player : P4 | Father Node : to_move : P3, board : {’P1’: (2, 1), ’P2’: (4, 2), ’P3’:
    (4, 4), ’P4’: (1, 4)} | Action : MOVE_TOP | Current node : to_move : P4, board : {’P1’:
    (2, 1), ’P2’: (4, 2), ’P3’: (4, 3), ’P4’: (1, 4)} ]
[Current Player : P4 | Father Node : to_move : P3, board : {’P1’: (1, 2), ’P2’: (3, 1), ’P3’:
    (4, 4), ’P4’: (1, 4)} | Action : MOVE_LEFT | Current node : to_move : P4, board : {’P1’:
    (1, 2), ’P2’: (3, 1), ’P3’: (3, 4), ’P4’: (1, 4)} ]
[Current Player : P4 | Father Node : to_move : P3, board : {’P1’: (1, 2), ’P2’: (3, 1), ’P3’:
    (4, 4), ’P4’: (1, 4)} | Action : MOVE_TOP | Current node : to_move : P4, board : {’P1’:
    (1, 2), ’P2’: (3, 1), ’P3’: (4, 3), ’P4’: (1, 4)} ]
[Current Player : P4 | Father Node : to_move : P3, board : {’P1’: (1, 2), ’P2’: (4, 2), ’P3’:
    (4, 4), ’P4’: (1, 4)} | Action : MOVE_LEFT | Current node : to_move : P4, board : {’P1’:
    (1, 2), ’P2’: (4, 2), ’P3’: (3, 4), ’P4’: (1, 4)} ]
[Current Player : P4 | Father Node : to_move : P3, board : {’P1’: (1, 2), ’P2’: (4, 2), ’P3’:
    (4, 4), ’P4’: (1, 4)} | Action : MOVE_TOP | Current node : to_move : P4, board : {’P1’:
    (1, 2), ’P2’: (4, 2), ’P3’: (4, 3), ’P4’: (1, 4)} ]
```

2. Terminal states along with parent

Terminal states in this game can be of two types:

1. Terminal states with no moves
2. Terminal state with one of the player winning

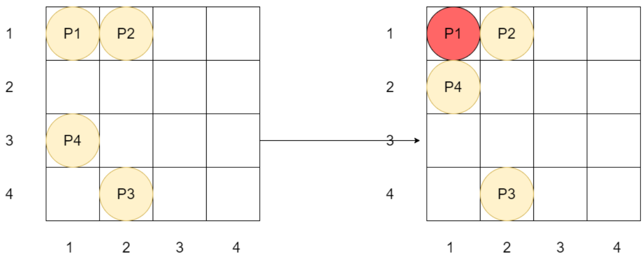
As there are many terminal states in both the category, I have listed some of them

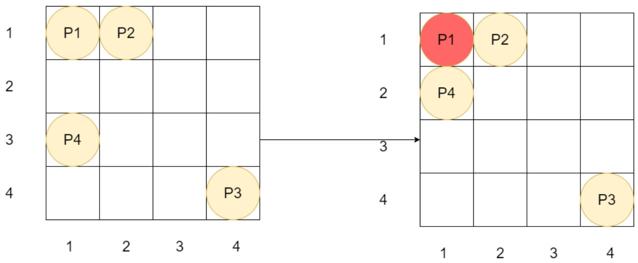
2. Terminal states - NO MOVES [Ouput by code]

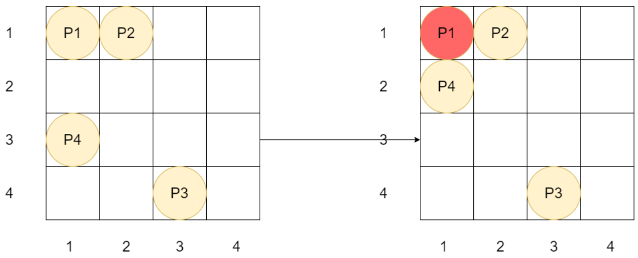
These are only few of many, all of them are part of the file game_tree.txt included along with this submission

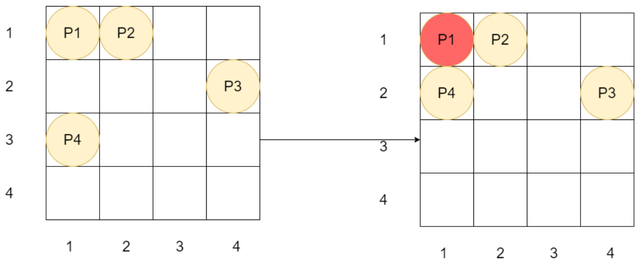
```
[Current Player : P1 | Father Node : to_move : P4, board : {’P1’: (1, 1), ’P2’: (2, 1), ’P3’:
    (2, 4), ’P4’: (1, 3)} | Action : MOVE_TOP | Current node : to_move : P1, board : {’P1’:
    (1, 1), ’P2’: (2, 1), ’P3’: (2, 4), ’P4’: (1, 2)} | NO MOVES  ]
[Current Player : P1 | Father Node : to_move : P4, board : {’P1’: (1, 1), ’P2’: (2, 1), ’P3’:
    (4, 4), ’P4’: (1, 3)} | Action : MOVE_TOP | Current node : to_move : P1, board : {’P1’:
    (1, 1), ’P2’: (2, 1), ’P3’: (4, 4), ’P4’: (1, 2)} | NO MOVES  ]
[Current Player : P1 | Father Node : to_move : P4, board : {’P1’: (1, 1), ’P2’: (2, 1), ’P3’:
    (3, 3), ’P4’: (1, 3)} | Action : MOVE_TOP | Current node : to_move : P1, board : {’P1’:
    (1, 1), ’P2’: (2, 1), ’P3’: (3, 3), ’P4’: (1, 2)} | NO MOVES  ]
[Current Player : P1 | Father Node : to_move : P4, board : {’P1’: (1, 1), ’P2’: (2, 1), ’P3’:
    (4, 2), ’P4’: (1, 3)} | Action : MOVE_TOP | Current node : to_move : P1, board : {’P1’:
    (1, 1), ’P2’: (2, 1), ’P3’: (4, 2), ’P4’: (1, 2)} | NO MOVES  ]
```

2. Terminal states - NO MOVES [PICTORIAL REPRESENTATION]





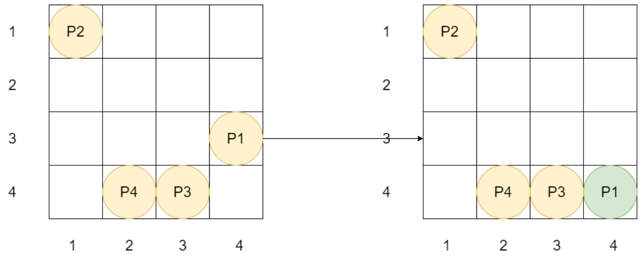


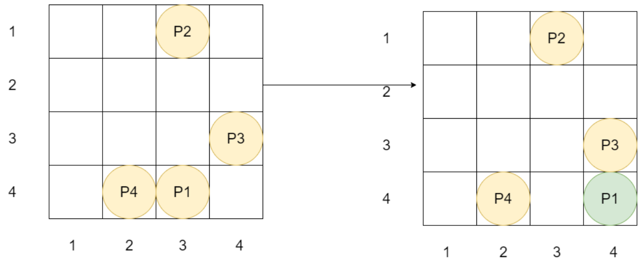


2. Terminal states - WINNING STATES [Output by code]

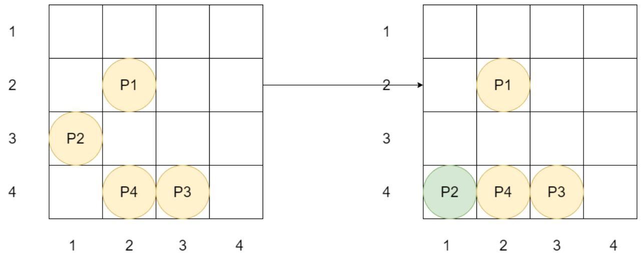
These are only few of many, all of them are part of the file game_tree.txt included along with this submission

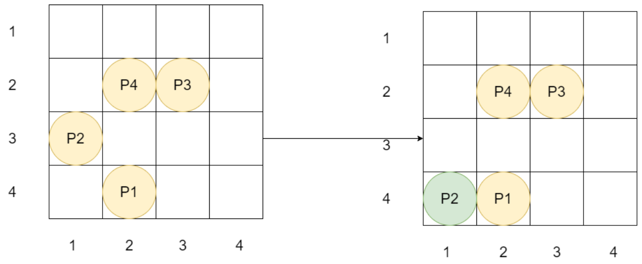
```
"""WINS[P1]"""

[Current Player : P2 | Father Node : to_move : P1, board : {’P1’: (4, 3), ’P2’: (1, 1), ’P3’:
    (3, 4), ’P4’: (2, 4)} | Action : MOVE_BOTTOM | Current node : to_move : P2, board : {’P1’:
     (4, 4), ’P2’: (1, 1), ’P3’: (3, 4), ’P4’: (2, 4)} | WINS [ P1 ]

[Current Player : P2 | Father Node : to_move : P1, board : {’P1’: (3, 4), ’P2’: (3, 1), ’P3’:
    (4, 3), ’P4’: (2, 4)} | Action : MOVE_RIGHT | Current node : to_move : P2, board : {’P1’:
    (4, 4), ’P2’: (3, 1), ’P3’: (4, 3), ’P4’: (2, 4)} | WINS [ P1 ]
```



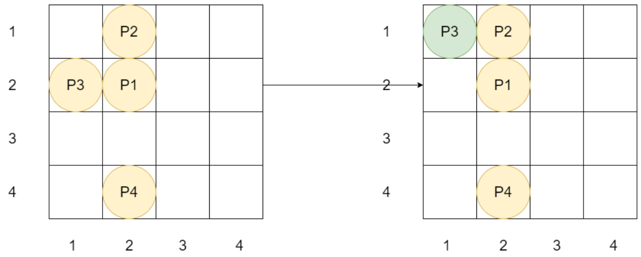


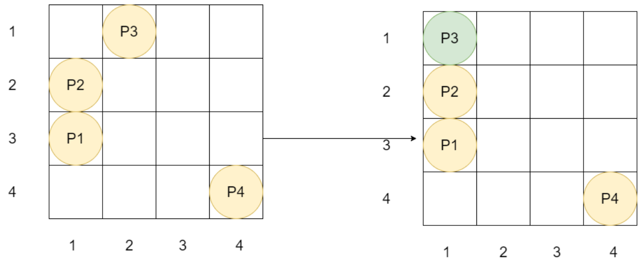
```
"""WINS[P2]"""

[Current Player : P3 | Father Node : to_move : P2, board : {’P1’: (2, 2), ’P2’: (1, 3), ’P3’: (3, 4), ’P4’: (2, 4)} | Action : MOVE_BOTTOM | Current node : to_move : P3, board : {’P1’: (2, 2), ’P2’: (1, 4), ’P3’: (3, 4), ’P4’: (2, 4)} | WINS [ P2 ]
[Current Player : P3 | Father Node : to_move : P2, board : {’P1’: (2, 4), ’P2’: (1, 3), ’P3’: (3, 2), ’P4’: (2, 2)} | Action : MOVE_BOTTOM | Current node : to_move : P3, board : {’P1’: (2, 4), ’P2’: (1, 4), ’P3’: (3, 2), ’P4’: (2, 2)} | WINS [ P2 ]
```



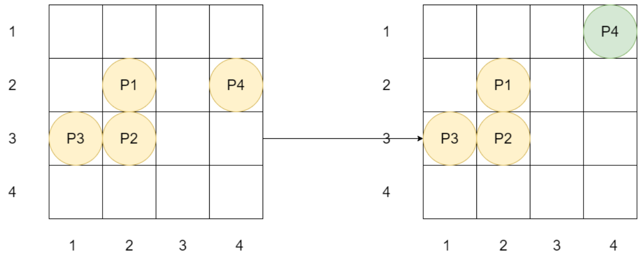


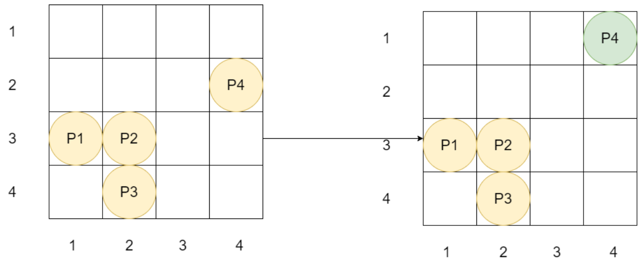
```
"""WINS[P3]"""

[Current Player : P4 | Father Node : to_move : P3, board : {’P1’: (2, 2), ’P2’: (2, 1), ’P3’: (1, 2), ’P4’: (2, 4)} | Action : MOVE_TOP | Current node : to_move : P4, board : {’P1’: (2, 2), ’P2’: (2, 1), ’P3’: (1, 1), ’P4’: (2, 4)} | WINS [ P3 ]
[Current Player : P4 | Father Node : to_move : P3, board : {’P1’: (1, 3), ’P2’: (1, 2), ’P3’: (2, 1), ’P4’: (4, 4)} | Action : MOVE_LEFT | Current node : to_move : P4, board : {’P1’: (1, 3), ’P2’: (1, 2), ’P3’: (1, 1), ’P4’: (4, 4)} | WINS [ P3 ]
```





```
"""WINS[P4]"""

[Current Player : P1 | Father Node : to_move : P4, board : {’P1’: (2, 2), ’P2’: (2, 1), ’P3’:
    (1, 3), ’P4’: (4, 2)} | Action : MOVE_TOP | Current node : to_move : P1, board : {’P1’:
    (2, 2), ’P2’: (2, 1), ’P3’: (1, 3), ’P4’: (4, 1)} | WINS [ P4 ]
[Current Player : P1 | Father Node : to_move : P4, board : {’P1’: (1, 3), ’P2’: (2, 3), ’P3’:
    (2, 4), ’P4’: (4, 2)} | Action : MOVE_TOP | Current node : to_move : P1, board : {’P1’:
    (1, 3), ’P2’: (2, 3), ’P3’: (2, 4), ’P4’: (4, 1)} | WINS [ P4 ]
```





Complete game tree is included as part of file game_tree.txt, this file is included along with submission and also can be generated by running P1.py

1.4 Question 3

If Player 1 wins, the utility should be 200. If Player 2 wins, the utility should be 300. If Player 3 wins, the utility should be 400 and if Player 4 wins, the Utility should be 500. In any case, because this is not a zero-sum game, the other players also receive

Utility values:

1. When Player 1 wins, Player 2 receives utility=10, Player 3 receives utility=30, and Player 4 receives utility=10;
2. When Player 2 wins, Player 1 receives utility=100, Player 3 receives utility=150, and Player 4 receives utility=200;
3. When Player 3 wins, Player 1 receives utility=150, Player 2 receives utility=200, and Player 4 receives utility=300;
4. When Player 4 wins, Player 1 receives utility=220, Player 2 receives utility=330, and Player 3 receives utility=440;

If the minimax values of the repeated states shall be ignored, compute the MINIMAX values in all other states of the game, using the following program assignment.

1.5 Programming Assignment

B. Write code that computes the MINIMAX values for all non-repeated game nodes and display the values of MINIMAX in the following format: [Current player = . . . — Father node (if not initial node) = . . . — Action= ????— Current game node =. . . — if the current game node corresponds to a winning situation for one of the players, write WINS[PLAYER???]— MINIMAX = ?????]

SOLUTION FOR QUESTION 3 and 4

Preliminary observations

1. Based on given utility values for terminal states, **players receive greater utility when ’P4’ wins.**
2. Minimax decision should include path to states where P4 wins.

CODE TO COMPUTE MINIMAX VALUES

Generating MINIMAX values for each node include the following steps:

1. Performing a **depth first search on the game tree** generated above as stated in textbook
2. Inorder to print game tree with MINIMAX values we perform a **breadth first search** to generate nodes with utility values

```

"""
Here we compute minimax values by performing a depth first search through game tree.
As it is a multiplayer game, each player tries to maximize their utility values.
Utility of every non terminal state is initialised to {P1:’?’, P2: ’?’, P3: ’?’, P4: ’?’}
Utility of every terminal node is initialised as given in the problem
"""

def minimax_value(node):
    player = node.state.to_move
    for child in node.children:
        child_utility = child.state.utility
        if child_utility[player] == ’?’:
            child_utility = minimax_value(child)
        if child_utility[player] is not ’?’:
            if node.state.utility[player] == ’?’ or (node.state.utility[player] < child_utility[player]):
                node.state.utility = child.state.utility
                node.best_action = child.action
    return node.state.utility
```

CODE TO GENERATE GAME TREE WITH MINIMAX VALUES

```python
"""
Printing the game tree with Minimax Values associated with it.
Here we again traverse the game tree in a breadth first manner to print all nodes of the game tree.
"""

def game_tree_with_minimax_values(game, game_head):
    iteration = 0
    print_game_tree(game_head, False, game, False, 0, iteration, True)
    bread_first_frontier = deque([game_head])
    explored = []
    while bread_first_frontier:
        node = bread_first_frontier.popleft()
        explored.append(node.state)
        is_terminal = game.terminal_test(node.state)
        if is_terminal:
            continue
        for child in node.children:
            iteration += 1
            is_terminal = game.terminal_test(child.state)
            if child.state in explored:
                print_game_tree(child, True, game, is_terminal, len(bread_first_frontier), iteration, True)
            else:
                if child not in bread_first_frontier:
                    print_game_tree(child, False, game, is_terminal, len(bread_first_frontier), iteration, True)
                    bread_first_frontier.append(child)
```

Observations

1. Minimax values of all repeated states is ignored
2. Nodes whose children are all repeated states are also ignored

SAMPLE OUTPUT

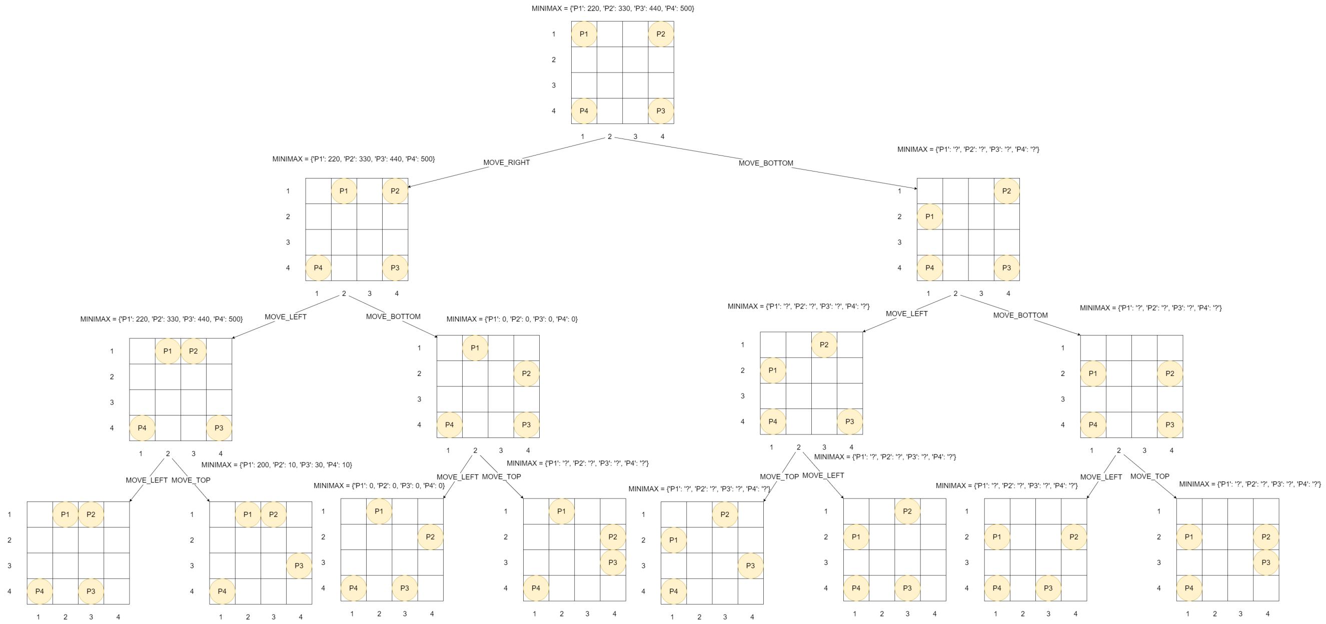
Complete output is part of the file game_tree_minimax.txt, which is included as part of this submission. It can also be generated by running the file P1.py

```
[Current Player : P1 | Father Node : None | Action : None | Current node : to_move : P1, board : {’P1’: (1, 1), ’P2’: (4, 1), ’P3’: (4, 4), ’P4’: (1, 4)} | MINIMAX = {’P1’: 220, ’P2’: 330, ’P3’: 440, ’P4’: 500} ]
---------------------------------------------------------
[Current Player : P2 | Father Node : to_move : P1, board : {’P1’: (1, 1), ’P2’: (4, 1), ’P3’: (4, 4), ’P4’: (1, 4)} | Action : MOVE_RIGHT | Current node : to_move : P2, board : {’P1’: (2, 1), ’P2’: (4, 1), ’P3’: (4, 4), ’P4’: (1, 4)} | MINIMAX = {’P1’: 220, ’P2’: 330, ’P3’: 440, ’P4’: 500} ]
[Current Player : P2 | Father Node : to_move : P1, board : {’P1’: (1, 1), ’P2’: (4, 1), ’P3’: (4, 4), ’P4’: (1, 4)} | Action : MOVE_BOTTOM | Current node : to_move : P2, board : {’P1’: (1, 2), ’P2’: (4, 1), ’P3’: (4, 4), ’P4’: (1, 4)} | MINIMAX = {’P1’: ’?’, ’P2’: ’?’, ’P3’: ’?’, ’P4’: ’?’} ]
----------------------------------------------------------
[Current Player : P3 | Father Node : to_move : P2, board : {’P1’: (2, 1), ’P2’: (4, 1), ’P3’: (4, 4), ’P4’: (1, 4)} | Action : MOVE_LEFT | Current node : to_move : P3, board : {’P1’: (2, 1), ’P2’: (3, 1), ’P3’: (4, 4), ’P4’: (1, 4)} | MINIMAX = {’P1’: 220, ’P2’: 330, ’P3’: 440, ’P4’: 500} ]
[Current Player : P3 | Father Node : to_move : P2, board : {’P1’: (2, 1), ’P2’: (4, 1), ’P3’: (4, 4), ’P4’: (1, 4)} | Action : MOVE_BOTTOM | Current node : to_move : P3, board : {’P1’: (2, 1), ’P2’: (4, 2), ’P3’: (4, 4), ’P4’: (1, 4)} | MINIMAX = {’P1’: 0, ’P2’: 0, ’P3’: 0, ’P4’: 0} ]
[Current Player : P3 | Father Node : to_move : P2, board : {’P1’: (1, 2), ’P2’: (4, 1), ’P3’: (4, 4), ’P4’: (1, 4)} | Action : MOVE_LEFT | Current node : to_move : P3, board : {’P1’: (1, 2), ’P2’: (3, 1), ’P3’: (4, 4), ’P4’: (1, 4)} | MINIMAX = {’P1’: ’?’, ’P2’: ’?’, ’P3’: ’?’, ’P4’: ’?’} ]
[Current Player : P3 | Father Node : to_move : P2, board : {’P1’: (1, 2), ’P2’: (4, 1), ’P3’: (4, 4), ’P4’: (1, 4)} | Action : MOVE_BOTTOM | Current node : to_move : P3, board : {’P1’:
     (1, 2), 'P2': (4, 2), 'P3': (4, 4), 'P4': (1, 4)} | MINIMAX = {'P1': '?', 'P2': '?', 'P3'
    : '?', 'P4': '?'} ]
---------------------------------------------------------
[Current Player : P4 | Father Node : to_move : P3, board : {'P1': (2, 1), 'P2': (3, 1), 'P3':
    (4, 4), 'P4': (1, 4)} | Action : MOVE_LEFT | Current node : to_move : P4, board : {'P1':
    (2, 1), 'P2': (3, 1), 'P3': (3, 4), 'P4': (1, 4)} | MINIMAX = {'P1': 220, 'P2': 330, 'P3':
     440, 'P4': 500} ]
[Current Player : P4 | Father Node : to_move : P3, board : {'P1': (2, 1), 'P2': (3, 1), 'P3':
    (4, 4), 'P4': (1, 4)} | Action : MOVE_TOP | Current node : to_move : P4, board : {'P1':
    (2, 1), 'P2': (3, 1), 'P3': (4, 3), 'P4': (1, 4)} | MINIMAX = {'P1': 200, 'P2': 10, 'P3':
    30, 'P4': 10} ]
[Current Player : P4 | Father Node : to_move : P3, board : {'P1': (2, 1), 'P2': (4, 2), 'P3':
    (4, 4), 'P4': (1, 4)} | Action : MOVE_LEFT | Current node : to_move : P4, board : {'P1':
    (2, 1), 'P2': (4, 2), 'P3': (3, 4), 'P4': (1, 4)} | MINIMAX = {'P1': 0, 'P2': 0, 'P3': 0,
    'P4': 0} ]
[Current Player : P4 | Father Node : to_move : P3, board : {'P1': (2, 1), 'P2': (4, 2), 'P3':
    (4, 4), 'P4': (1, 4)} | Action : MOVE_TOP | Current node : to_move : P4, board : {'P1':
    (2, 1), 'P2': (4, 2), 'P3': (4, 3), 'P4': (1, 4)} | MINIMAX = {'P1': '?', 'P2': '?', 'P3':
     '?', 'P4': '?'} ]
[Current Player : P4 | Father Node : to_move : P3, board : {'P1': (1, 2), 'P2': (3, 1), 'P3':
    (4, 4), 'P4': (1, 4)} | Action : MOVE_LEFT | Current node : to_move : P4, board : {'P1':
    (1, 2), 'P2': (3, 1), 'P3': (3, 4), 'P4': (1, 4)} | MINIMAX = {'P1': '?', 'P2': '?', 'P3':
     '?', 'P4': '?'} ]
[Current Player : P4 | Father Node : to_move : P3, board : {'P1': (1, 2), 'P2': (3, 1), 'P3':
    (4, 4), 'P4': (1, 4)} | Action : MOVE_TOP | Current node : to_move : P4, board : {'P1':
    (1, 2), 'P2': (3, 1), 'P3': (4, 3), 'P4': (1, 4)} | MINIMAX = {'P1': '?', 'P2': '?', 'P3':
     '?', 'P4': '?'} ]
[Current Player : P4 | Father Node : to_move : P3, board : {'P1': (1, 2), 'P2': (4, 2), 'P3':
    (4, 4), 'P4': (1, 4)} | Action : MOVE_LEFT | Current node : to_move : P4, board : {'P1':
    (1, 2), 'P2': (4, 2), 'P3': (3, 4), 'P4': (1, 4)} | MINIMAX = {'P1': '?', 'P2': '?', 'P3':
     '?', 'P4': '?'} ]
[Current Player : P4 | Father Node : to_move : P3, board : {'P1': (1, 2), 'P2': (4, 2), 'P3':
    (4, 4), 'P4': (1, 4)} | Action : MOVE_TOP | Current node : to_move : P4, board : {'P1':
    (1, 2), 'P2': (4, 2), 'P3': (4, 3), 'P4': (1, 4)} | MINIMAX = {'P1': '?', 'P2': '?', 'P3':
     '?', 'P4': '?'} ]
```

1.6 Question 4

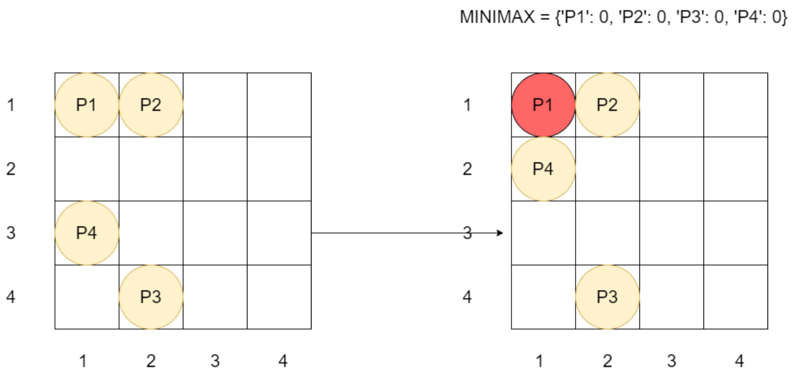
Place the MINIMAX values for each non-repeated game state in the drawing of the game tree based on the output of your code. (15 points)

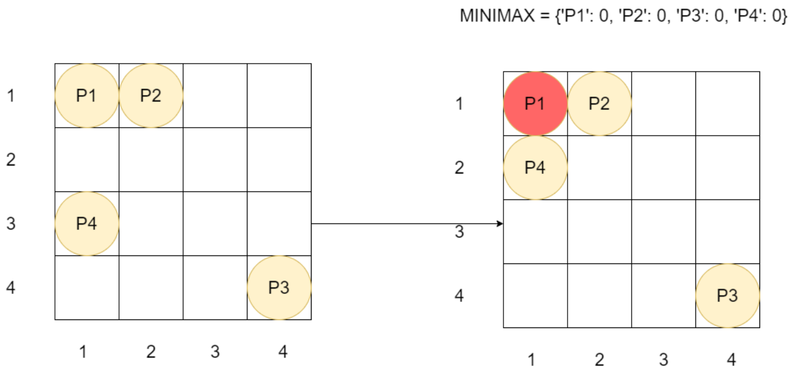
MINIMAX VALUES IN DRAWING [3-LEVELS]

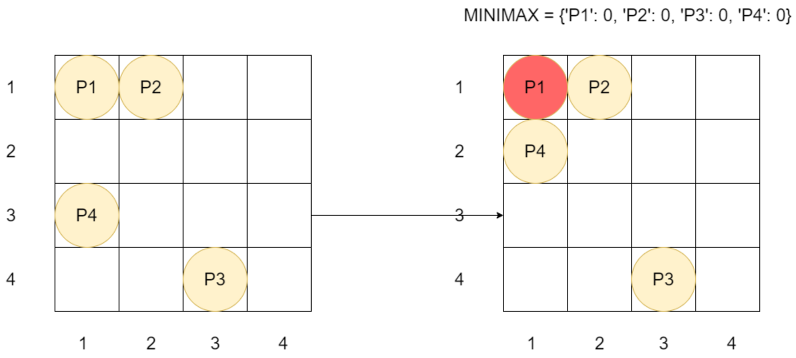

MINIMAX = {'P1': 220, 'P2': 330, 'P3': 440, 'P4': 500}
MOVE_RIGHT
MOVE_BOTTOM
MINIMAX = {'P1': 220, 'P2': 330, 'P3': 440, 'P4': 500}
MINIMAX = {'P1': '?', 'P2': '?', 'P3': '?', 'P4': '?'}
MOVE_LEFT
MOVE_BOTTOM
MINIMAX = {'P1': 220, 'P2': 330, 'P3': 440, 'P4': 500}
MINIMAX = {'P1': 0, 'P2': 0, 'P3': 0, 'P4': 0}
MINIMAX = {'P1': '?', 'P2': '?', 'P3': '?', 'P4': '?'}
MOVE_LEFT
MOVE_BOTTOM
MINIMAX = {'P1': '?', 'P2': '?', 'P3': '?', 'P4': '?'}
MOVE_LEFT
MOVE_TOP
MINIMAX = {'P1': 200, 'P2': 10, 'P3': 30, 'P4': 10}
MINIMAX = {'P1': 0, 'P2': 0, 'P3': 0, 'P4': 0}
MOVE_LEFT
MOVE_TOP
MINIMAX = {'P1': '?', 'P2': '?', 'P3': '?', 'P4': '?'}
MINIMAX = {'P1': '?', 'P2': '?', 'P3': '?', 'P4': '?'}
MOVE_TOP
MOVE_LEFT
MINIMAX = {'P1': '?', 'P2': '?', 'P3': '?', 'P4': '?'}
MINIMAX = {'P1': '?', 'P2': '?', 'P3': '?', 'P4': '?'}
MOVE_LEFT
MOVE_TOP
MINIMAX = {'P1': '?', 'P2': '?', 'P3': '?', 'P4': '?'}
P1
P2
P3
P4
1
2
3
4

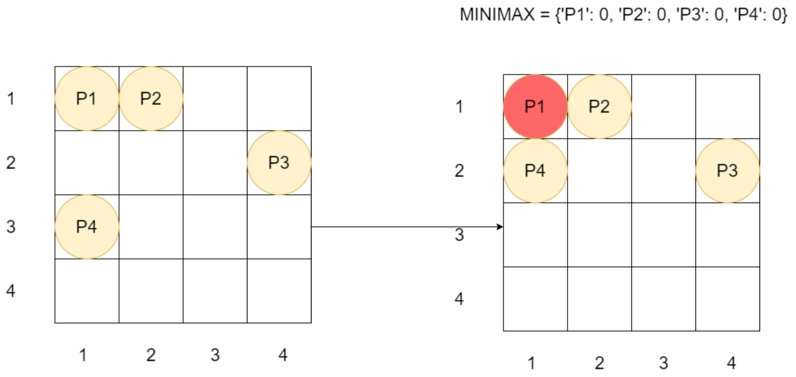
MINIMAX VALUES IN DRAWING [TERMINAL STATES - NO MOVES]

```
[Current Player : P1 | Father Node : to_move : P4, board : {'P1': (1, 1), 'P2': (2, 1), 'P3':
    (2, 4), 'P4': (1, 3)} | Action : MOVE_TOP | Current node : to_move : P1, board : {'P1':
    (1, 1), 'P2': (2, 1), 'P3': (2, 4), 'P4': (1, 2)} | NO MOVES  | MINIMAX = {'P1': 0, 'P2':
    0, 'P3': 0, 'P4': 0} ]
[Current Player : P1 | Father Node : to_move : P4, board : {'P1': (1, 1), 'P2': (2, 1), 'P3':
    (4, 4), 'P4': (1, 3)} | Action : MOVE_TOP | Current node : to_move : P1, board : {'P1':
    (1, 1), 'P2': (2, 1), 'P3': (4, 4), 'P4': (1, 2)} | NO MOVES  | MINIMAX = {'P1': 0, 'P2':
    0, 'P3': 0, 'P4': 0} ]
[Current Player : P1 | Father Node : to_move : P4, board : {'P1': (1, 1), 'P2': (2, 1), 'P3':
    (3, 3), 'P4': (1, 3)} | Action : MOVE_TOP | Current node : to_move : P1, board : {'P1':
    (1, 1), 'P2': (2, 1), 'P3': (3, 3), 'P4': (1, 2)} | NO MOVES  | MINIMAX = {'P1': 0, 'P2':
    0, 'P3': 0, 'P4': 0} ]
[Current Player : P1 | Father Node : to_move : P4, board : {'P1': (1, 1), 'P2': (2, 1), 'P3':
    (4, 2), 'P4': (1, 3)} | Action : MOVE_TOP | Current node : to_move : P1, board : {'P1':
    (1, 1), 'P2': (2, 1), 'P3': (4, 2), 'P4': (1, 2)} | NO MOVES  | MINIMAX = {'P1': 0, 'P2':
    0, 'P3': 0, 'P4': 0} ]
```







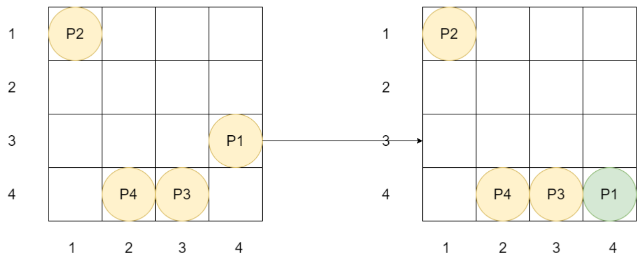


MINIMAX VALUES IN DRAWING [TERMINAL STATES - WINNING STATES]

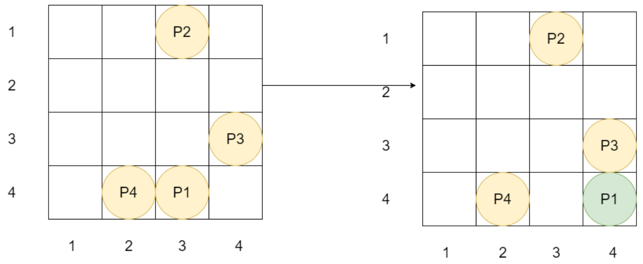
```
"""WINS[P1]"""

[Current Player : P2 | Father Node : to_move : P1, board : {'P1': (4, 3), 'P2': (1, 1), 'P3':
    (3, 4), 'P4': (2, 4)} | Action : MOVE_BOTTOM | Current node : to_move : P2, board : {'P1':
     (4, 4), 'P2': (1, 1), 'P3': (3, 4), 'P4': (2, 4)} | WINS [ P1 | MINIMAX = {'P1': 200, 'P2
    ': 10, 'P3': 30, 'P4': 10} ]
```

```
[Current Player : P2 | Father Node : to_move : P1, board : {'P1': (3, 4), 'P2': (3, 1), 'P3': (4, 3), 'P4': (2, 4)} | Action : MOVE_RIGHT | Current node : to_move : P2, board : {'P1': (4, 4), 'P2': (3, 1), 'P3': (4, 3), 'P4': (2, 4)} | WINS [ P1 | MINIMAX = {'P1': 200, 'P2': 10, 'P3': 30, 'P4': 10} ]
```

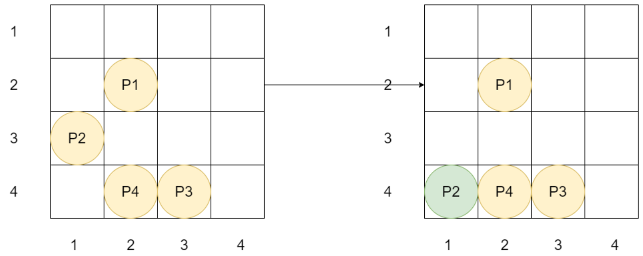
MINIMAX = {'P1': 200, 'P2': 10, 'P3': 30, 'P4': 10}



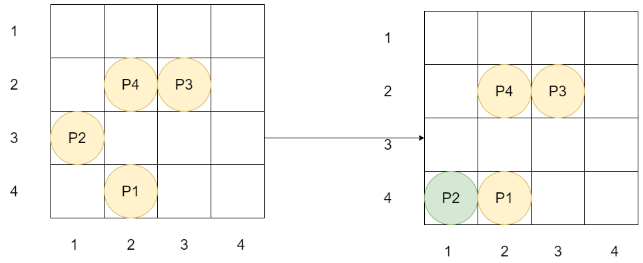
MINIMAX = {'P1': 200, 'P2': 10, 'P3': 30, 'P4': 10}



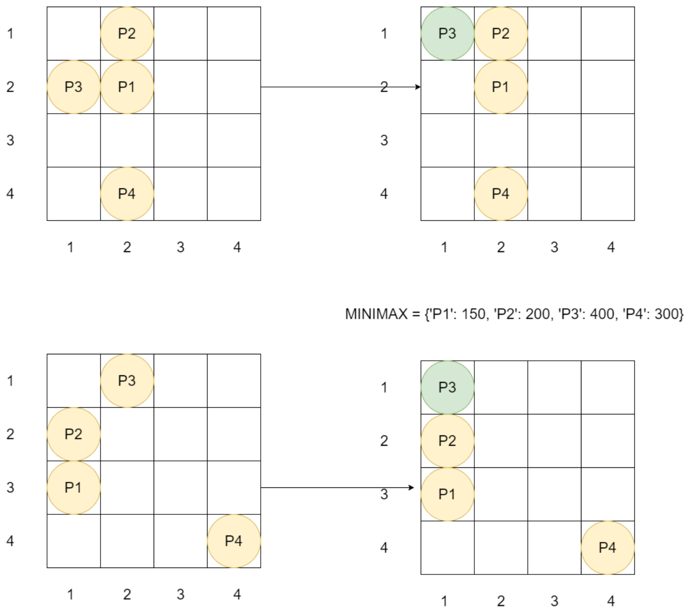
```
"""WINS[P2]"""

[Current Player : P3 | Father Node : to_move : P2, board : {'P1': (2, 2), 'P2': (1, 3), 'P3': (3, 4), 'P4': (2, 4)} | Action : MOVE_BOTTOM | Current node : to_move : P3, board : {'P1': (2, 2), 'P2': (1, 4), 'P3': (3, 4), 'P4': (2, 4)} | WINS [ P2 | MINIMAX = {'P1': 100, 'P2': 300, 'P3': 150, 'P4': 200} ]
[Current Player : P3 | Father Node : to_move : P2, board : {'P1': (2, 4), 'P2': (1, 3), 'P3': (3, 2), 'P4': (2, 2)} | Action : MOVE_BOTTOM | Current node : to_move : P3, board : {'P1': (2, 4), 'P2': (1, 4), 'P3': (3, 2), 'P4': (2, 2)} | WINS [ P2 | MINIMAX = {'P1': 100, 'P2': 300, 'P3': 150, 'P4': 200} ]
```



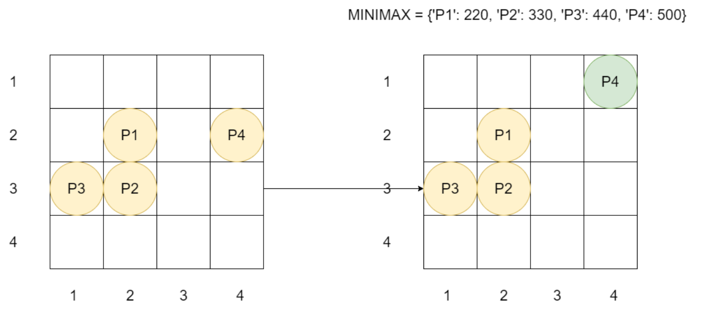
MINIMAX = {'P1': 100, 'P2': 300, 'P3': 150, 'P4': 200}

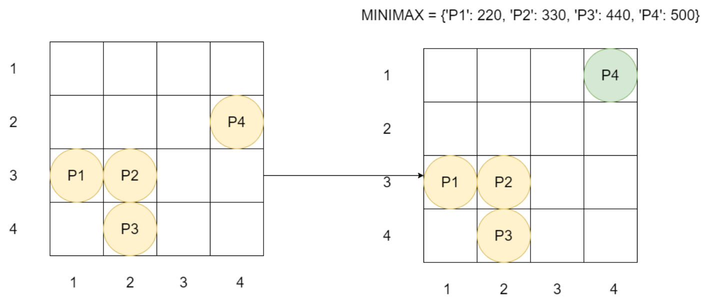


```
"""WINS[P3]"""

[Current Player : P4 | Father Node : to_move : P3, board : {'P1': (2, 2), 'P2': (2, 1), 'P3':
    (1, 2), 'P4': (2, 4)} | Action : MOVE_TOP | Current node : to_move : P4, board : {'P1':
    (2, 2), 'P2': (2, 1), 'P3': (1, 1), 'P4': (2, 4)} | WINS [ P3 | MINIMAX = {'P1': 150, 'P2'
    : 200, 'P3': 400, 'P4': 300} ]
[Current Player : P4 | Father Node : to_move : P3, board : {'P1': (1, 3), 'P2': (1, 2), 'P3':
    (2, 1), 'P4': (4, 4)} | Action : MOVE_LEFT | Current node : to_move : P4, board : {'P1':
    (1, 3), 'P2': (1, 2), 'P3': (1, 1), 'P4': (4, 4)} | WINS [ P3 | MINIMAX = {'P1': 150, 'P2'
    : 200, 'P3': 400, 'P4': 300} ]
```



```
"""WINS[P4]"""

[Current Player : P1 | Father Node : to_move : P4, board : {'P1': (2, 2), 'P2': (2, 1), 'P3':
    (1, 3), 'P4': (4, 2)} | Action : MOVE_TOP | Current node : to_move : P1, board : {'P1':
    (2, 2), 'P2': (2, 1), 'P3': (1, 3), 'P4': (4, 1)} | WINS [ P4 | MINIMAX = {'P1': 220, 'P2'
    : 330, 'P3': 440, 'P4': 500} ]
[Current Player : P1 | Father Node : to_move : P4, board : {'P1': (1, 3), 'P2': (2, 3), 'P3':
    (2, 4), 'P4': (4, 2)} | Action : MOVE_TOP | Current node : to_move : P1, board : {'P1':
    (1, 3), 'P2': (2, 3), 'P3': (2, 4), 'P4': (4, 1)} | WINS [ P4 | MINIMAX = {'P1': 220, 'P2'
    : 330, 'P3': 440, 'P4': 500} ]
```





Complete output is part of file game_tree_minimax.txt, included as part of submission, It can also be generated by running the file P1.py

1.7 EXTRA CREDIT

C. If you allow an additional action for the game, namely that any player can jump over an occupied square, and represent this new action as : Jump+Down, Jump+Left, Jump+Right or Jump+Down, modify your program to generate the new game tree. Produce the output in the same format as when doing assignment A. (10 points) Comment on the difference between the game trees: Which player won faster in which variant of the game??? Are the repeated states any different? (10 points)

SOLUTION

Identifying unique game state remains the same with following new **actions/moves** available for each player

Actions	MOVE_LEFT	MOVE_RIGHT	MOVE_TOP	MOVE_BOTTOM
	JUMP_LEFT	JUMP_RIGHT	JUMP_TOP	JUMP_BOTTOM

Actions based on JUMP are only available if

1. The corresponding MOVE action is blocked by another player.
2. The position to which it jumps is vacant.

CHANGES IN CODE OF GAME REPRESENTATION

```
    def compute_moves(self, to_move, board):
        players = ['P1', 'P2', 'P3', 'P4']
        x, y = board[to_move]
        blocked_positions = [board[other_player] for other_player in players if other_player is not to_move]
        actions = ['MOVE_LEFT', 'MOVE_RIGHT', 'MOVE_TOP', 'MOVE_BOTTOM']
        # Imposing boundary restrictions
        if x >= 4:
            actions.remove('MOVE_RIGHT')
        elif x <= 1:
            actions.remove('MOVE_LEFT')

        if y >= 4:
            actions.remove('MOVE_BOTTOM')
        elif y <= 1:
            actions.remove('MOVE_TOP')

        # Imposing blocking restrictions by other players
        possible_actions = actions.copy()
        for action in possible_actions:
            xn = x
            yn = y
            if action == 'MOVE_LEFT':
                xn = x - 1
            elif action == 'MOVE_RIGHT':
                xn = x + 1
            elif action == 'MOVE_TOP':
                yn = y - 1
            elif action == 'MOVE_BOTTOM':
                yn = y + 1
            if (xn, yn) in blocked_positions:
                actions.remove(action)
                if action == 'MOVE_LEFT':
                    if x > 2 and (x - 2, y) not in blocked_positions:
                        actions.append('JUMP_LEFT')
                elif action == 'MOVE_RIGHT':
                    if x <= 2 and (x + 2, y) not in blocked_positions:
                        actions.append('JUMP_RIGHT')
                elif action == 'MOVE_TOP':
                    if y > 2 and (x, y - 2) not in blocked_positions:
                        actions.append('JUMP_TOP')
                elif action == 'MOVE_BOTTOM':
                    if y <= 2 and (x, y + 2) not in blocked_positions:
                        actions.append('JUMP_BOTTOM')
        return actions

    @staticmethod
    def new_board_config(state, move):
        current_player = state.to_move
        current_board_config = state.board.copy()
        x, y = current_board_config[current_player]
        xn = x
        yn = y
        if move == 'MOVE_LEFT':
            xn = x - 1
        elif move == 'JUMP_LEFT':
            xn = x - 2
        elif move == 'MOVE_RIGHT':
            xn = x + 1
        elif move == 'JUMP_RIGHT':
            xn = x + 1
        elif move == 'MOVE_TOP':
            yn = y - 1
        elif move == 'JUMP_TOP':
            yn = y - 2
        elif move == 'JUMP_BOTTOM':
            yn = y + 2
        elif move == 'MOVE_BOTTOM':
            yn = y + 1
        current_board_config[current_player] = (xn, yn)
        return current_board_config
```

Observations

1. Here as players can jump when they are blocked, there are no **TERMINAL STATES** with no moves
2. Total possible state space for the problem according my representation $16 * 15 * 14 * 13 * 4 = 174720$ of which only around 20k are explored without JUMP moves
3. But, with JUMP moves the total nodes reached increased substantially witnessed in [Runtime : more than 4hrs in Google Colab]
4. Runtime can be cut down by using Evaluation function corresponding to a state

Questions

Which player won faster in which variant of the game???

1. In this implementation all the players reach their WINNING state at lower depth compared to earlier. For P1 - Depth : 21 vs 14
2. MINIMAX value for initial state does not change, it still corresponds to the P4 WINNING state

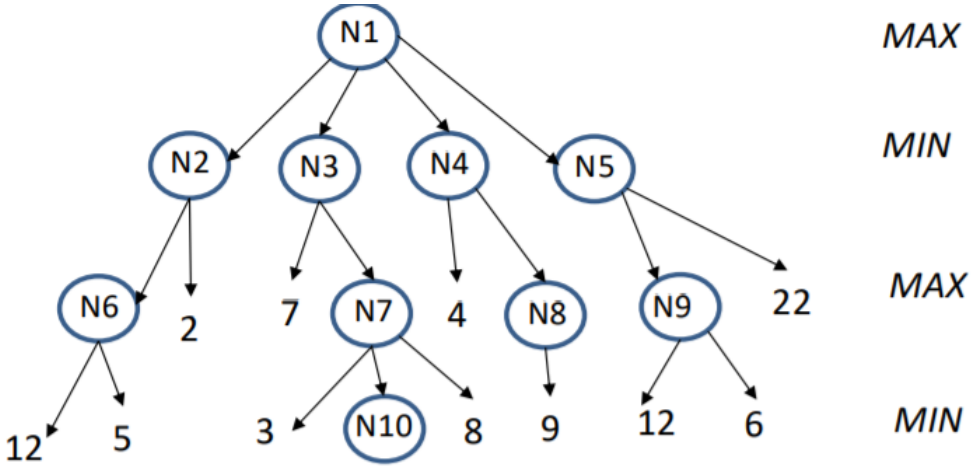
Are the repeated states any different?

1. As the game reaches more state, the number of repeated states also increase substantially.

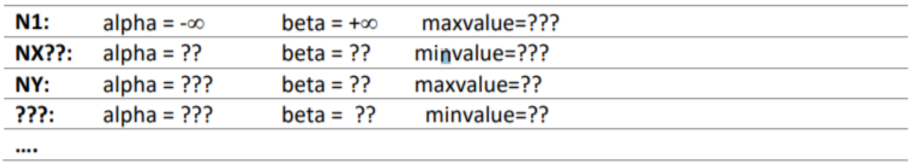
Output file for this part is 60MB in size. So I have included a link to my drive which contains the file game_tree_adv.txt. It can also be generated by running P1.py.

2 ALPHA BETA SEARCH

Find the move ordering (killer move) that allows you to prune the largest number of subtrees when using alpha-beta pruning on the following game tree:



You will receive 10 points if you find the killer move and explain why it is a killer move. You should produce a trace of alpha-beta pruning using the format showing how the killer move operates (10 points):



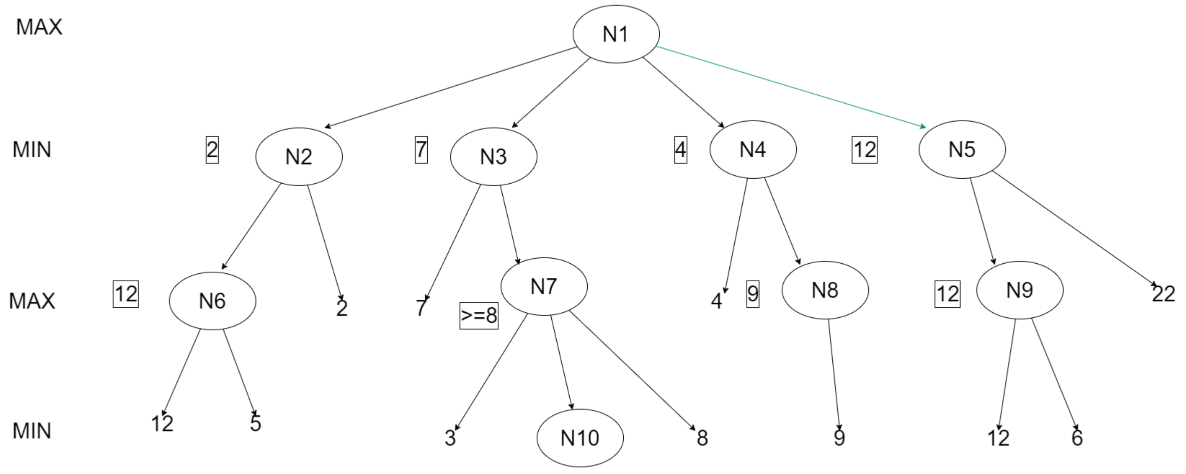
N1:	alpha = -∞	beta = +∞	maxvalue=???
NX??:	alpha = ??	beta = ??	minvalue=???
NY:	alpha = ???	beta = ??	maxvalue=??
???:	alpha = ???	beta = ??	minvalue=??
....			

Indicate on the Search Tree which subtrees shall be pruned and if it is alpha or beta pruning (6 points) when you indicate correctly which subtrees shall be pruned.

SOLUTION

Killer moves involve examining first the successors that are best

In order to find the best successor for the problem let us consider estimates for minimax values for successors of the max node N1



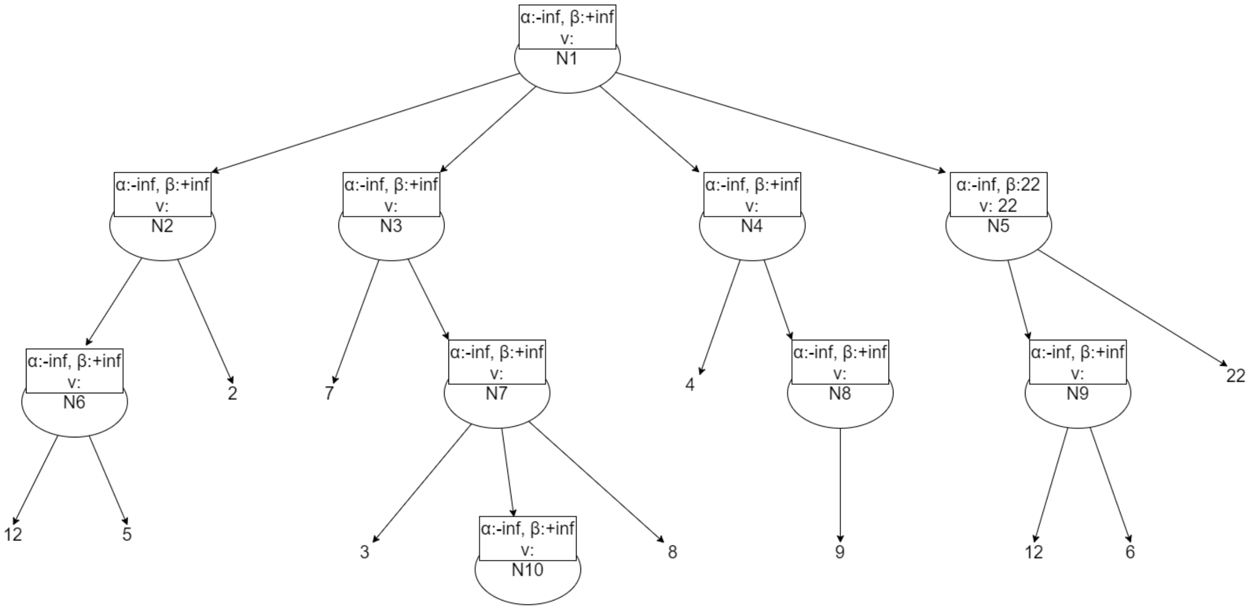
On observation we find that move to N5 is the best one. In order to verify why it is a killer move, let us examine how many nodes are not inspected in the process.

$\alpha - \beta$ *Pruning*

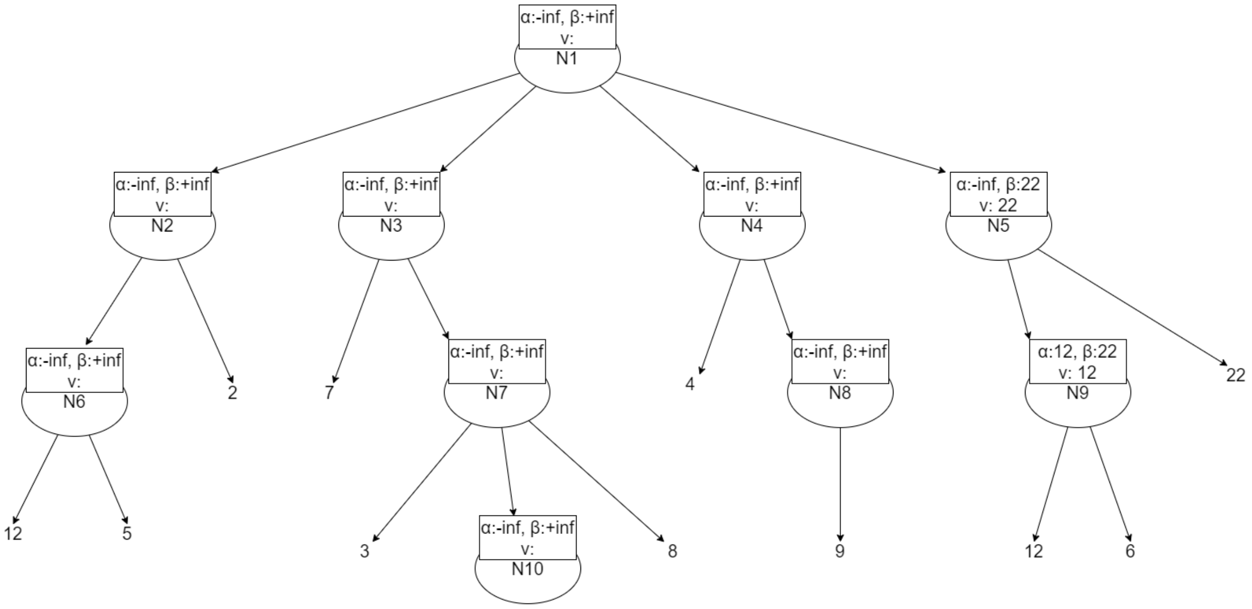
Node	α	β	value
N1	-∞	∞	maxvalue = -∞
N5	-∞	∞	minvalue = ∞
22 return 22			
N5	-∞	22	minvalue = 22
N9	-∞	22	maxvalue = -∞
12 return 12			
N9	12	22	maxvalue = 12
6 return 6			
N9	12	22	maxvalue = 12
N5	-∞	12	minvalue = 12
N1	12	∞	maxvalue = 12
N4	12	∞	minvalue = ∞
4 return 4			
N4	12	∞	minvalue = 4 (ALPHA CUT)
N1	12	∞	maxvalue = 12
N3	12	∞	minvalue = ∞
7 return 7			
N3	12	∞	minvalue = 7 (ALPHA CUT)
N1	12	∞	maxvalue = 12
N2	12	∞	minvalue = ∞
2 return 2			
N2	12	∞	minvalue = 2 (ALPHA CUT)
N1	12	∞	maxvalue = 12

Following is the pictorial trace of the pruning with killer move :

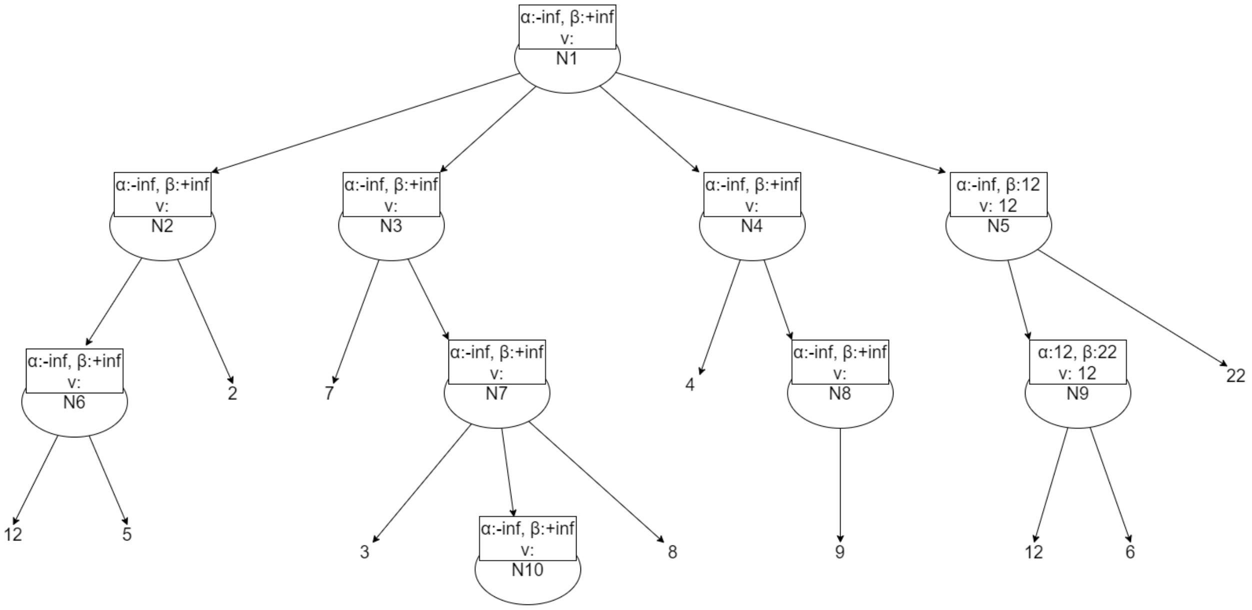
STEP - 1



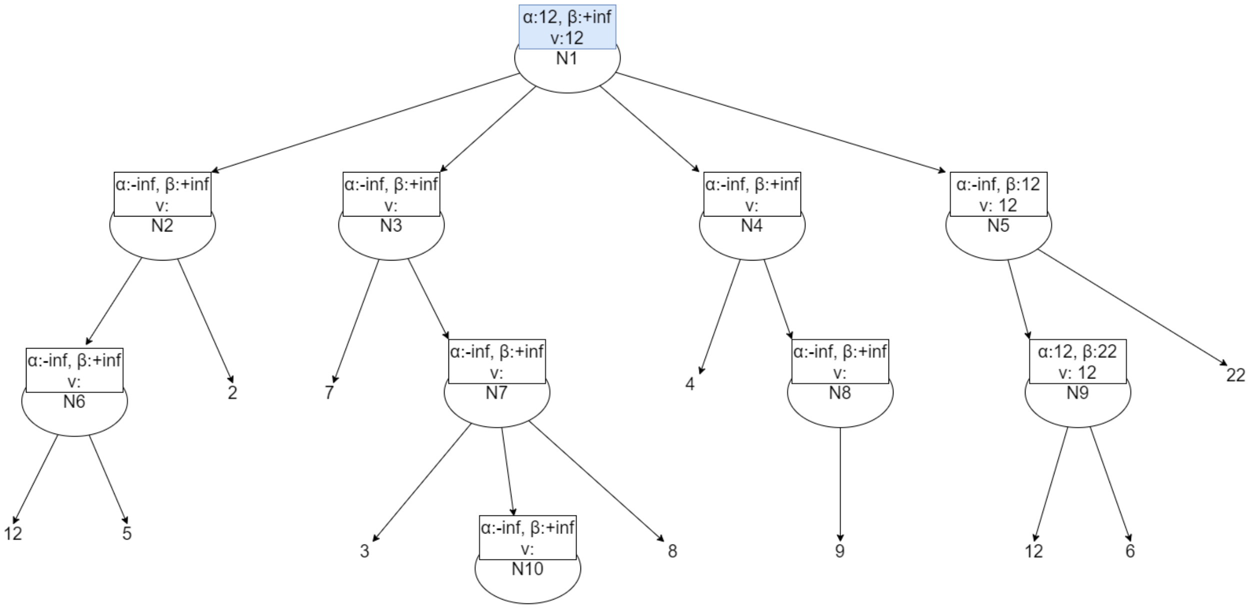
STEP - 2


α:-inf, β:+inf
v:
N1
α:-inf, β:+inf
v:
N2
α:-inf, β:+inf
v:
N3
α:-inf, β:+inf
v:
N4
α:-inf, β:22
v: 22
N5
α:-inf, β:+inf
v:
N6
2
7
α:-inf, β:+inf
v:
N7
4
α:-inf, β:+inf
v:
N8
α:12, β:22
v: 12
N9
22
12
5
3
α:-inf, β:+inf
v:
N10
8
9
12
6

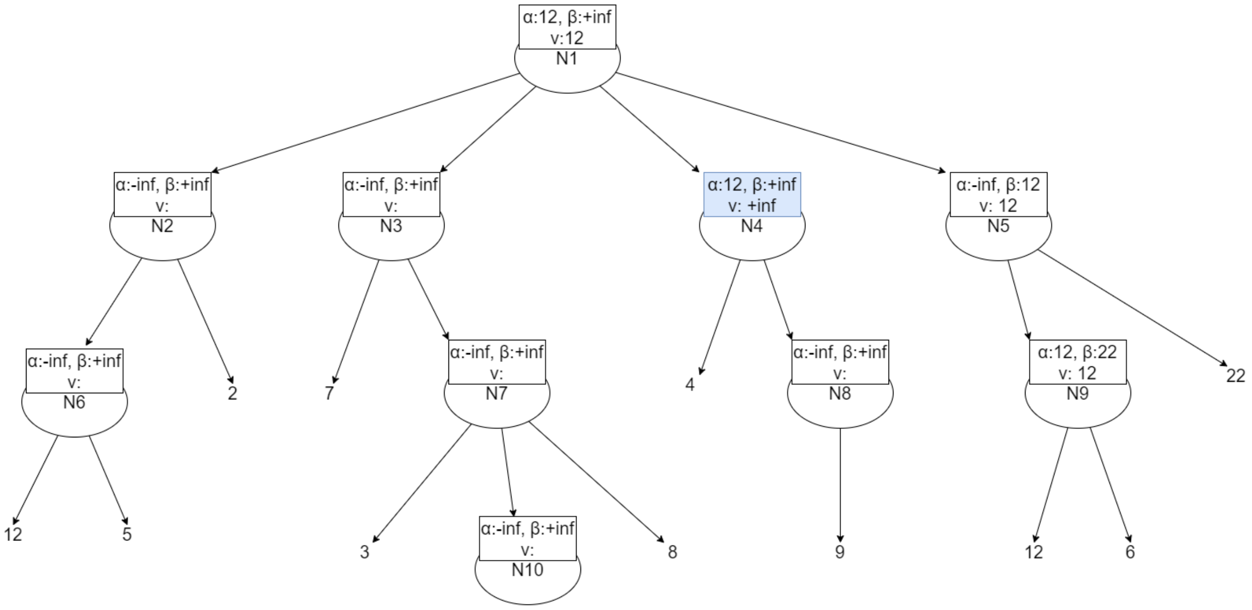
STEP - 3


α:-inf, β:+inf
v:
N1
α:-inf, β:+inf
v:
N2
α:-inf, β:+inf
v:
N3
α:-inf, β:+inf
v:
N4
α:-inf, β:12
v: 12
N5
α:-inf, β:+inf
v:
N6
2
7
α:-inf, β:+inf
v:
N7
4
α:-inf, β:+inf
v:
N8
α:12, β:22
v: 12
N9
22
12
5
3
α:-inf, β:+inf
v:
N10
8
9
12
6

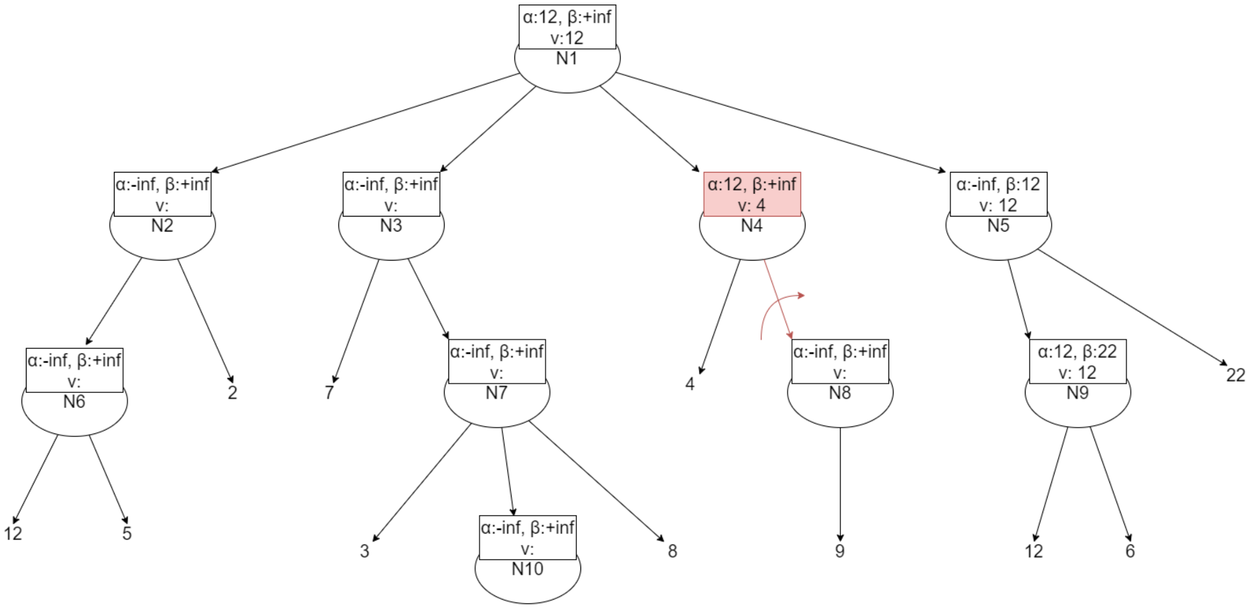
STEP - 4


α:12, β:+inf
v:12
N1
α:-inf, β:+inf
v:
N2
α:-inf, β:+inf
v:
N3
α:-inf, β:+inf
v:
N4
α:-inf, β:12
v: 12
N5
α:-inf, β:+inf
v:
N6
2
7
α:-inf, β:+inf
v:
N7
4
α:-inf, β:+inf
v:
N8
α:12, β:22
v: 12
N9
22
12
5
3
α:-inf, β:+inf
v:
N10
8
9
12
6

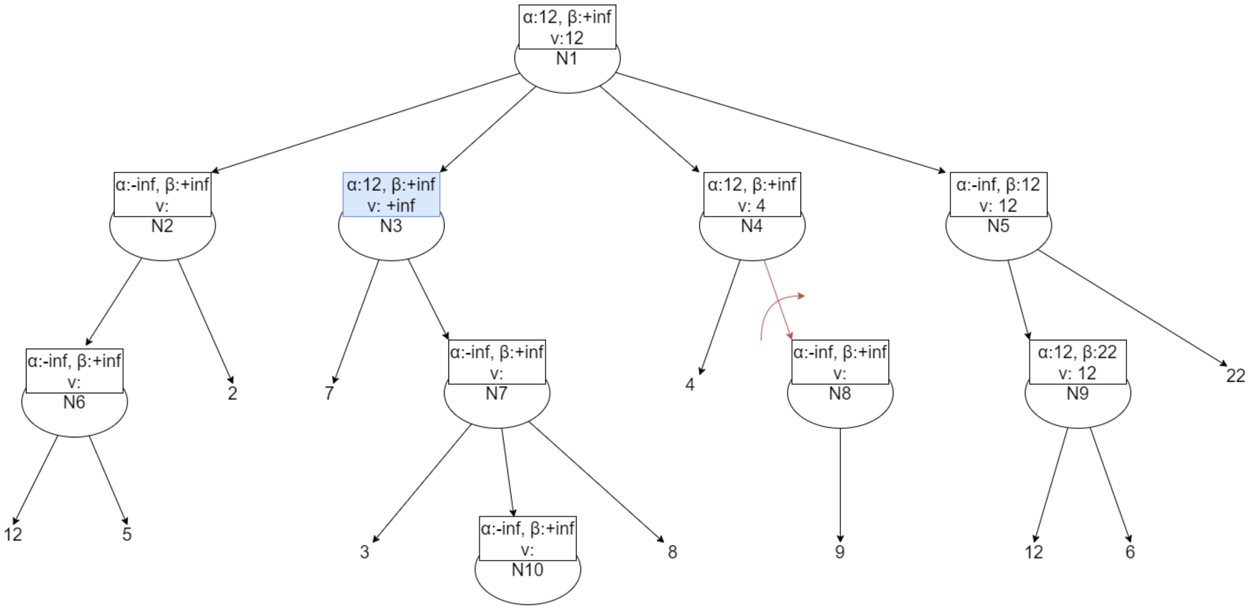
STEP - 5


α:12, β:+inf
v:12
N1
α:-inf, β:+inf
v:
N2
α:-inf, β:+inf
v:
N3
α:12, β:+inf
v: +inf
N4
α:-inf, β:12
v: 12
N5
α:-inf, β:+inf
v:
N6
2
7
α:-inf, β:+inf
v:
N7
4
α:-inf, β:+inf
v:
N8
α:12, β:22
v: 12
N9
22
12
5
3
α:-inf, β:+inf
v:
N10
8
9
12
6

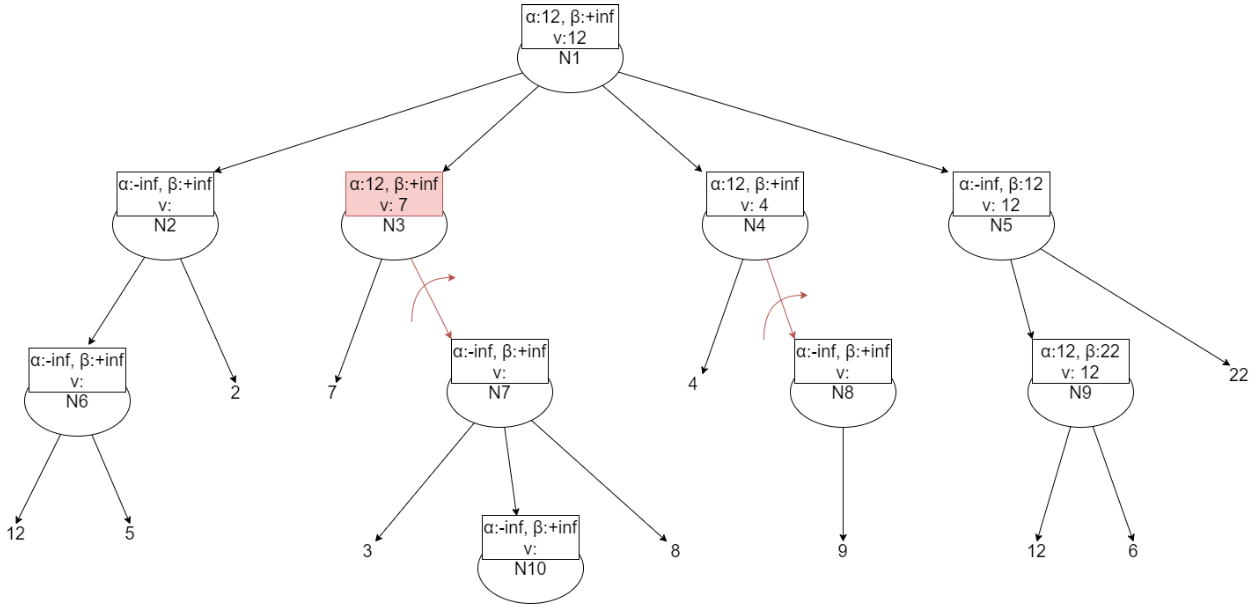
STEP - 6


α:12, β:+inf
v:12
N1
α:-inf, β:+inf
v:
N2
α:-inf, β:+inf
v:
N3
α:12, β:+inf
v: 4
N4
α:-inf, β:12
v: 12
N5
α:-inf, β:+inf
v:
N6
2
7
α:-inf, β:+inf
v:
N7
4
α:-inf, β:+inf
v:
N8
α:12, β:22
v: 12
N9
22
12
5
3
α:-inf, β:+inf
v:
N10
8
9
12
6

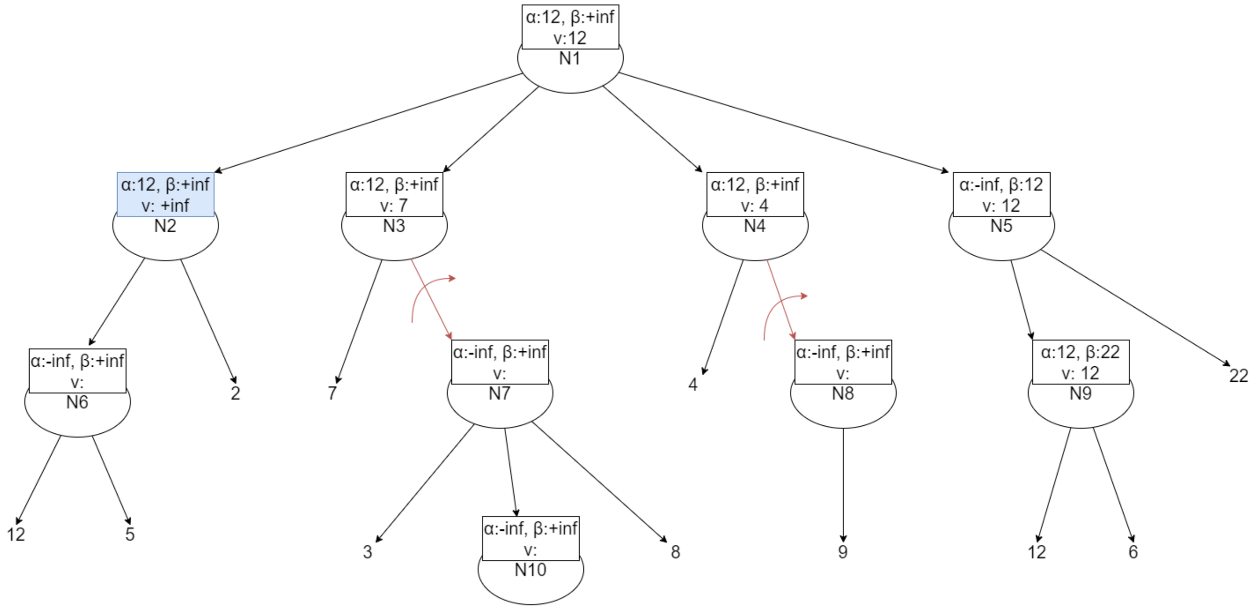
STEP - 7


α:12, β:+inf
v:12
N1
α:-inf, β:+inf
v:
N2
α:12, β:+inf
v: +inf
N3
α:12, β:+inf
v: 4
N4
α:-inf, β:12
v: 12
N5
α:-inf, β:+inf
v:
N6
2
7
α:-inf, β:+inf
v:
N7
4
α:-inf, β:+inf
v:
N8
α:12, β:22
v: 12
N9
22
12
5
3
α:-inf, β:+inf
v:
N10
8
9
12
6

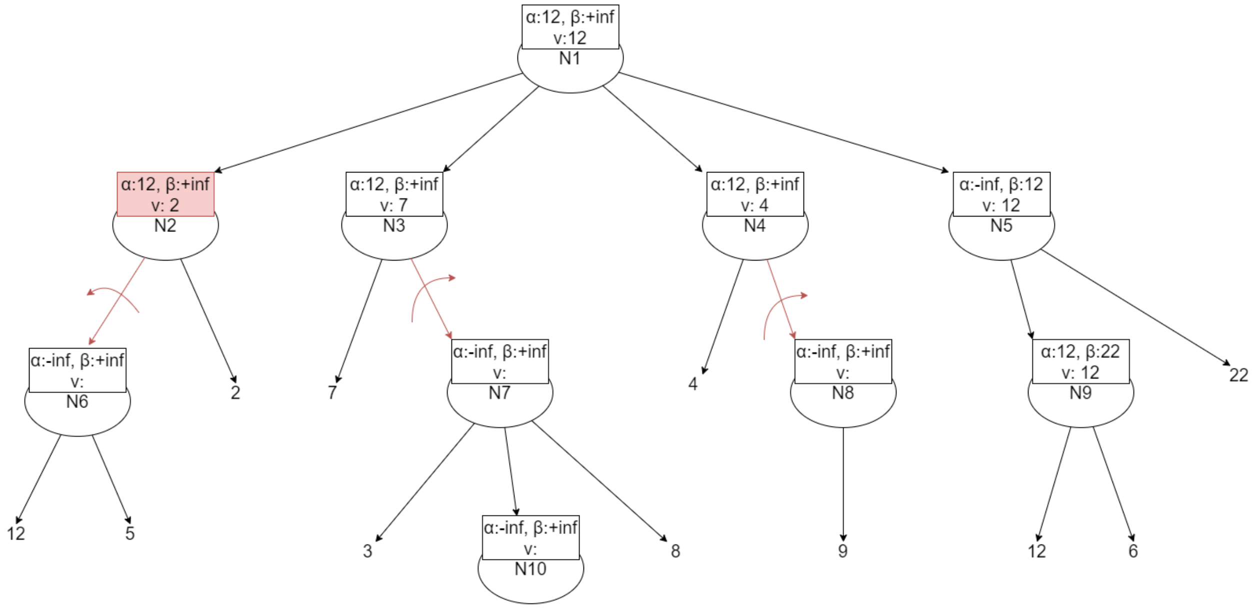
STEP - 8



STEP - 9



STEP - 10

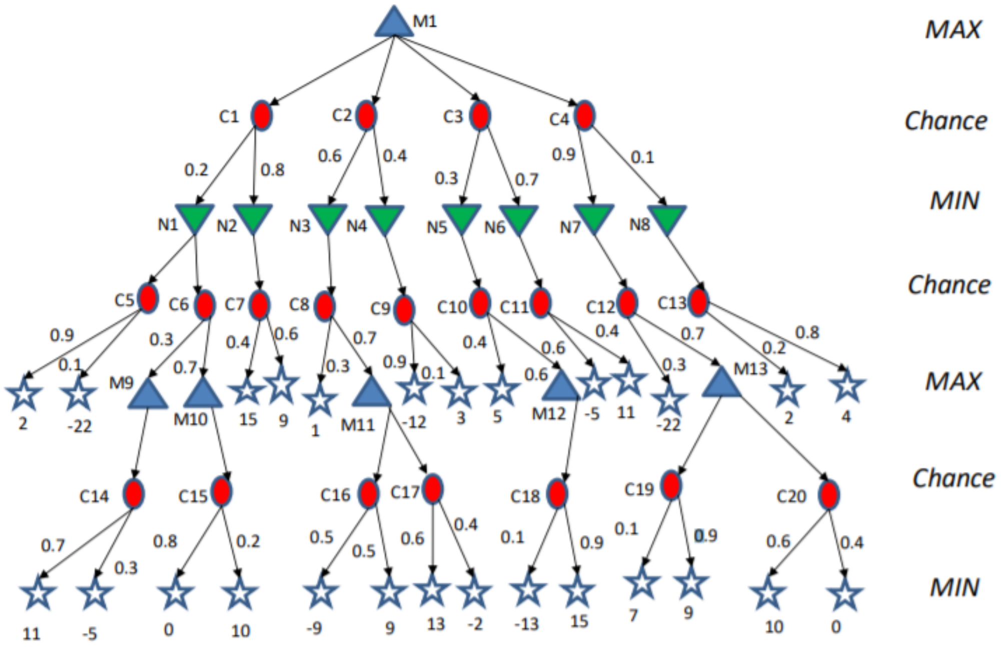


Observations

1. It is a killer move as it does not visit the nodes N6, N7, N8, N10 and terminal states of 12, 5, 3, 8, 9.
2. Any other move shall visit some of the other nodes.
3. The tree involves three prunings and all of them occur at MINVALUE node so all are ALPHA CUTS

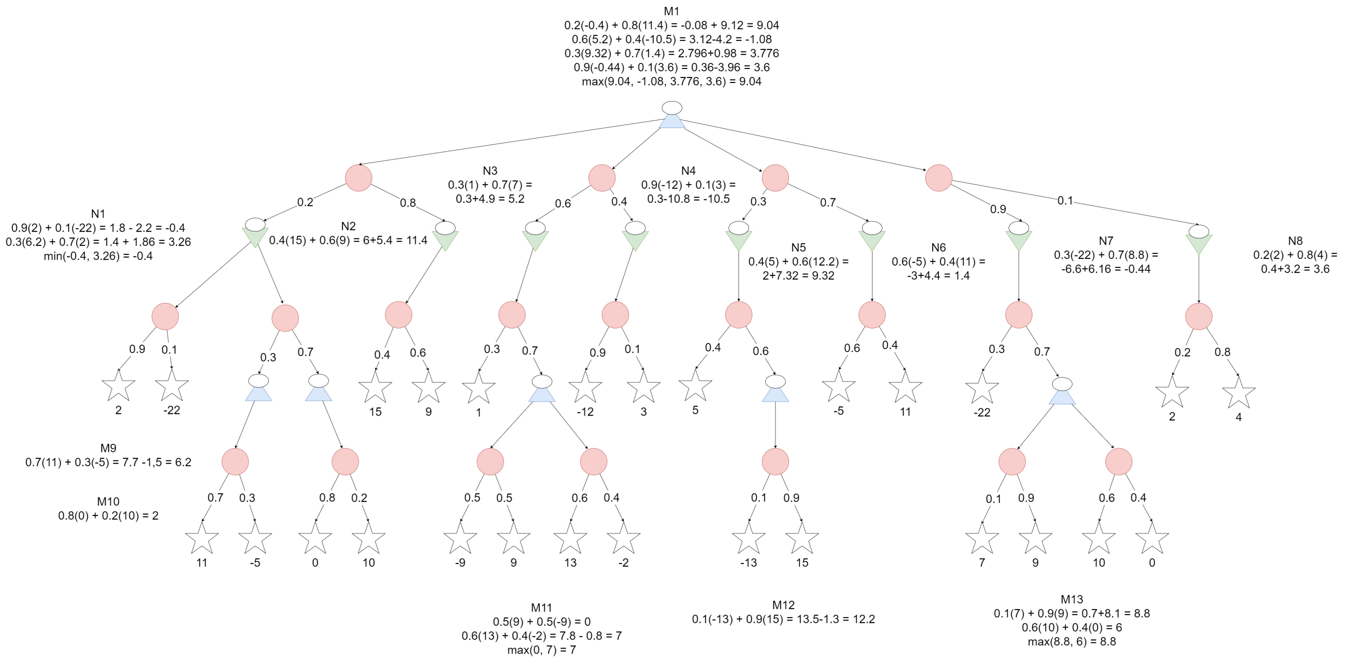
3 CHANCE GAMES

Given the following game tree:



Compute the Expectiminimax value in the following nodes, making sure you show how you computed the value: M1, M9, M10, M11, M12, M13 , N1 , N2 , N3 , N4 , N5 , N6 , N7 , N8

SOLUTION



M1
0.2(-0.4) + 0.8(11.4) = -0.08 + 9.12 = 9.04
0.6(5.2) + 0.4(-10.5) = 3.12-4.2 = -1.08
0.3(9.32) + 0.7(1.4) = 2.796+0.98 = 3.776
0.9(-0.44) + 0.1(3.6) = 0.36-3.96 = 3.6
max(9.04, -1.08, 3.776, 3.6) = 9.04

N1
0.9(2) + 0.1(-22) = 1.8 - 2.2 = -0.4
0.3(6.2) + 0.7(2) = 1.4 + 1.86 = 3.26
min(-0.4, 3.26) = -0.4

N2
0.4(15) + 0.6(9) = 6+5.4 = 11.4

N3
0.3(1) + 0.7(7) =
0.3+4.9 = 5.2

N4
0.9(-12) + 0.1(3) =
0.3-10.8 = -10.5

N5
0.4(5) + 0.6(12.2) =
2+7.32 = 9.32

N6
0.6(-5) + 0.4(11) =
-3+4.4 = 1.4

N7
0.3(-22) + 0.7(8.8) =
-6.6+6.16 = -0.44

N8
0.2(2) + 0.8(4) =
0.4+3.2 = 3.6

M9
0.7(11) + 0.3(-5) = 7.7 -1,5 = 6.2

M10
0.8(0) + 0.2(10) = 2

M11
0.5(9) + 0.5(-9) = 0
0.6(13) + 0.4(-2) = 7.8 - 0.8 = 7
max(0, 7) = 7

M12
0.1(-13) + 0.9(15) = 13.5-1.3 = 12.2

M13
0.1(7) + 0.9(9) = 0.7+8.1 = 8.8
0.6(10) + 0.4(0) = 6
max(8.8, 6) = 8.8

Following are the Expectiminimax values for each of the nodes

1. In M1: 9.04
2. In M9: 6.2
3. In M10: 2
4. In M11: 7
5. In M12: 12.2
6. In M13: 8.8
7. In N1: -0.4
8. In N2: 11.4
9. In N3: 5.2
10. In N4: -10.5
11. In N5: 9.32
12. In N6: 1.4
13. In N7: -0.44
14. In N8: 3.6

First picture also has calculations embedded in it.