IQRA University

CSC 471 Artificial Intelligence

Lab# 05
Adversarial Search

Objective:

The experiment provides introduction to adversarial search schemes and uses MiniMax algorithm to plan in such scenarios.

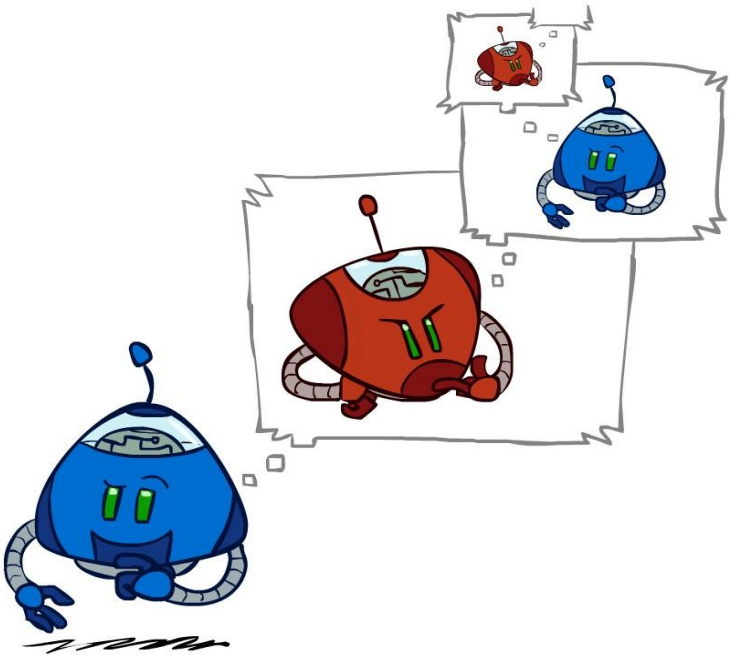
Name of Student: FARAZ ALAM

Roll No: 13948 **Sec.** BS-CS

Date of Experiment: 18-11-2023

Lab 05: Adversarial Search

Multi-agent environments, in which each agent needs to consider the actions of other agents and how they affect its own welfare. The unpredictability of these other agents can introduce contingencies into the agent's problem-solving process. If the agents' goals are in conflict it gives rise to adversarial search problems where the adversary is planning against the agent.

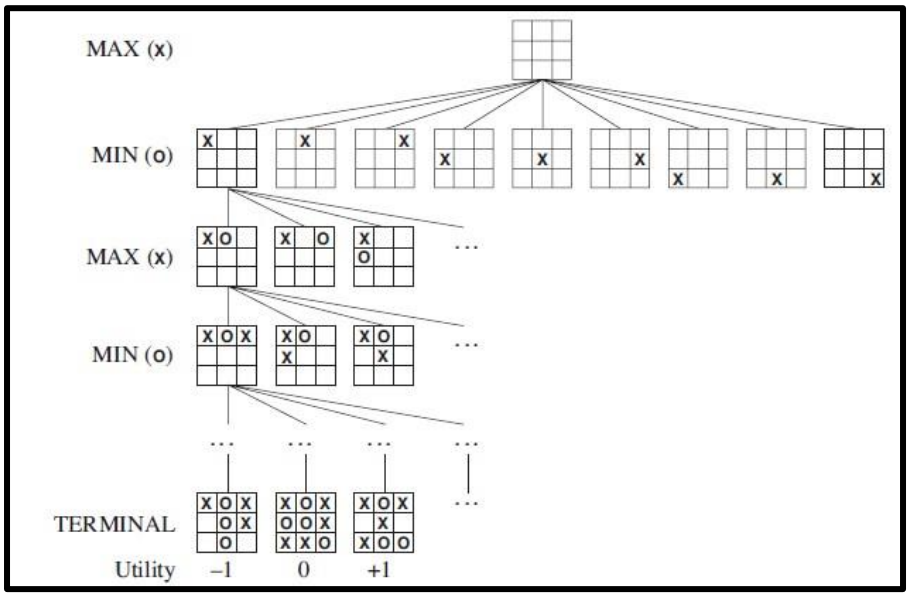


In AI, the most common games are of a rather specialized kind—what game theorists call deterministic, turn-taking, two-player, zero-sum games of perfect information (such as chess). In our terminology, this means deterministic, fully observable environments in which two agents act alternately and in which the utility values at the end of the game are always equal and opposite. For example, if one player wins a game of chess, the other player necessarily loses. It is this opposition between the agents' utility functions that makes the situation adversarial.

Minimax Search

The minimax search is especially known for its usefulness in calculating the best move in two player games where all the information is available, such as chess or tic tac toe. It consists of navigating through a tree which captures all the possible moves in the game, where each move is represented in terms of loss and gain for one of the players. It follows that this can only be used to make decisions in zero-sum games, where one player's loss is the other player's gain. Theoretically, this search algorithm is based on von Neumann's

minimax theorem which states that in these types of games there is always a set of strategies which leads to both players gaining the same value and that seeing as this is the best possible value one can expect to gain, one should employ this set of strategies.



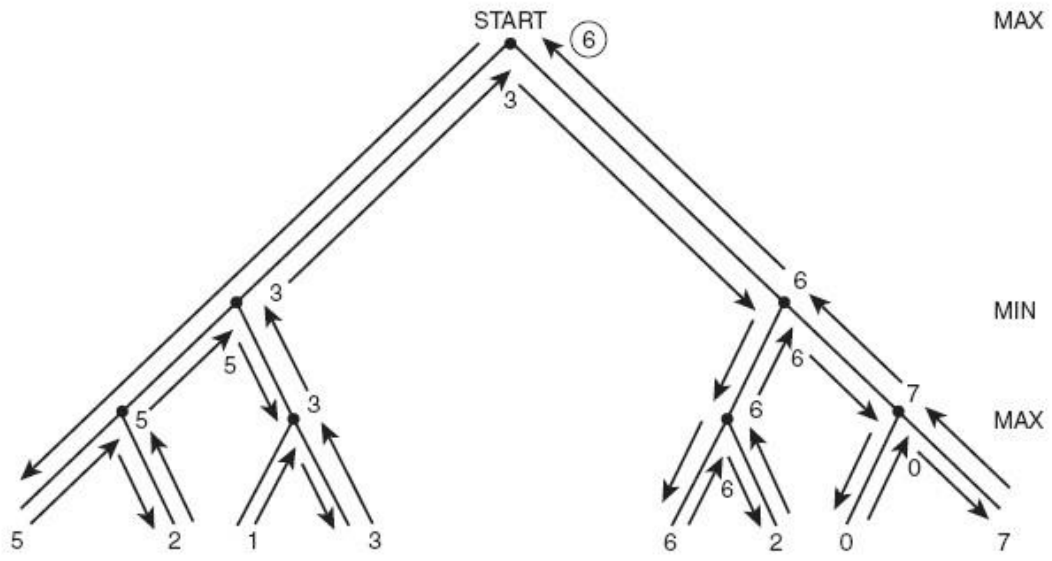
Minimax Algorithm

The minimax algorithm computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are backed up through the tree as the recursion unwinds. If the maximum depth of the tree is m and there are b legal moves at each point, then the time complexity of the minimax algorithm is $O(b^m)$. The space complexity is O(bm) for an algorithm that generates all actions at once, or O(m) for an algorithm that generates actions one at a time. For real games, of course, the time cost is totally impractical, but this algorithm serves as the basis for the mathematical analysis of games and for more practical algorithms.

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

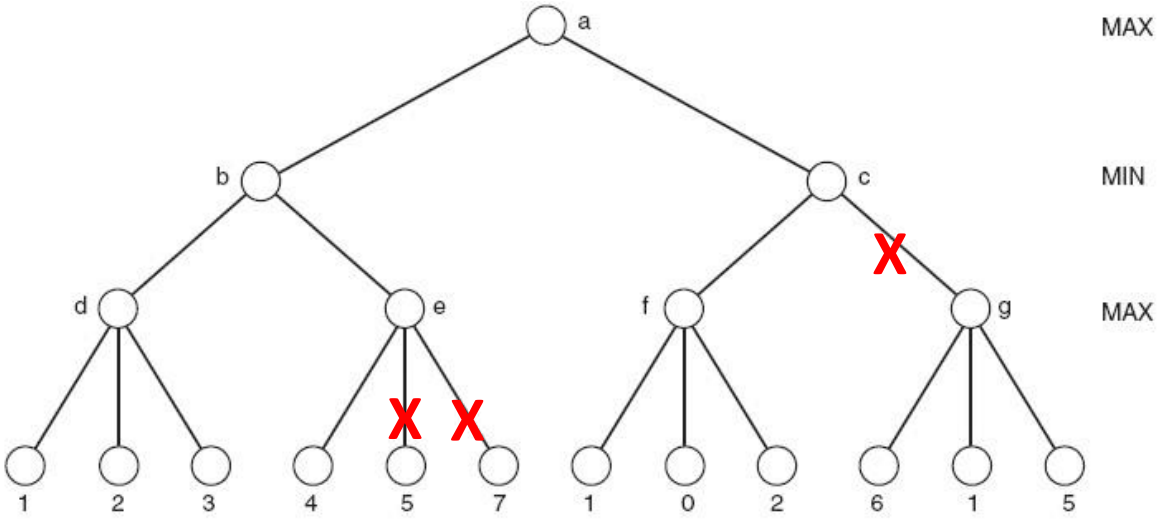
```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```



Alpha Beta Pruning:

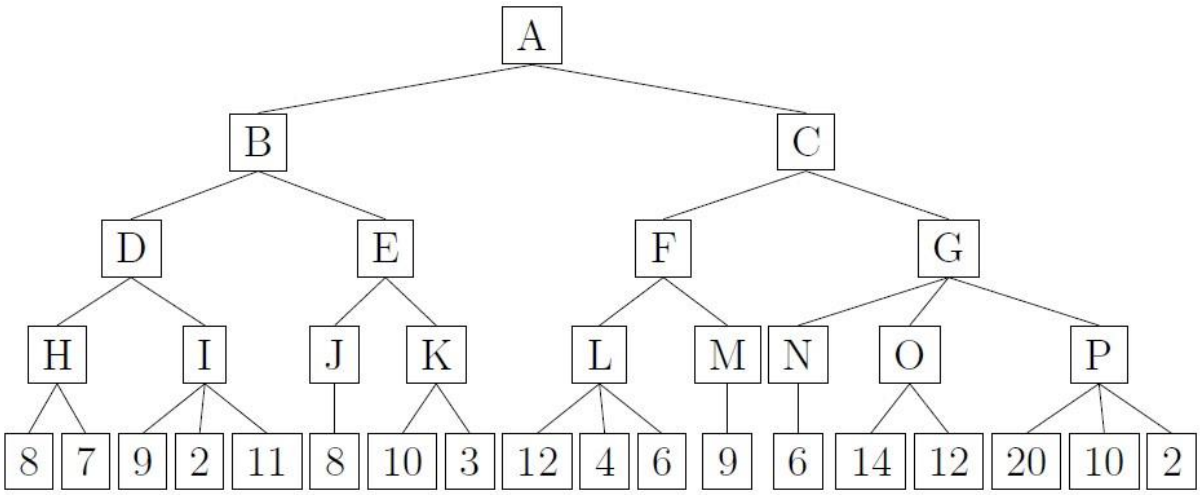
Using alpha–beta pruning, it is possible to remove sections of the game tree that are not worth examining, to make searching for a good move more efficient. The principle behind alpha–beta pruning is that if a move is determined to be worse than another move that has already been examined, then further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



Student Exercise

Task 1

Implement the minimax search algorithm for the following graph where all the leaf nodes are provided with utility values. Starting from node A with max's turn, provide both the cost of the selected solution and the path for it.



In [2]:

```python
def evaluate(leaf_node, max_turn):
    # Return the values of terminal nodes
    if max_turn:
        return max(leaf_node.values())
    return min(leaf_node.values())
```

In [3]:

```python
def minimax(node, depth, max_turn, path=[]):
    if depth == 0 or not node:  # Base case: Reached maximum depth or terminal node
        return evaluate(node, max_turn), path + [node]

    if max_turn:
        # Maximizer's turn
        max_eval = float('-inf')
        best_path = []
        for child in node:
            eval_child, child_path = minimax(graph[child], depth - 1, False, path + [child])
            if eval_child > max_eval:
                max_eval = eval_child
                best_path = child_path
        return max_eval, best_path
    else:
        # Minimizer's turn
        min_eval = float('inf')
        best_path = []
        for child in node:
            eval_child, child_path = minimax(graph[child], depth - 1, True, path + [child])
            if eval_child < min_eval:
                min_eval = eval_child
                best_path = child_path
        return min_eval, best_path
```

In [4]:

```python
# Initial call for the root node 'A'

result, optimal_path = minimax(graph['A'], depth=3, max_turn=True)

print("Optimal value for the Max player:", result)
print("Optimal path  for the Max player:", optimal_path)
```

```
Optimal value for the Max player: 9
Optimal path  for the Max player: ['C', 'F', 'M', {12: 9}]
```