**National University of Computer & Emerging Sciences, Karachi Computer Science Department**

**Summer 2023, Lab Manual – 06**

|  |  |
| --- | --- |
| **Course Code: AI-2002** | **Course: Artificial Intelligence Lab** |
| **Instructor(s):** | **Sania Urooj** |

**Content:**

* Adversarial Search
* Elements of game Playing Search
* Types of Algorithms in Adversarial Search
* MinMax Algorithm
* Alpha-Beta Pruning(Conditions and working) 4.Constraint Satisfaction Problems
* Solving Constraint Satisfaction Problem.

**Objective:**

* An Introduction to Game theory and Adversarial Search Algorithm.
* Constraint Satisfaction Problem in AI with multiple examples

**Introduction to Game Theory:**

Game Theory is a branch of mathematics used to model the strategic interaction between diﬀerent players in a context with predeﬁned rules and outcomes. According to game theory, a game is played between two players. To complete the game, one has to win the game, and the other loses automatically.

**Adversarial Search:**

Adversarial search is a game-playing technique where the agents are surrounded by a competitive environment. A conﬂicting goal is given to the agents (multiagent). These agents compete with one another and try to defeat one another in order to win the game. Such conﬂicting goals give rise to the [adversarial search](https://www.javatpoint.com/ai-adversarial-search). Here, game-playing means discussing those games where human intelligence and logic factor is used, excluding other factors such as luck factor. Tic-tac-toe, chess, checkers, etc., are such types of games where no luck factor works, only the mind work.

**Elements of Game Playing search**

To play a game, we use a game tree to know all the possible choices and to pick the best one out. There are the following elements of game-playing:

So: It is the initial state from which a game begins.

**PLAYER (s):** It deﬁnes which player is having the current turn to make a move in the state. ACTIONS (s): It deﬁnes the set of legal moves to be used in a state.

**RESULT (s, a):** It is a transition model which deﬁnes the result of a move. TERMINAL-TEST (s): It deﬁnes that the game has ended and returns true.

**UTILITY** (s,p): It deﬁnes the ﬁnal value with which the game has ended. This function is also known as the Objective function or Payoﬀ function. The price which the winner will get i.e. (-1): If the PLAYER loses. (+1): If the PLAYER wins.

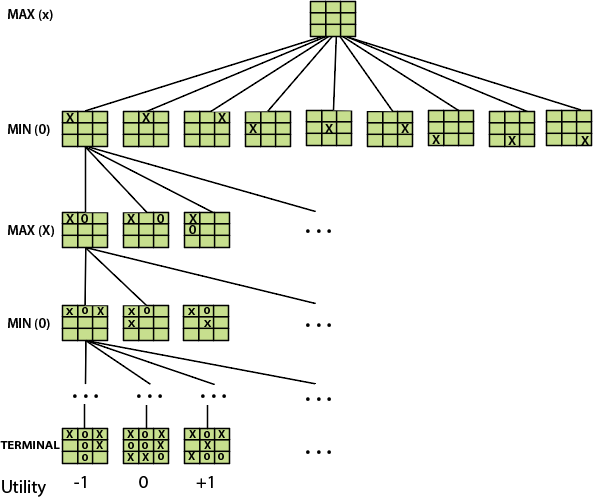
(0): If there is a draw between the PLAYERS.

For example, in chess, tic-tac-toe, we have two or three possible outcomes. Either to win, to lose, or to draw the match with values +1,-1, or 0.

Let’s understand the working of the elements with the help of a game tree designed for

tic-tac-toe.

Here, the node represents the game state and the edges represent the moves taken by the players.



**A game tree for tic-tac-toe**

i) **INITIAL STATE (So)**: The top node in the game tree represents the initial state in the tree and shows all the possible choices to pick out one.

**PLAYER (s)**: There are two players, MAX and MIN. MAX begins the game by picking one best move and placing X in the empty square box.

**ACTIONS (s)**: Both the players can make moves in the empty boxes chance by chance.

**RESULT (s, a)**: The moves made by MIN and MAX will decide the outcome of the game. **TERMINAL-TEST(s)**: When all the empty boxes will be ﬁlled, it will be the terminating state of the game.

**UTILITY:** At the end, we will get to know who wins: MAX or MIN, and accordingly, the price will be given to them.

**Types of algorithms in Adversarial Search:**

Unlike Normal Search, in Adversarial search, the result depends on the players which will decide the result of the game. It is also obvious that the solution for the goal state will be an optimal solution because the player will try to win the game with the shortest path and under limited time.

**MiniMax Algorithm:**

Minimax is a decision-making strategy, which is used to minimize the losing chances in a game and to maximize the winning chances. This strategy is also known as ‘Minmax’. It is a two-player game strategy where if one wins, the other loses the game. We can easily understand this strategy via a **game tree** where the nodes represent the states of the game and the edges represent the moves made by the players in the game. Players will be two namely:

**MIN:** Decrease the chances of **MAX** winning the game.

**MAX:** Increases his chances of winning the game.

In the minimax strategy, the result of the game or the utility value is generated by a **heuristic function** by propagating from the initial node to the root node. It follows the **backtracking technique** and backtracks to ﬁnd the best choice. MAX will choose that path that will increase its utility value and MIN will choose the opposite path which could help it to minimize MAX’s utility value.

MINIMAX Algorithm:

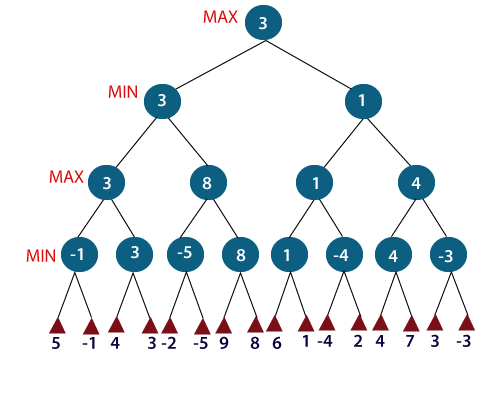
The MINIMAX strategy follows the **DFS (Depth-first search)** concept. Here, we have two players **MIN and MAX,** and the game is played alternatively between them, i.e., when **MAX** made a move, then the next turn is of **MIN.** It means the move made by MAX is ﬁxed and, he cannot change it. The same concept is followed in DFS strategy, i.e., we follow the same path and cannot change in the middle. That’s why in MINIMAX algorithm, instead of BFS, we follow DFS.

Keep on generating the game tree/ search tree till a limit **d.**

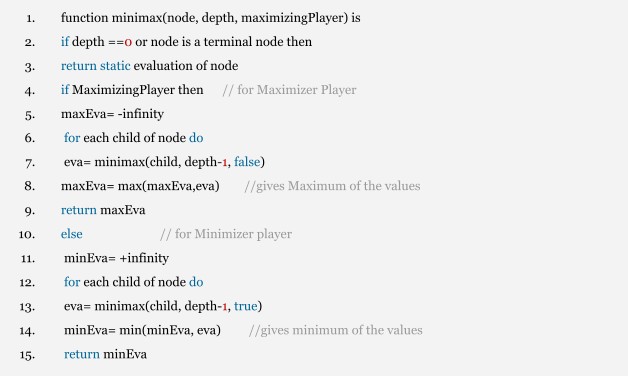
Compute the move using a heuristic function.

Propagate the values from the leaf node to the current position following the minimax strategy.

Make the best move from the choices.



For example, in the above ﬁgure, the two players **MAX** and **MIN** are there. **MAX** starts the game by choosing one path and propagating all the nodes of that path. Now, **MAX** will backtrack to the initial node and choose the best path where his utility value will be the maximum. After this, its **MIN** chance. **MIN** will also propagate through a path and again will backtrack, but **MIN** will choose the path which could minimize **MAX** winning chances or the utility value. So, if the level is minimizing, the node will accept the minimum value from the successor nodes. If the level is maximizing, the node will accept the maximum value from the successor.

**Pseudo Code for MiniMax Algorithm**

**Alpha-Beta Pruning:**

Alpha-beta pruning is an advanced version of the MINIMAX algorithm. The drawback of the minimax strategy is that it explores each node in the tree deeply to provide the best path among all the paths. Alpha-beta pruning reduces this drawback of the minimax strategy by less exploring the nodes of the search tree.

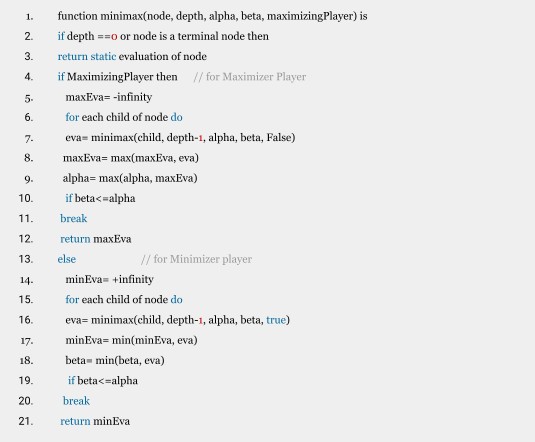
The method used in alpha-beta pruning is that it **cut oﬀ the search** by exploring less number of nodes. It makes the same moves as a minimax algorithm does, but it prunes the unwanted branches using the pruning technique (discussed in adversarial search). Alpha-beta pruning works on two threshold values, i.e. **-∞(alpha)** and +**∞(beta).**

-∞: It is the best highest value, a MAX player can have. It is the lower bound, which

represents a negative inﬁnity value.

● **+∞**: It is the best lowest value, a MIN player can have. It is the upper bound which represents positive inﬁnity.

Pseudo-code for Alpha-beta Pruning:

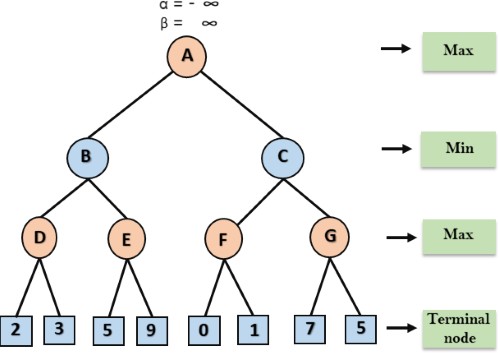


**Working of Alpha-Beta Pruning:**

Let's take an example of two-player search tree to understand the working of Alpha-beta pruning

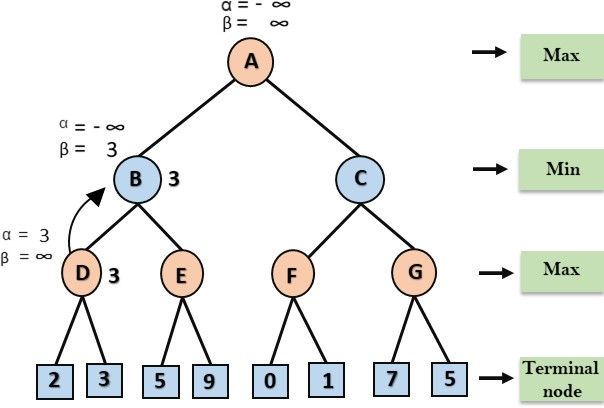
**Step 1:** At the ﬁrst step the, Max player will start ﬁrst move from node A where α= -∞ and β=

+∞, these value of alpha and beta passed down to node B where again α= -∞ and β= +∞, and Node B passes the same value to its child.



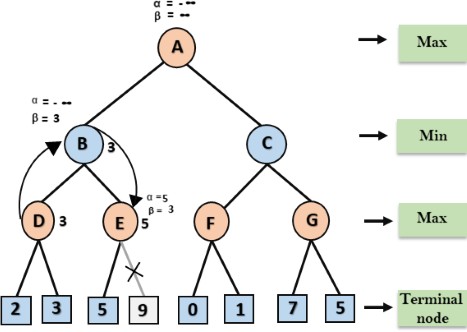
**Step 2:** At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with ﬁrstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

**Step 3:** Now algorithm backtrack to node B, where the value of β will change as this is a turn of Min, Now β= +∞, will compare with the available subsequent nodes value, i.e. min (∞, 3) = 3, hence at node B now α= -∞, and β= 3.



In the next step, algorithm traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.

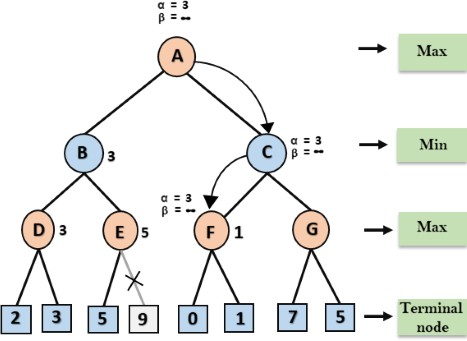
**Step 4:** At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.



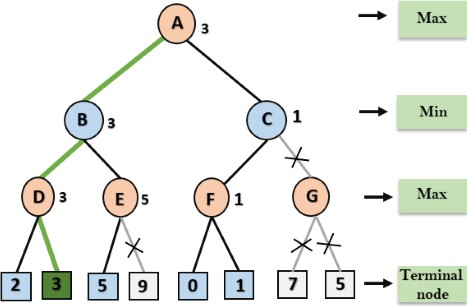
**Step 5:** At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.

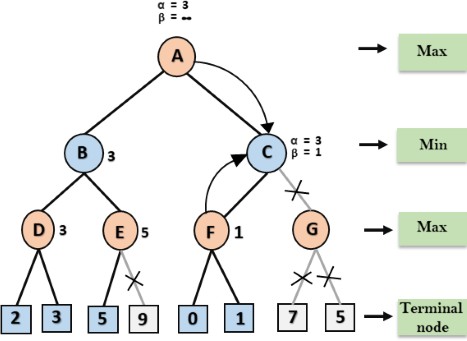
At node C, α=3 and β= +∞, and the same values will be passed on to node F.

**Step 6:** At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.



**Step 7:** Node F returns the node value 1 to node C, at C α= 3 and β= +∞, here the value of beta will be changed, it will compare with 1 so min (∞, 1) = 1. Now at C, α=3 and β= 1, and again it satisﬁes the condition α>=β, so the next child of C which is G will be pruned, and the algorithm will not compute the entire sub-tree G.





**Step 8:** C now returns the value of 1 to A here the best value for A is max (3, 1) = 3. Following is the ﬁnal game tree which is the showing the nodes which are computed and nodes which has never been computed. Hence the optimal value for the maximizer is 3 for this example.

**Constraint Satisfaction Problems:**

Constraint satisfaction is a technique where a problem is solved when its values satisfy certain constraints or rules of the problem. Such type of technique leads to a deeper understanding of the problem structure as well as its complexity.

Constraint satisfaction depends on three components, namely:

X: It is a set of variables.

D: It is a set of domains where the variables reside. There is a speciﬁc domain for each variable.

C: It is a set of constraints which are followed by a set of variables.

In constraint satisfaction, domains are the spaces where the variables reside, following the problem-speciﬁc constraints. These are the three main elements of a constraint satisfaction technique. The constraint value consists of a pair of {scope, rel}. The scope is a tuple of variables that participate in the constraint and rel is a relation that includes a list of values that the variables can take to satisfy the constraints of the problem.

**Solving Constraint Satisfaction Problems**

The requirements to solve a constraint satisfaction problem (CSP) are as follows:

A state-space notion of the solution.

A state in state-space is deﬁned by assigning values to some or all variables such as

{X1=v1, X2=v2, and so on…}.

**An assignment of values to a variable can be done in three ways:**

**Consistent or Legal Assignment:** An assignment which does not violate any constraint or rule is called Consistent or legal assignment.

**Complete Assignment:** An assignment where every variable is assigned with a value, and the solution to the CSP remains consistent. Such assignment is known as Complete assignment.

**Partial Assignment:** An assignment which assigns values to some of the variables only. Such type of assignments are called Partial assignments.

Types of Domains in CSP

**There are following two types of domains which are used by the variables :**

**Discrete Domain:** It is an inﬁnite domain which can have one state for multiple variables. **For example,** a start state can be allocated inﬁnite times for each variable.

**Finite Domain:** It is a ﬁnite domain which can have continuous states describing one domain for one speciﬁc variable. It is also called a continuous domain.

With respect to the variables, basically there are following types of constraints:

**Unary Constraints:** It is the simplest type of constraints that restricts the value of a single variable.

**Binary Constraints:** It is the constraint type which relates two variables. A value x2 will contain a value which lies between x1 and x3.

**Global Constraints:** It is the constraint type which involves an arbitrary number of variables.

Some special types of solution algorithms are used to solve the following types of constraints:

**Linear Constraints:** These type of constraints are commonly used in linear programming where each variable containing an integer value exists in linear form only.

**Non-linear Constraints:** These type of constraints are used in non-linear programming where each variable (an integer value) exists in a non-linear form.

**Note:** A special constraint that works in the real world is known as the Preference constraint. Constraint Propagation

In local state spaces, the choice is only one, i.e., to search for a solution. But in CSP, we have two choices either:

We can search for a solution or We can perform a special type of inference called constraint propagation.

Constraint propagation is a special type of inference that helps in reducing the legal number of values for the variables. The idea behind constraint propagation is local consistency.

In local consistency, variables are treated as nodes, and each binary constraint is treated as an arc

in the given problem. There are the following local consistencies which are discussed below:

Node Consistency: A single variable is said to be node consistent if all the values in the variable’s domain satisfy the unary constraints on the variables.

Arc Consistency: A variable is arc consistent if every value in its domain satisﬁes the binary constraints of the variables.

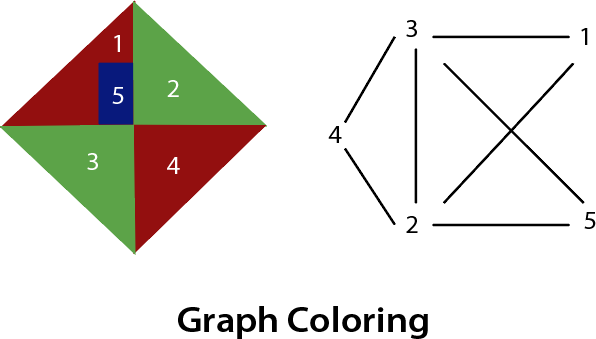
**Path Consistency**: When the evaluation of a set of two variables with respect to a third variable can be extended over another variable, satisfying all the binary constraints. It is similar to arc consistency.

**k-consistency:** This type of consistency is used to deﬁne the notion of stronger forms of propagation. Here, we examine the k-consistency of the variables.

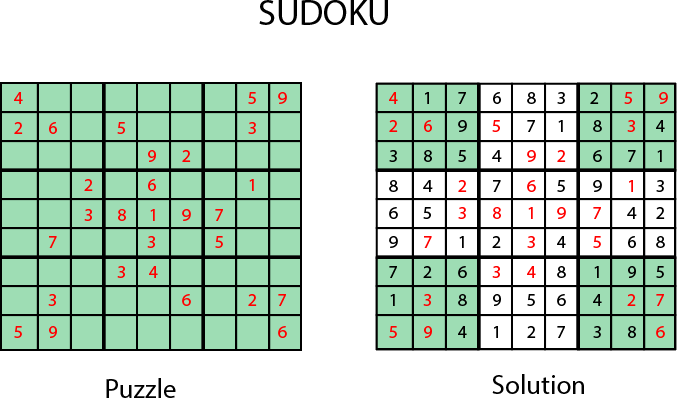
**CSP Problem examples**

Constraint satisfaction includes those problems which contain some constraints while solving the problem. CSP includes the following problems:

**Graph Coloring:** The problem where the constraint is that no adjacent sides can have the same color.

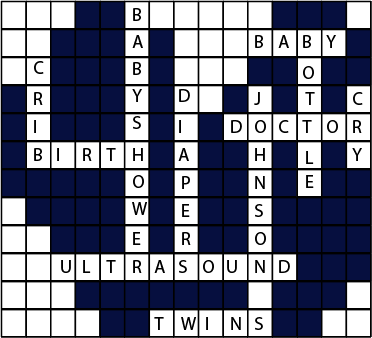


**Sudoku Playing:** The gameplay where the constraint is that no number from 0-9 can be repeated in the same row or column.

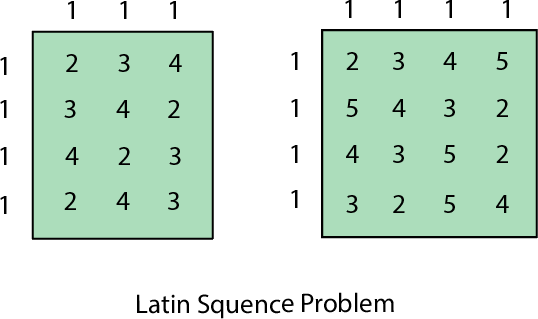


**n-queen problem:** In n-queen problem, the constraint is that no queen should be placed either diagonally, in the same row or column.

**Crossword:** In crossword problem, the constraint is that there should be the correct formation of the words, and it should be meaningful.



**Latin square Problem:** In this game, the task is to search the pattern which is occurring several times in the game. They may be shuﬄed but will contain the same digits.



**Example 1:** Implement minimax algorithm.

class Node:

def \_\_init\_\_(self, value=None, children=None):

self.value = value

self.children = children if children is not None else []

def minimax(node, depth, maximizing\_player):

if depth == 0 or not node.children:

return node.value

if maximizing\_player:

max\_eval = float('-inf')

for child in node.children:

eval = minimax(child, depth - 1, False)

max\_eval = max(max\_eval, eval)

return max\_eval

else:

min\_eval = float('inf')

for child in node.children:

eval = minimax(child, depth - 1, True)

min\_eval = min(min\_eval, eval)

return min\_eval

# Sample tree

root = Node('A')

n1 = Node('B')

n2 = Node('C')

n3 = Node(9)

n4 = Node(8)

n5 = Node(7)

root.children = [n1, n2]

n1.children = [n3, n4]

n2.children = [n5]

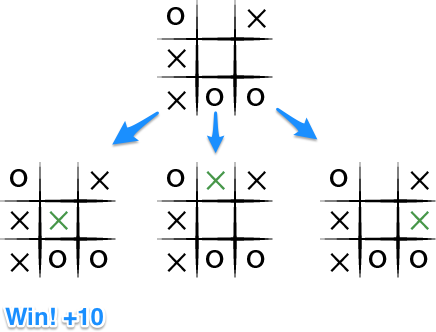
# Example usage

depth = 2

result = minimax(root, depth, True)

print("The optimal value from the root is:", result)

**Example 2:** Implement Game Search Algorithm to solve the tic-tac-toe problem mentioned below.



**#minimax example**

import math

def print\_board(board):

    print(f"{board[0]} | {board[1]} | {board[2]}")

    print("-"\*9)

    print(f"{board[3]} | {board[4]} | {board[5]}")

    print("-"\*9)

    print(f"{board[6]} | {board[7]} | {board[8]}")

def get\_empty\_cells(board):

    return [i for i in range(9) if board[i] == " "]

def check\_win(board, player):

    win\_conditions = [

        [0, 1, 2], [3, 4, 5], [6, 7, 8],  # rows

        [0, 3, 6], [1, 4, 7], [2, 5, 8],  # columns

        [0, 4, 8], [2, 4, 6]             # diagonals

    ]

    for condition in win\_conditions:

        if board[condition[0]] == board[condition[1]] == board[condition[2]] == player:

            return True

    return False

def get\_score(board, player):

    if check\_win(board, player):

        return 1

    elif check\_win(board, "X" if player == "O" else "O"):

        return -1

    else:

        return 0

def minimax(board, depth, player):

    empty\_cells = get\_empty\_cells(board)

    if check\_win(board, "X"):

        return 1

    elif check\_win(board, "O"):

        return -1

    elif not empty\_cells:

        return 0

    scores = []

    for cell in empty\_cells:

        board[cell] = player

        score = minimax(board, depth+1, "O" if player == "X" else "X")

        scores.append(score)

        board[cell] = " "

    if player == "X":

        return max(scores)

    else:

        return min(scores)

def get\_best\_move(board, player):

    empty\_cells = get\_empty\_cells(board)

    best\_score = -math.inf if player == "X" else math.inf

    best\_move = None

    for cell in empty\_cells:

        board[cell] = player

        score = minimax(board, 0, "O" if player == "X" else "X")

        board[cell] = " "

        if player == "X" and score > best\_score:

            best\_score = score

            best\_move = cell

        elif player == "O" and score < best\_score:

            best\_score = score

            best\_move = cell

    return best\_move

def play\_game():

    board = [" "]\*9

    players = ["X", "O"]

    turn = 0

    while not check\_win(board, players[0]) and not check\_win(board, players[1]) and " " in board:

        print\_board(board)

        player = players[turn]

        if player == "X":

            move = get\_best\_move(board, player)

        else:

            move = int(input(f"Player {player}, choose your move (1-9): ")) - 1

        board[move] = player

        turn = (turn + 1) % 2

    print\_board(board)

    if check\_win(board, "X"):

        print("X wins!")

    elif check\_win(board, "O"):

        print("O wins!")

    else:

        print("Tie game")

play\_game()

**Example 3:** Implement the tree using alpha beta pruning.



#alpha beta pruning example

import math

class Node:

    def \_\_init\_\_(self, value=None):

        self.value = value

        self.children = []

def alpha\_beta(node, depth, alpha, beta, maximizing\_player=True):

    if depth == 0 or not node.children:

        return node.value

    if maximizing\_player:

        value = -math.inf

        for child in node.children:

            value = max(value, alpha\_beta(child, depth-1, alpha, beta, False))

            alpha = max(alpha, value)

            if beta <= alpha:

                break

        return value

    else:

        value = math.inf

        for child in node.children:

            value = min(value, alpha\_beta(child, depth-1, alpha, beta, True))

            beta = min(beta, value)

            if beta <= alpha:

                break

        return value

# Sample tree

root = Node()

root.value = 'A'

n1 = Node('B')

n2 = Node('C')

root.children = [n1, n2]

n3 = Node('D')

n4 = Node('E')

n5 = Node('F')

n6 = Node('G')

n1.children = [n3, n4]

n2.children = [n5, n6]

n7 = Node(2)

n8 = Node(3)

n9 = Node(5)

n10 = Node(9)

n3.children = [n7, n8]

n4.children = [n9, n10]

n11 = Node(0)

n12 = Node(1)

n13 = Node(7)

n14 = Node(5)

n5.children = [n11, n12]

n6.children = [n13, n14]

# Example usage

value = alpha\_beta(root, 3, -math.inf, math.inf)

print(value)

**Example 4:** Define a CSP problem with three variables ('a', 'b', 'c') that can take on values between 1 and 3. We add constraints to the problem to ensure that no two variables can have the same value. We then call the **getSolutions** method to find all possible solutions to the problem

!pip install python-constraint

from constraint import \*

# Define the problem

problem = Problem()

# Add variables to the problem

problem.addVariables(['a', 'b', 'c'], range(1, 4))

# Add constraints to the problem

problem.addConstraint(lambda a, b: a != b, ('a', 'b'))

problem.addConstraint(lambda b, c: b != c, ('b', 'c'))

problem.addConstraint(lambda a, c: a != c, ('a', 'c'))

# Solve the problem

solutions = problem.getSolutions()

# Print the solutions

for solution in solutions:

    print(solution)