**Department of Computing**

**CS370: Artificial Intelligence**

**Class: BSCS-10AB**

**Lab 03: Minimax with Alpha Beta Pruning**

**Date: 16-02-2023**

**Instructor: Dr. Hashir Kiani**

**Lab Engineer: Ms Shakeela Bibi**

# Lab 03: Minimax Algorithm in Game Theory | (Alpha-Beta Pruning)

**Introduction**

Finding the minimax value of a game is an important problem in a variety of fields, including game theory, decision theory, statistics, philosophy, economics, robotics, and security. Classical algorithms such as the minimax algorithm can be used to find the minimax value. In this lab you will implement minimax algorithm, as well as how it can be sped up using alpha-beta pruning.

Alpha-Beta pruning is not actually a new algorithm, rather an optimization technique for minimax algorithm. It reduces the computation time by a huge factor. This allows us to search much faster and even go into deeper levels in the game tree. It cuts off branches in the game tree which need not be searched because there already exists a better move. It is called Alpha-Beta pruning because it passes 2 extra parameters in the minimax function, namely alpha and beta.

Let’s define the parameters alpha and beta.

Alpha is the best value that the maximizer currently can guarantee at that level or above.

Beta is the best value that the minimizer currently can guarantee at that level or above.

**Objective**

The objective of this lab is to implement minimax algorithm, as well as how it can be sped up using alpha-beta pruning.

**Tools/Software Requirement**

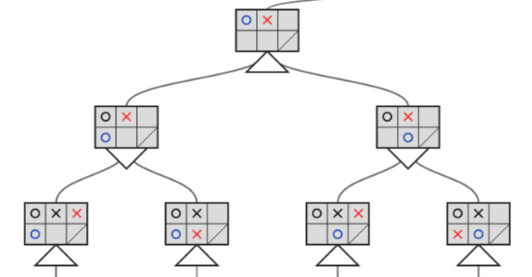
Python, & its libraries

**Description**

## **Minimax Algorithm:**

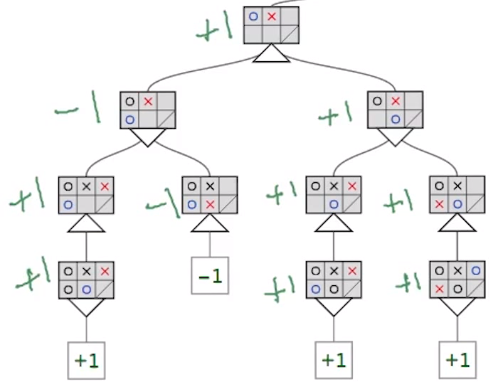
The minimax algorithm is more intuitive to understand in terms of a brute-force approach. It tries to see every possible outcome and then tries to optimize whatever options it has in hand.

We generate a tree consisting of branches going to a depth containing all the set of all possible moves made by the player(s) in the game. We will next, need to traverse this tree.



The downward triangle represents a location in the tree where minimax will minimize the opponent’s advantage. Whereas the upward triangles are the locations where minimax maximizes the user’s advantage.

Intuitively, to know a player’s advantage, it first needs to know which path leads to victory. This is where the brute force approach comes into the picture. This essentially means that the algorithm must traverse to the bottom of the tree and go through every possible move. Next, it has to assign some weight (e.g., +1 for a win and -1 for a loss) and propagate this weight up through the tree.

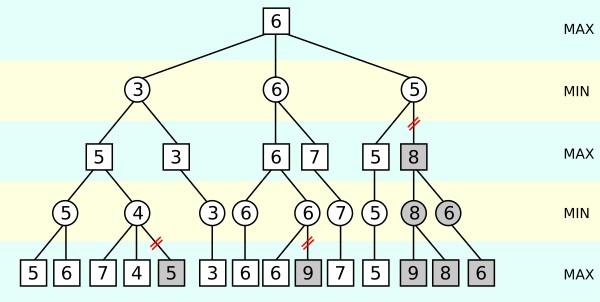


## **Alpha-Beta Pruning**

Alpha Beta Pruning is all about reducing the size (pruning) of our search tree. While a brute-force approach is an easier approach to use, it doesn’t necessarily mean it is the most optimal approach. Many times, one doesn’t need to visit all possible branches to come up with the best possible solution in hand.

Thus, we need to provide the min-max algorithm with some stopping criteria using which it would stop searching a region of the tree once it finds the guaranteed minimum or maximum at that level. This would prevent the algorithm from using additional computational time, making it much more responsive and faster.

In the image below, we have a tree with various scores assigned to each node. Some nodes are shaded in red, indicating there’s no need to review them.



At the bottom left of the tree, minimax goes through the values 5 and 6 on the bottom max level. It determines that 5 must be assigned to the min level right above it.

But, after looking at 7 and 4 of the right max level branch, it realizes that the above min level node must be assigned a maximum value of 4. Since the second max level right above the first min level will take the maximum between 5 and at most 4, it’s clear that it’ll choose 5. Following this, it would continue traversing the tree to perform the same exact set of operations within the tree’s other branches.

Concluding, we can easily understand how alpha-beta pruning acts as a great optimization over the min-max algorithm and how these algorithms are the foundation of state-space searching techniques, paving the way for much more advanced approaches to solving such problems.

Useful Links:

**Minimax with Alpha Beta Pruning**

[Minimax with Alpha Beta Pruning](https://www.youtube.com/watch?v=zp3VMe0Jpf8)

**Note before moving to the task please watch the video for pseudo code explanation**.

<https://www.youtube.com/watch?v=l-hh51ncgDI>

**Task:**

**Implement minimax algorithm, & speed it up using alpha-beta pruning.**

**Minimax pseudo code:**

**function minimax(position, depth, maximizingPlayer)**

**if depth == 0 or game over in position**

**return static evaluation of position**

**if maximizingPlayer**

**maxEval = -infinity**

**for each child of position**

**eval = minimax(child, depth - 1, false)**

**maxEval = max(maxEval, eval)**

**return maxEval**

**else**

**minEval = +infinity**

**for each child of position**

**eval = minimax(child, depth - 1, true)**

**minEval = min(minEval, eval)**

**return minEval**

**// initial call**

**minimax(currentPosition, 3, true)**

**Alpha Beta Pseudo code**

**function minimax(position, depth, alpha, beta, maximizingPlayer)**

**if depth == 0 or game over in position**

**return static evaluation of position**

**if maximizingPlayer**

**maxEval = -infinity**

**for each child of position**

**eval = minimax(child, depth - 1, alpha, beta false)**

**maxEval = max(maxEval, eval)**

**alpha = max(alpha, eval)**

**if beta <= alpha**

**break**

**return maxEval**

**else**

**minEval = +infinity**

**for each child of position**

**eval = minimax(child, depth - 1, alpha, beta true)**

**minEval = min(minEval, eval)**

**beta = min(beta, eval)**

**if beta <= alpha**

**break**

**return minEval**

**// initial call**

**minimax(currentPosition, 3, -∞, +∞, true)**

**General Functions**

def current\_player(position):

# Count the number of X's and O's in the game board

x\_count = sum(row.count('X') for row in position)

o\_count = sum(row.count('O') for row in position)

# If there are more X's than O's, the current player is O. Otherwise, the current player is X.

if x\_count > o\_count:

return 'O'

else:

return 'X'

def generate\_children(position):

# Assume that the position is represented as a list of lists, where each sublist represents a row of the game board

children = []

# Iterate over every empty cell in the game board

for i in range(len(position)):

for j in range(len(position[i])):

if position[i][j] == ' ':

# If the cell is empty, create a new child position where the current player has placed their piece in that cell

# Make a copy of the original position

child = [row[:] for row in position]

# Place the current player's piece in the cell

child[i][j] = current\_player(child)

children.append(child)

return children

def game\_over(position):

# Check rows for a win

for row in position:

if len(set(row)) == 1 and row[0] != ' ':

return True, row[0]

# Check columns for a win

for i in range(len(position[0])):

column = [position[j][i] for j in range(len(position))]

if len(set(column)) == 1 and column[0] != ' ':

return True, column[0]

# Check diagonals for a win

diagonal1 = [position[i][i] for i in range(len(position))]

if len(set(diagonal1)) == 1 and diagonal1[0] != ' ':

return True, diagonal1[0]

diagonal2 = [position[i][len(position)-1-i] for i in range(len(position))]

if len(set(diagonal2)) == 1 and diagonal2[0] != ' ':

return True, diagonal2[0]

# Check for a tie

if all(position[i][j] != ' ' for i in range(len(position)) for j in range(len(position[i]))):

return True, 'Tie'

# If there is no winner and the game is not over, return False

return False, None

def evaluate(position, player):

# Define the evaluation function for a given player

def eval\_player(symbol):

if symbol == player:

return 1

elif symbol == ' ':

return 0

else:

return -1

# Check rows for the number of pieces in a row for the player

row\_counts = [sum(eval\_player(symbol) for symbol in row)

for row in position]

row\_count = max(row\_counts) if max(row\_counts) >= 1 else 0

# Check columns for the number of pieces in a column for the player

col\_counts = [sum(eval\_player(position[i][j])

for i in range(3)) for j in range(3)]

col\_count = max(col\_counts) if max(col\_counts) >= 1 else 0

# Check diagonal for the number of pieces in the diagonal for the player

diagonal1\_count = sum(eval\_player(position[i][i]) for i in range(3))

diagonal2\_count = sum(eval\_player(position[i][2-i]) for i in range(3))

diagonal\_count = max(diagonal1\_count, diagonal2\_count) if max(

diagonal1\_count, diagonal2\_count) >= 1 else 0

# Return the total count

return row\_count + col\_count + diagonal\_count

**Task 1**

**Code:**

**def minimax(position, depth, maximizingPlayer):**

values = [3,4,2,1,7,8,9,10,2,11,1,12,14,9,13,16]

**# Check if the game is over or if the search has reached maximum depth.**

**if depth == 0 or game\_over(position):**

**return evaluate(position)**

**# If it is the turn of the maximizing player.**

**if maximizingPlayer:**

**maxEval = -float('inf')**

**# Evaluate the children of the current position.**

**for child in generate\_children(position, current\_player(position)):**

**# Recursively call minimax on each child.**

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

**# Update the maximum value.**

**maxEval = max(maxEval, eval)**

**return maxEval**

**# If it is the turn of the minimizing player.**

**else:**

**minEval = float('inf')**

**# Evaluate the children of the current position.**

**for child in generate\_children(position, current\_player(position)):**

**# Recursively call minimax on each child.**

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

**# Update the minimum value.**

**minEval = min(minEval, eval)**

**return minEval**

**minimax(0, 3, -float('inf'))**

**ScreenShot:**



**Task 2**

**Code:**

**def minimax(position, depth, alpha, beta, maximizingPlayer):**

values = [3,4,2,1,7,8,9,10,2,11,1,12,14,9,13,16]

**# If we've reached the maximum depth or the game is over in this position, return the evaluation score**

**if depth == 0 or game\_over(position):**

**return evaluate(position)**

**if maximizingPlayer:**

**# We're maximizing, so set the initial maximum evaluation to negative infinity**

**max\_eval = -float('inf')**

**# Generate all possible child positions from this position**

**for child in generate\_children(position):**

**# Recursively evaluate the child position, with a lower depth and as a minimizing player**

**eval = minimax(child, depth - 1, alpha, beta, False)**

**# Update the maximum evaluation with the child's evaluation**

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

**# Update the alpha value to the maximum evaluation so far**

**alpha = max(alpha, eval)**

**# If the beta value is less than or equal to the alpha value, we can stop evaluating the other child positions**

**if beta <= alpha:**

**break**

**# Return the maximum evaluation**

**return max\_eval**

**else:**

**# We're minimizing, so set the initial minimum evaluation to positive infinity**

**min\_eval = float('inf')**

**# Generate all possible child positions from this position**

**for child in generate\_children(position):**

**# Recursively evaluate the child position, with a lower depth and as a maximizing player**

**eval = minimax(child, depth - 1, alpha, beta, True)**

**# Update the minimum evaluation with the child's evaluation**

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

**# Update the beta value to the minimum evaluation so far**

**beta = min(beta, eval)**

**# If the beta value is less than or equal to the alpha value, we can stop evaluating the other child positions**

**if beta <= alpha:**

**break**

**# Return the minimum evaluation**

**return min\_eval**

**minimax(0, 3, -float('inf'), float('inf'), True)**

**ScreenShot:**



**Findings From The Lab:**

**Alpha-beta pruning** is an optimization technique that can be applied to the minimax algorithm, which is a commonly used algorithm in artificial intelligence for finding the optimal move in a two-player, zero-sum game.

* **Improved efficiency**: Alpha-beta pruning reduces the number of nodes that need to be evaluated by the minimax algorithm.
* **Increased depth of search**: alpha-beta pruning reduces the number of nodes that need to be evaluated, it can allow the algorithm to search deeper into the game tree than would be possible with the standard minimax algorithm.
* **Faster search times**: By reducing the number of nodes that need to be evaluated, alpha-beta pruning can also lead to faster search times, allowing the algorithm to make a decision more quickly.