CS300 - Spring 2024-2025 Sabancı University Homework #1

TicTacToe AI

Introduction

In this assignment, you will implement an N-ary tree structure to represent the state of the game of Tic-Tac-Toe and also implement **Minimax** algorithm with alpha beta pruning to determine the optimal move set for AI. This assignment will walk you through tree-based decision making and introduce you to basic game AI logic.

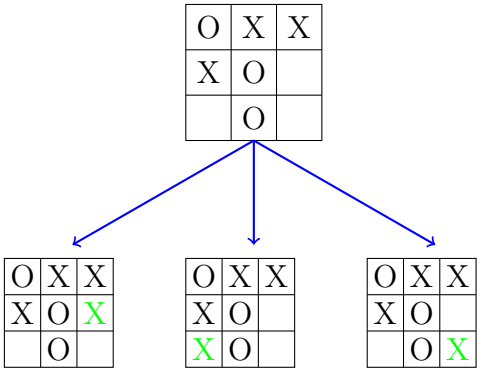


Figure 1: Tic-Tac-Toe Game Tree Illustration

In order to represent every possible state of the board, you will implement a N-ary tree. N-ary trees are generalized binary trees with more than two child. Each node can have at most N children. You can think of tree node having a list of nodes to children and a key. In our game, which is Tic-Tac-Toe, each node will have a copy of board and each children will have the all possible states for AI to explore. Due to the nature of the minimax algorithm, you should also implement an Insert function and you should also traverse the tree in post-order manner for minimax algorithm to work. Details of the tree consturction will be explained later.

This tree will be used for AI to make decisions. But before going through the whole program flow, AI will make decisions based on an algorithm called Minimax with alpha and beta pruning for optimization.

Minimax Algorithm with alpha and beta pruning

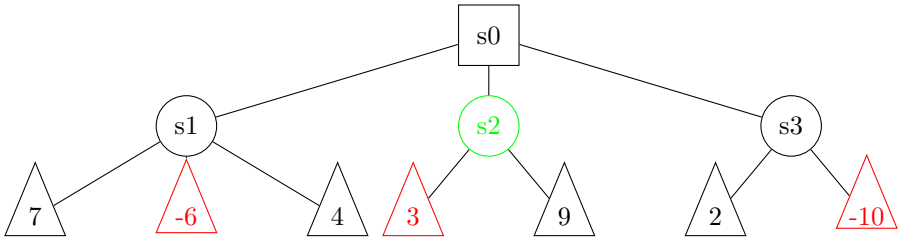
In minimax algorithm, there are two types of players. The maximizing player and the minimizing player. Starting player will be the maximizing player and second player will be the minimizing player. What we mean by maximizing and minimizing is that each player receives points upon the

moves they make. This is called an utility function. Utility function or payoff function decides how good is each termianl state(leaf nodes) for the player. In our game, utilities will be as follows:

- Win: +10 points
- Draw: 0 points
- Loss: -10 points

Players alternate among the tree starting from the maximizing player. This means for depth d=0, we have the maximizing players decision and then for depth d=1, which is for minimizing player, we will list all the possible move set for the AI and for depth d=2, player's possible moves to each of the AI's moves. This will go until we exhaust all the move sets. Max wants to maximize termianl nodes and min player wants to minimize the termianl nodes. Note that since Tic-Tac-Toe is played on 3x3 board, our max depth for the tree can be 9. After calculating all the possible moves, AI makes a move and update the board.

Minimax Algorithm Example



In the above, Square is maximizing player's node and circle is minimizing player's node. Notice that if max goes to s1, min goes to -6, if max goes to s2, min goes to 3 and if max goes to s3, min goes to -10. Since the set of nodes max gets is {-6, 3, -10} max chooses s2 and gets 3 points. In our game we will only have 10, 0 and -10 which will be determined by the state of the board.

Let us explore further our example above. Consider the case where we explore the node s2. So far we know that maximizing player can guarantee -6. We explore s2 and notice we can have 3. When exploring s3, we notice that we get at max 2 from s3, so we do not need to further explore down the -10 path as it would make no difference over the choice of path for the max player because the maximum we can achieve from s3 is always smaller than s2. (At max we can have 2 from the s3, because if something was bigger than 3 in the place of -10, min would choose 2, otherwise they would choose even smaller number than 3 like -10.)For this we have the following parameters.

- Alpha(α): Best value max player can guarantee so far
- Beta(β): Best value min player can guarantee so far

During you search prune the branches when $\beta \leq \alpha$.
Below we provide a pseudocode for the algorithm minimax with alpha and beta pruning. You should implement this algorithm exactly and move in post-order(depth first) manner in the tree.

The way you find the best move is as follows:

- Build a tree from the current state
- expand the root to generate all possible moves
- Evaluate each move using Minimax
- Select the move with highest value for AI.

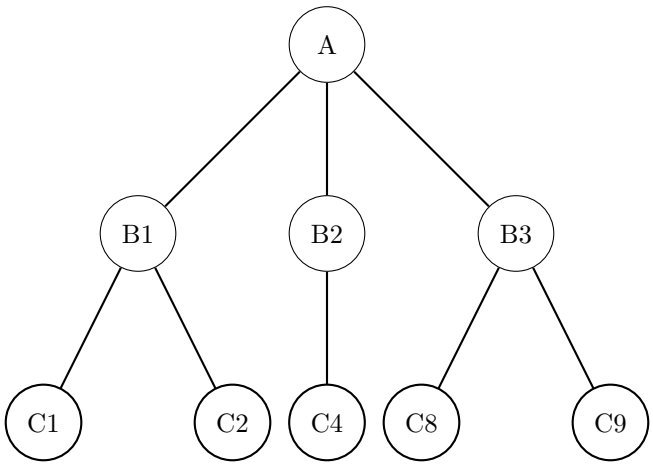
If on a level, there are multiple moves with same value for the payoff, select the one that occurs first in iteration order. Or in other words, only update the move if you strictly find a better solution than the existing one.

Algorithm 1 Alpha-Beta Minimax

```
1: function AlphaBetaMinimax(state, depth, α, β, maximizingPlayer)
2:     if IsTerminal(state) or depth = 0 then return Utility(state)
3:     end if
4:     if maximizingPlayer then
5:         maxEval ← −∞                          ▷ note: you can use INTMIN here
6:         for all action ∈ Actions(state) do
7:             eval ← AlphaBetaMinimax(Result(state, action), depth − 1, α, β, False)
8:             maxEval ← max(maxEval, eval)
9:             α ← max(α, eval)
10:            if β ≤ α then
11:                break                          ▷ Beta cut-off
12:            end if
13:        end for
           return maxEval
14:    else
15:        minEval ← ∞
16:        for all action ∈ Actions(state) do
17:            eval ← AlphaBetaMinimax(Result(state, action), depth − 1, α, β, True)
18:            minEval ← min(minEval, eval)
19:            β ← min(β, eval)
20:            if β ≤ α then
21:                break                          ▷ Alpha cut-off
22:            end if
23:        end for
           return minEval
24:    end if
25: end function
```

N-ary Tree

An N-ary tree is a tree data structure where each node can have up to N children. Unlike binary trees, which are limited to two children per node, an N-ary tree lets you model situations where a state can branch out in many directions just like our Tic-Tac-Toe game.



You will directly implement this structure where each node have a board, a children array where each element is a node pointer, the value and move.

TreeNode

Each node in the tree encapsulates four key pieces of information that are crucial for evaluating game moves. First, the game state field holds a complete snapshot of the board configuration at that node, ensuring that all future decisions are based on an accurate representation of the game. Second, the move field records the specific action (represented by row and column coordinates) that was taken to reach that state from its parent, allowing you to trace the sequence of moves leading to that point. Third, the children field is a collection that stores all the child nodes, each representing a potential future move from the current state; this forms the branches of the N-ary tree and allows the algorithm to explore every possible continuation of the game. Finally, the value field is used to store the evaluation score of the state, which is computed during the recursive minimax process; this score indicates how favorable the state is for one player over the other and is critical in selecting the optimal move.

Insertion

Insertion is performed by systematically expanding a node to include all its possible future game states. The process starts by scanning the board in a row-major order, examining each cell from the top-left to the bottom-right, to identify all positions that are still available for a move. For every available cell, a new game state is created by duplicating the current board and then applying the corresponding move. Each of these new states, along with the move that led to it, is encapsulated in a new node that is added as a child of the current node. This method ensures that all potential moves are accounted for in a consistent and predictable order, which is crucial for reliable tie-breaking and effective pruning during the decision-making process.

Traversal

The traversal of the game state tree is performed using a depth-first approach. Starting at the root, the algorithm recursively visits each child node in the order they were created, moving as deep as possible into each sequence of moves until it reaches a terminal state or a predefined depth. Once a branch has been fully explored, the algorithm backtracks to examine the next available branch. This process is essentially a post-order traversal, meaning that each node is processed after all of its children have been visited.

General Flow of the Program

1. Create a tree structure where:
    - Each node represents a game state (board configuration)
    - Each node stores:
        - A copy of the board
        - The move that led to this state (row, column)
        - An evaluation value
        - References to child nodes
2. For the AI decision process:
    - Start with the current board state as the root node
    - Generate all possible next moves (children of the root)
    - For each possible move:
        - Create a new game state by applying the move
        - Evaluate this state recursively using the Minimax algorithm
3. The recursive evaluation process:
    - If the state is terminal (win, loss, or draw) or maximum depth is reached:
        - Return the evaluation score (+10 for AI win, -10 for human win, 0 for draw)
    - Otherwise:
        - Generate all possible next moves (children)
        - If it's the maximizing player's turn:
            - Find the maximum evaluation among all children
            - Update alpha value
        - If it's the minimizing player's turn:
            - Find the minimum evaluation among all children
            - Update beta value
        - Prune branches where beta $\leq$ alpha
4. After evaluating all possible immediate moves:
    - If AI is maximizing, select the move with the highest evaluation
    - If AI is minimizing, select the move with the lowest evaluation
5. Apply the selected move to the game board
6. Destruct the old tree and construct a new tree with our new board replacing the root node.
7. Keep repeating these steps untill one of the players wins or there is a Tie.

Input and Output Format

Your program should begin with taking input from the user. The following sentence should be printed out for the player.

```
Do you want to play as X or O? (X goes first):
```

After user enters input, you should display the welcoming sentence and also you should print whether the player is playing as X or O and do it for the AI as well. After that, you should print the board with an index based on 1.

```
Welcome to Tic Tac Toe!
You are playing as X.
The AI is playing as O.
Enter a number from 1-9 to make your move:

 1 | 2 | 3
---+---+---
 4 | 5 | 6
---+---+---
 7 | 8 | 9
```

Once you output that, you should keep outputting the final version of the game board and keep asking for the move for the player after AI makes a move.
After each move, print the position that AI chooses and also count the number of nodes AI visits in the tree. If player enters an index that is already taken, it should also indicate to user to pick another place for the move and should keep asking until the player enters a correct position.

```
That position is already taken. Try again.
Your move (1-9):
```

Your program should also announce the winner properly.
If the player winds, you should print

```
Congratulations! You win!
```

If AI wins, you should print

```
The AI wins!
```

If its a draw then you should print

```
It's a draw!
```

Sample Run

Below you can find some sample runs about the code so you can better understand the flow of the program and input/output format.

```
Do you want to play as X or O? (X goes first): X
Welcome to Tic Tac Toe!
You are playing as X.
The AI is playing as O.
Enter a number from 1-9 to make your move:
```

```
 1 | 2 | 3
---+---+---
 4 | 5 | 6
---+---+---
 7 | 8 | 9

Your move (1-9): 1

 X |   |
---+---+---
   |   |
---+---+---
   |   |

AI is making a move...
AI chose position 5.
Nodes explored: 4090

 X |   |
---+---+---
   | O |
---+---+---
   |   |

Your move (1-9): 2

 X | X |
---+---+---
   | O |
---+---+---
   |   |

AI is making a move...
AI chose position 3.
Nodes explored: 221

 X | X | O
---+---+---
   | O |
---+---+---
   |   |

Your move (1-9): 4

 X | X | O
---+---+---
 X | O |
---+---+---
   |   |

AI is making a move...
AI chose position 7.
```

```
Nodes explored: 25

 X | X | O
---+---+---
 X | O |
---+---+---
 O |   |

The AI wins!
```

Sample Run 2

```
Do you want to play as X or O? (X goes first): O
Welcome to Tic Tac Toe!
You are playing as O.
The AI is playing as X.
Enter a number from 1-9 to make your move:

 1 | 2 | 3
---+---+---
 4 | 5 | 6
---+---+---
 7 | 8 | 9

AI is making a move...
AI chose position 1.
Nodes explored: 30710

 X |   |
---+---+---
   |   |
---+---+---
   |   |

Your move (1-9): 4

 X |   |
---+---+---
 O |   |
---+---+---
   |   |

AI is making a move...
AI chose position 2.
Nodes explored: 1682

 X | X |
---+---+---
 O |   |
---+---+---
   |   |

Your move (1-9): 5
```

```
 X | X |
---+---+---
 O | O |
---+---+---
   |   |

AI is making a move...
AI chose position 3.
Nodes explored: 65

 X | X | X
---+---+---
 O | O |
---+---+---
   |   |

The AI wins!
```

Submission Details

- Submissions will be done via SUcourse with CodeRunner in order to prevent compilation errors
- Ensure that this file is the latest version of your homework program.
- Submit via `SUCourse` ONLY.
- Submissions through e-mail or other means (e.g., paper) will receive no credit.
- Submit your own work, even if it is not fully functional. Submitting similar programs is easily detectable.

Good Luck!