University of Sarajevo
Faculty of Electrical Engineering
Department of Data Science and Artificial Intelligence

Laboratory Exercise 6

Artificial Intelligence

Authors: Benjamin Bandić, Nedim Bečić
Professor: Izudin Džafić

Contents

1 Goal of the Exercise

The goal of this laboratory exercise is to understand and implement the Minimax algorithm, a fundamental concept in adversarial search for two-player games. Students will first build a simple game of Tic-Tac-Toe and then develop an AI opponent that uses Minimax to make optimal decisions. The core of the exercise will be to enhance the basic Minimax algorithm by implementing Alpha-Beta pruning, a crucial optimization technique that significantly improves the AI's performance by reducing the number of nodes evaluated in the game tree.

By the end of this exercise, students will have a practical understanding of game theory in AI, tree-based search algorithms, and the importance of optimization in developing intelligent agents for games.

2 Theoretical Introduction to Adversarial Search

2.1 The Minimax Algorithm

Minimax is a decision-making algorithm used in two-player, zero-sum games where players have perfect information (like Tic-Tac-Toe, Chess, Checkers). The algorithm explores a tree of all possible moves to determine the optimal move for the current player. It assumes that both players play perfectly.

The two players are typically referred to as the **Maximizer** and the **Minimizer**.

- **Maximizer (MAX):** This player (our AI) tries to achieve the highest possible score.
- **Minimizer (MIN):** This player (the opponent) tries to achieve the lowest possible score.

The algorithm works by performing a depth-first traversal of the game tree. At each node, it makes a decision based on the values of its children nodes.

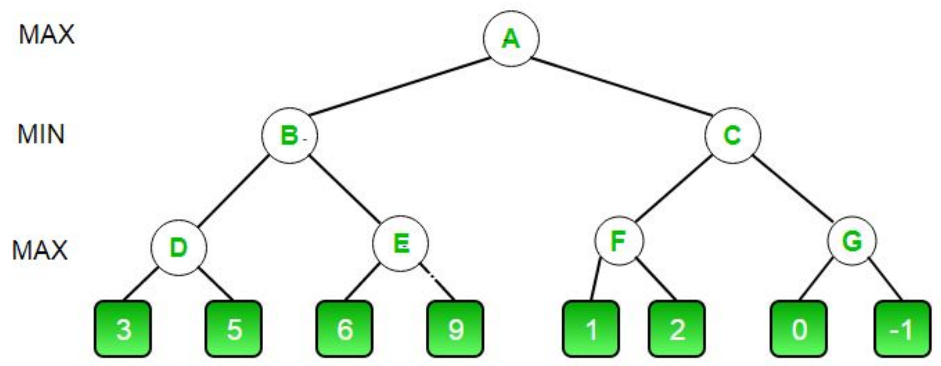
- **MAX nodes** choose the child with the maximum value.
- **MIN nodes** choose the child with the minimum value.

This process is repeated recursively until it reaches terminal nodes (game over states), which are assigned a utility value (e.g., +1 for an AI win, -1 for a loss, 0 for a draw). These values are then propagated back up the tree to determine the best move at the root.

2.2 Alpha-Beta Pruning

The primary drawback of Minimax is that it must explore the entire game tree, which is computationally expensive for complex games. Alpha-Beta pruning is an optimization technique that reduces the number of nodes the Minimax algorithm needs to evaluate. It does this by "pruning" branches of the search tree that cannot possibly influence the final decision.

The algorithm maintains two values:



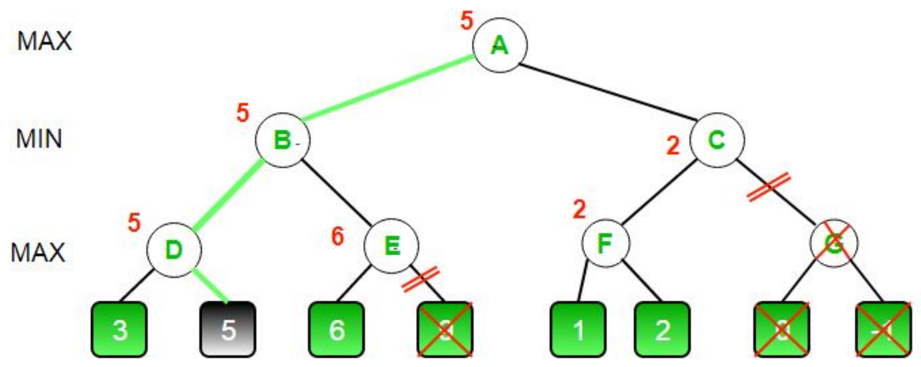
A simple game tree illustrating the Minimax algorithm. The MAX player (at the top) wants to choose a move that leads to the highest possible value, while the MIN player tries to choose moves that lead to the lowest value. The algorithm explores the tree to determine the optimal move from the root.

- **Alpha (α):** The best value (highest score) that the Maximizer can currently guarantee at that level or above. It is initialized to $-\infty$.
- **Beta (β):** The best value (lowest score) that the Minimizer can currently guarantee at that level or above. It is initialized to $+\infty$.

The Pruning Condition: Pruning occurs when at any node, **alpha becomes greater than or equal to beta** ($\alpha \geq \beta$).

- When evaluating a **MIN node**, if its value becomes less than or equal to the alpha value of its parent (MAX node), the MAX player is already guaranteed a better option from a previously explored branch. Thus, the remaining children of this MIN node do not need to be explored. This is a **beta-cutoff**.
- When evaluating a **MAX node**, if its value becomes greater than or equal to the beta value of its parent (MIN node), the MIN player is already guaranteed a better option. Thus, the remaining children of this MAX node are pruned. This is an **alpha-cutoff**.

By effectively pruning away irrelevant branches, Alpha-Beta pruning can significantly speed up the search without affecting the final outcome.



An example of Alpha-Beta Pruning. After the left subtree is evaluated, the MAX player at node A knows it can achieve a score of at least 5. When evaluating the right subtree, the algorithm finds that node F has a value of 2. Since the MIN player at node C would choose 2 (which is less than 5), the MAX player at A will never choose to go right. Therefore, the entire branch under node G is "pruned" and never evaluated, saving significant computation.

3 Project Structure

For this exercise, you will create a console-based Tic-Tac-Toe game. The project will be structured into several classes to separate the game logic from the AI implementation.

3.1 `GameState.h`: The Game Logic

This class represents the state of the Tic-Tac-Toe board. It is responsible for:

- Storing the board (e.g., a 3x3 character array).
- Applying a move to the board.
- Checking for a win, loss, or draw (terminal states).
- Generating a list of legal moves available from the current state.
- Displaying the board to the console.

```cpp
#pragma once
#include <vector>
#include <iostream>

const char PLAYER_X = 'X';
const char PLAYER_O = 'O';
const char EMPTY_CELL = ' ';

class GameState {
private:
    char board[3][3];

public:
    GameState();
    void printBoard() const;
    bool makeMove(int row, int col, char player);
    std::vector<std::pair<int, int>> getAvailableMoves() const;
    char checkWinner() const; // Returns 'X', 'O', 'D' (Draw), or ' ' (
   Ongoing)
```

```
    char getCurrentPlayer();
};
```

A skeleton for the GameState class.

3.2 `AIPlayer.h`: The AI Opponent

This is the core class where the search algorithms will be implemented. It will contain:

- A method to find the best move given a game state.
- A recursive implementation of the Minimax algorithm.
- An optimized version of the Minimax algorithm using Alpha-Beta pruning.

```
#pragma once
#include  GameState.h

struct Move {
    int row = -1;
    int col = -1;
    int score = 0;
};

class AIPlayer {
public:
    AIPlayer(char aiMarker);
    Move findBestMove(GameState& state);

private:
    char aiMarker;
    char opponentMarker;
    Move minimax(GameState& state, char player);
    Move minimax_alpha_beta(GameState& state, char player, int alpha,
   int beta);
};
```

A skeleton for the AIPlayer class.

3.3 `main.cpp`: The Game Loop

This file serves as the entry point of your application. It will contain the main game loop, which:

- Initializes the game state.
- Alternates turns between the human player and the AI.
- Takes input from the human player.
- Calls the AI player to calculate its move.
- Checks for the end of the game and declares the winner.

4 Main Tasks: Implementing Tic-Tac-Toe with AI

4.1 Task 1: Implement the Game Logic

Complete the `GameState` class.

- **Constructor:** Initialize the 3x3 board with empty characters.
- `printBoard()`: Write a function that prints the current board state to the console in a user-friendly format.
- `makeMove()`: Implement the logic to place a player's mark ('X' or 'O') on the board at a given row and column. Ensure the move is legal (the cell is empty).
- `getAvailableMoves()`: This function should return a vector of pairs, where each pair represents the (row, col) coordinates of an empty cell.
- `checkWinner()`: This is a critical function. It must check all rows, columns, and diagonals to see if a player has won. If the board is full and there's no winner, it should declare a draw. Return 'X' or 'O' for a win, 'D' for a draw, and a space ' ' if the game is still in progress.

4.2 Task 2: Implement the Minimax Algorithm

In the `AIPlayer` class, implement the `minimax` function. This function should be recursive.

- **Base Case:** The recursion stops when it reaches a terminal state (win, loss, or draw). In this case, it should return a score: +10 for an AI win, -10 for a human win, and 0 for a draw.
- **Recursive Step:**
  - Get a list of all available moves on the current board.
  - Loop through each available move.
  - For each move, create a new `GameState` by applying that move.
  - Recursively call `minimax` on this new state for the other player.
  - If the current player is the AI (Maximizer), find the move that leads to the maximum score and return it.
  - If the current player is the Human (Minimizer), find the move that leads to the minimum score and return it.
- The `findBestMove` function will initiate the minimax process for the AI's turn.

4.3 Task 3: Implement Alpha-Beta Pruning

Create a new function, `minimax_alpha_beta`, by modifying your original minimax implementation.

- Add two parameters to the function signature: `int alpha` and `int beta`.
- **Maximizer's Turn:**

  - After each recursive call, update alpha: `alpha = max(alpha, returned_score)`.
  - Add the pruning condition: `if (alpha >= beta) break;`. This will stop evaluating further moves from this node.

- **Minimizer's Turn:**
  - After each recursive call, update beta: `beta = min(beta, returned_score)`.
  - Add the pruning condition: `if (alpha >= beta) break;`.
- Modify `findBestMove` to call this new function, initializing alpha to a very small number (`-INFINITY`) and beta to a very large number (`+INFINITY`).

5 Bonus Task: Visualizing the Search Tree

Use the `natId` C++ framework to create a graphical visualization of the Alpha-Beta search tree for a given Tic-Tac-Toe game state. This will provide a clear, intuitive understanding of how the algorithm explores possibilities and where pruning occurs.

- **Task 1: Create a Data Structure for the Tree**
  - Before implementing the visualization, modify your recursive `minimax_alpha_beta` function to not just return a score, but to build a tree data structure that mirrors the search process.
  - Define a `SearchNode` struct or class that stores: the board state, the resulting score, the alpha and beta values at that point, and a vector of child `SearchNode` pointers. Also include a boolean flag, `isPruned`, to mark subtrees that were not explored.
- **Task 2: Create a `TreeCanvas` for Drawing**
  - Implement a `TreeCanvas` class that inherits from `gui::Canvas`.
  - This class will hold the root of the search tree generated by your AI.
  - The main logic will reside in the overridden `onDraw(const gui::Rect& rect)` method. This method will be responsible for recursively traversing your `SearchNode` structure and drawing it.
- **Task 3: Implement Tree Rendering Logic**
  - Inside `onDraw`, write a helper function, e.g., `drawNode(SearchNode* node, td::Point pos)`, that renders a single node.
  - **Draw the Board:** Use a 3x3 grid of `gui::Shape` rectangles to represent the board. Use `gui::DrawableString` to draw the 'X' and 'O' marks inside the cells.
  - **Draw Connections:** Draw lines (`gui::Shape::createLine`) from the parent node's position to each child's position.
  - **Display Info:** Use `gui::DrawableString` to display the node's calculated score, as well as the alpha and beta values, next to the board.

  - **Visualize Pruning:** Check the `isPruned` flag of each node. If it is true, draw that node and all its descendants in a different color (e.g., `td::ColorID::LightGray`) to clearly show which parts of the search tree were ignored by the alpha-beta optimization.

- **Task 4: Integrate into an Application**
  - Create a main view that contains your `TreeCanvas`.
  - Add a button labeled "Visualize AI Move". When clicked, this button should:
    1. Run the `minimax_alpha_beta` search, which now generates the complete search tree.
    2. Pass the root of this tree to your `TreeCanvas` instance.
    3. Call `reDraw()` on the canvas to trigger the visualization.