Student name: Bibi Sabnam Chattoo

Module: Introduction to Game AI

Cohort: DGD C1

Program: Diploma in Game Development

Lecturer: Abdallah Ousman Peerally

Introduction

In this project, I developed a **Tic Tac Toe game** in C featuring both human and AI players. The game is built on a **rule-based system** that enforces valid moves, win conditions, and turn-taking. To enhance AI decision-making, I implemented a **heuristic evaluation function** to score board states. The AI uses the **Minimax algorithm with Alpha-Beta pruning** to select optimal moves efficiently. This approach combines classic gameplay with foundational AI techniques for intelligent, real-time play.

1. RULE-BASED SYSTEM **– Where the game follows strict rules**

A **rule-based game** is controlled entirely by **if-else conditions, win checks, and constraints** that define valid game states and player actions.

Rule: Only place on empty squares

```
if(choice >= 1 && choice <= 9 && square[choice] == choice + '0')
    square[choice] = mark;
```

This ensures the player can't overwrite existing marks — a **rule of the game**.

Rule: Turns alternate between Player 1 and AI

```
player = (player % 2) ? 1 : 2;
```

This controls **whose turn it is** based on simple logic.

Rule: Game ends only if win or draw

```
i = checkWin();
...
while(i == -1);
```

Keeps the game **looping until a win or draw occurs**, based on **logical rules**.

Rule: Win conditions hardcoded

```
if(square[1] == square[2] && square[2] == square[3]) return 1;
```

This is part of the checkWin() function, which uses **exact conditions** to determine a win — a **rule** that defines game outcome.

2.

Heuristic 1: Win or Lose Detection

This part checks if someone has already won.

- If the AI (O) is winning → it gives **+10 points**.
- If the player (X) is winning → it gives **-10 points**.
- If no one is winning yet → it gives **0 points**.

This helps the AI know when a move leads to a win or loss.

Heuristic 2: Position Advantage

This part checks **how smart a move is** even if it doesn't win immediately.
It gives points based on:

- **+3** for the center position (because it controls most lines)
- **+2** for corners (strategically powerful)
- **+1** for edges (less valuable)
- Negative points if the player is in those spots (to block them)

This helps the AI choose positions that give it an advantage later.

Together we get this;

```
evaluate(board) = evaluateWinLoss(board) + evaluatePosition(board);
```

The AI adds both scores together and chooses the move with the **highest total score**.

In simple words; My AI first checks if any move can win or block a loss, and then it also looks at which spots are the smartest to play (like the center or corners). By combining these two strategies, the AI plays smarter and not randomly.

3. ALPHA-BETA PRUNING – **Optimization inside Minimax**

Alpha-Beta pruning improves the efficiency of **Minimax** by skipping branches of the decision tree that don't need to be explored.

Function definition showing alpha and beta:

```
int minimax(char b[10], int depth, int isMax, int alpha, int beta)
```

Alpha = Best score the maximizer (AI) can guarantee
Beta = Best score the minimizer (Player) can guarantee

Alpha updated during maximization (AI's turn):

```
alpha = (alpha > best) ? alpha : best;
```

Beta updated during minimization (Player's turn):

```
beta = (beta < best) ? beta : best;
```

Pruning condition:

```
if(beta <= alpha) break;
```

If a future branch can't improve the current outcome, **the function breaks early**, skipping deeper evaluation — this is **pruning**.

Summary Table

Concept	Code Line Example	Meaning
Rule-Based	if(choice >= 1 && square[choice] == choice + '0')	Only allow valid moves
	checkWin()	Explicit rules for victory
	function	
	player = (player % 2) ? 1 : 2;	Rules for turn alternation
Heuristic	int evaluate(char b[10]) { ... return 10 / -10 / 0; }	Scoring function for game state
	int score = evaluate(b);	AI makes choices based on this value
Alpha-Beta	int minimax(..., int alpha, int beta)	Passing alpha and beta limits
	alpha = max(alpha, best); beta = min(beta, best);	Updating limits
	if(beta <= alpha) break;	Cut off search when outcome is already worse