CS7IS2 - Artificial Intelligence Assignment 3 Report

Kaaviya Paranji Ramkumar

April 2025

1 Introduction

In this assignment, I've explored the fascinating world of game-playing AI algorithms by implementing and comparing different approaches for two classic board games: Tic Tac Toe and Connect 4. The central goal was to understand how different AI strategies perform across games of varying complexity and to analyze the trade-offs between different approaches.

Algorithms implemented

- **Minimax** is a decision-making algorithm for turn-based games that recursively evaluates all possible future game states. It assumes both players play optimally - the maximizing player (the AI) tries to maximize its score, while the minimizing player (opponent) tries to minimize the AI's score. By exploring the game tree, Minimax can select the move that leads to the best possible outcome assuming optimal play.
- **Alpha-beta pruning** is an optimization of the Minimax algorithm that significantly reduces the number of nodes that need to be evaluated. It works by eliminating branches of the search tree that cannot possibly influence the final decision. This allows for deeper searches in the same amount of time without affecting the final choice of move.
- **Q-Learning** is a model-free reinforcement learning algorithm that learns an optimal policy by estimating the value of state-action pairs. The algorithm builds a Q-table mapping states and actions to expected rewards, learning through trial and error. Over time, it discovers which actions lead to higher rewards in each state, without needing to know the rules of the game in advance.

2 Implementation

2.1 Code organization and object-oriented design

The experimental framework for this assignment was carefully planned and implemented using object-oriented programming principles to ensure modularity, re-usability, and clean separation of concerns. The code structure follows software engineering best practices, making it easy to extend and maintain.

The code base for maze generation and solving is currently maintained at *https://github.com/prkaaviya/TCD-Artificial-Intelligence-A3*

The project is organized in a hierarchical structure with clearly defined components like:

```
.
|-- agents/
|   |-- default_agent.py    # Default opponent (better than random)
|   |-- minimax_agent.py    # Minimax implementation
|   `-- qlearning_agent.py  # Q-learning implementation
|-- evaluation/
|   `-- evaluator.py       # Game evaluator functions to evaluate agents
|-- games/
```

```
|   |-- connect4.py         # Connect 4 implementation
|   `-- tic_tac_toe.py      # Tic Tac Toe implementation
|-- main.py                 # Main entry point
|-- results/                # Directory for storing results
`-- utils/
    `-- evaluation.py
```

2.2 Game environment

2.2.1 About Tic Tac Toe

For Tic Tac Toe, a straightforward game environment was implemented using a 3×3 NumPy array to represent the board state. The implementation includes:

- **State representation**: The board is represented as a 3×3 matrix where:
  - 0 represents an empty cell
  - 1 represents player 1's mark (X)
  - 2 represents player 2's mark (O)
- **Action space**: Actions are defined as (row, column) tuples, indicating where a player places their mark. With a 3×3 grid, there are initially 9 possible actions, decreasing as the game progresses.
- **Check valid action**: The game tracks available actions and verifies that a selected action corresponds to an empty cell.
- **Detect terminal state**: After each move, the game checks if a terminal state has been reached by:
  - Examining all rows, columns, and both diagonals for three matching marks in a row
  - Checking if the board is full (this represents draw condition)

The simplicity of Tic Tac Toe made the implementation straightforward, with minimal complexity in state management and win condition checking.

2.2.2 About Connect 4

Connect 4 required a more complex implementation to handle its larger board and the gravity mechanic:

- **State representation**: The board is represented as a 6×7 matrix (6 rows, 7 columns) using the same numerical encoding for players:
  - 0 for empty cells
  - 1 for player 1's discs (Red)
  - 2 for player 2's discs (Yellow)
- **Action space**: Unlike Tic Tac Toe, actions in Connect 4 are simply column indices (0-6) where a player drops their disc. The game handles the "gravity" by determining the lowest empty row in the chosen column.
- **Column tracking**: To efficiently determine valid moves and handle the gravity mechanic, the game maintains an array tracking the current height of each column.
- **Detect terminal state**: After each move, the game checks for terminal states by:
  - Examining only the lines affected by the most recent move (horizontal, vertical, and both diagonals)
  - This targeted checking is much more efficient than examining the entire board
- **Optimization**: Rather than checking the entire board after each move, the implementation only examines lines that could potentially contain a winning sequence including the newly placed piece.

2.2.3 Challenges in state and action representation

1. **Size of state space**
   - Tic Tac Toe: Small and manageable ($\sim$5,478 possible states)
   - Connect 4: Enormous ($\sim$4.5 trillion possible states)
2. **Action representation**
   - Tic Tac Toe: Direct (row, column) coordinates
   - Connect 4: Column selection only, with row determined by gravity
3. **Win condition complexity**
   - Tic Tac Toe: Checking 8 possible lines (3 rows, 3 columns, 2 diagonals)
   - Connect 4: Checking many more potential lines, optimized by only examining those affected by the latest move

These implementation differences directly impact the performance of the various AI algorithms, particularly when scaling from the simpler Tic Tac Toe to the more complex Connect 4 environment.

2.3 Implementations of different agents

2.3.1 Default agent

For the baseline opponent, a Default Agent was implemented that exhibits simple but strategic behavior. This agent follows a straightforward priority-based decision-making process:

- **Strategy**: The agent makes decisions based on three simple rules:
  1. If there exists a move that would result in an immediate win, take it
  2. If the opponent has a move that would result in their immediate win, block it
  3. Otherwise, select a random valid move
- **Implementation**: For each possible action, the agent simulates the result and checks if it leads to a terminal state. The implementation creates a copy of the game state to test moves without modifying the actual game.
- **Adaptability**: The same agent works for both Tic Tac Toe and Connect 4 without game-specific modifications, adapting to the different action spaces automatically.

This agent satisfies the requirement of being "better than random" while still being simple enough to test our more sophisticated algorithms against.

2.3.2 Minimax agent

The Minimax algorithm forms the foundation of traditional game-playing AI by recursively exploring the game tree to find optimal moves.

- **Core algorithm**: Minimax works by:
  - Recursively evaluating all possible future game states
  - Assuming that the opponent will play optimally
  - Alternating between maximizing the player's score and minimizing the opponent's score
  - Selecting the move that leads to the best possible outcome
- **Implementation details**

- Separate functions for maximizing and minimizing players
- Terminal state evaluation returning +1 for wins, -1 for losses, and 0 for draws
- Performance tracking for nodes visited and execution time

- **Depth limitation**: For Connect 4, depth-limited search was included to mitigate issues with scalability due to large state space:
  - Configurable maximum depth parameter to limit recursion
  - Evaluation function for non-terminal states when the depth limit is reached
- **Heuristic functions**: For depth-limited search in Connect 4, heuristic functions was designed to evaluate non-terminal states:
  - Evaluating all possible winning lines (horizontal, vertical, diagonal)
  - Assigning higher scores to positions with more of the player's pieces in a row
  - Considering both offensive potential (forming own lines) and defensive concerns (blocking opponent)

2.3.3 Minimax with Alpha-Beta pruning agent

Alpha-Beta pruning enhances the Minimax algorithm by eliminating branches that cannot influence the final decision.

- **Optimization principle**: Alpha-Beta works by:
  - Maintaining two values: alpha (best already found for maximizer) and beta (best already found for minimizer)
  - Pruning branches where it's proven that better options exist elsewhere in the tree
  - Producing identical decisions to regular Minimax, just more efficiently
- **Implementation details**
  - Modified versions of the max and min functions that track alpha and beta values
  - Early termination of branch exploration when a cutoff is detected
  - Same depth limitation and heuristic functions as the regular Minimax
- **Efficiency gains**: The implementation includes metrics to measure:
  - Number of nodes explored compared to regular Minimax
  - Execution time improvements
  - Decision quality validation (ensuring identical choices to regular Minimax)

2.3.4 Q-Learning agent

The Q-Learning agent takes a completely different approach, learning from experience rather than searching through future states.

- **Learning approach**
  - Builds a Q-table mapping state-action pairs to expected rewards
  - Updates Q-values using the formula: $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
  - Where α is the learning rate, γ is the discount factor, r is the reward, s is the current state, a is the action, and s' is the resulting state
- **State representation**

- For Tic Tac Toe: Direct representation of the 3×3 board as tuples
  - For Connect 4: Compact representation tracking pieces in each column to reduce state space

- **Exploration-exploitation balance**
  - Epsilon-greedy strategy with a decaying exploration rate
  - Initial exploration rate of 0.3, decaying by factor of 0.995 after each episode
  - Minimum exploration rate of 0.01 to ensure some exploration continues

- **Implementation features**
  - Separate training and exploitation modes
  - Persistent Q-table storage to save and load learned policies
  - Reward shaping to encourage winning (+1), avoid losing (-1), and slightly prefer draws over uncertain states (+0.2)

2.4 Evaluation Framework

To systematically compare the performance of the different agents, an evaluation system was implemented to completely automate the multiple playthroughs of the games between all agents:

- **Game evaluator structure**
  - Round-robin format where each agent plays against all others
  - Symmetric games (each agent plays as both first and second player)
  - Configurable number of games per evaluation

- **Performance metrics**
  - Win/loss/draw rates for each agent
  - Average game length (number of moves)
  - Execution time per move and per game
  - Nodes visited (for Minimax agents)

- **Visualization and analysis tools**
  - Bar charts comparing agent performance
  - CSV export for detailed statistical analysis
  - Detailed printout of each game results

- **Experimental controls**
  - Same initial conditions for all games
  - Independent agent instances for each game to prevent learning during evaluation
  - Special depth testing mode to analyze the effect of search depth on performance

3 Experimental Results and Analysis

In this section, I present the results of my experimental evaluation, examining the performance of each algorithm across both Tic Tac Toe and Connect 4 games. The analysis focuses on effectiveness (win rates), efficiency (nodes visited and execution time), and the relative strengths and weaknesses of each approach.

3.1 Setup and methodology

To ensure a comprehensive and fair comparison, I conducted evaluation playthrough with multiple games between all possible agent combinations. For each game environment:

- 100 games were played for each playthrough in Tic Tac Toe and Connect 4 (1,200 games total each game)
- Each algorithm played both as first player and second player
- All algorithms faced off against each other and the default agent
- Performance metrics (win rates, execution times, nodes visited) were recorded for each game

The Q-Learning agent was trained separately before evaluation, with 5,000 episodes for Tic Tac Toe and 50,000 episodes for Connect 4. This disparity in training episodes is because of much larger state space needed in Connect 4.

For Connect 4, I first confirmed the computational infeasibility of unlimited-depth Minimax by attempting to run it without depth limitation. As expected, the algorithm could not complete even a single move in a reasonable timeframe (30 minutes), demonstrating the necessity of depth-limited search for larger game spaces.

The commands used to execute these experiments were:

```
# For Connect 4 depth testing
python main.py --game connect4 --mode depth_test --num_games 3

# For Q-Learning training
python main.py --game tictactoe --mode train --train_episodes 5000
python main.py --game connect4 --mode train --train_episodes 50000

# For Tic Tac Toe games
python main.py --game tictactoe --mode evaluator --num_games 100 --load_qtable

# For Connect 4 games
python main.py --game connect4 --mode evaluator --num_games 1000 --load_qtable
```

3.2 Alpha-Beta pruning efficiency

The application of alpha-beta pruning during my experiments provided remarkable improvements which enhanced computational performance. Alpha-beta pruning leads to significant performance improvements across both game types although they manifest in varying degrees (Figure 1).

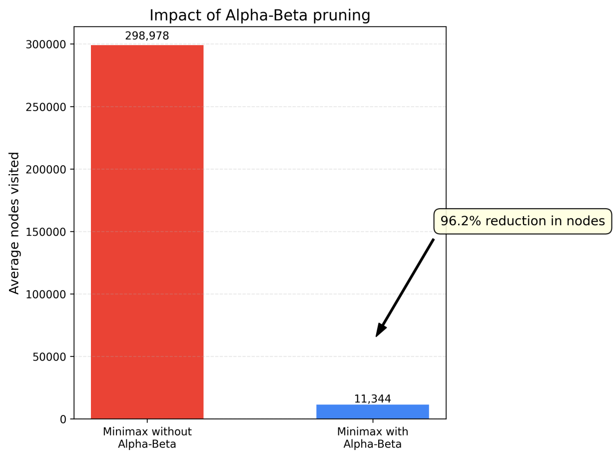
The Tic Tac Toe utilizes normal Minimax to analyze 298,978 nodes but alpha-beta pruning decreases this number to only 11,344 nodes which represents an impressive 96.2% reduction in nodes. The advantage in execution time directly mirrors this reduction resulting in 2.21 second game execution time reduction to just 0.08 seconds that represents a total 96.4% improvement.

Alpha-beta pruning reduced the node count for Connect 4 from 1,976 to 1,044 and cut execution time per game to 0.27 seconds from 0.54 seconds based on a depth limit of 3.

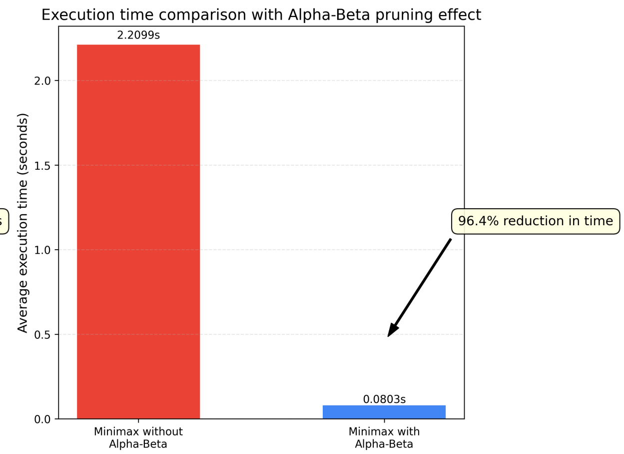
The shallow node exploration in Connect 4 detection accounts for the lower percentage of performance enhancement because of its size-bound constraints. Setting the depth limit to 3 restricts the search space severely thus creating fewer chances for optimizing the search. Even though Connect 4 improvements show fewer percentage changes the absolute gains from pruning strategies still stand substantial.

3.3 Algorithm performance analysis

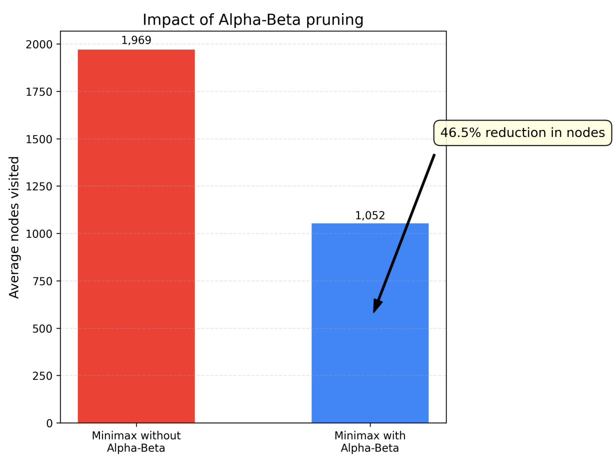
Table 1 presents the overall performance metrics for all agents in both games. Several interesting patterns emerge from these results.



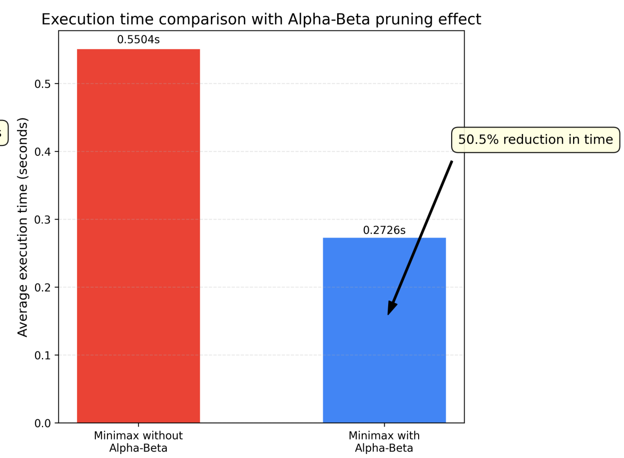
(a) Nodes visited in Tic Tac Toe



(b) Execution time in Tic Tac Toe



(c) Nodes visited in Connect 4



(d) Execution time in Connect 4

Figure 1: Impact of alpha-beta pruning on computational efficiency in both games

3.3.1 Tic Tac Toe performance

The Default Agent achieved 53.17% wins overall in Tic Tac Toe while both Minimax variants maintained similar levels at 50% and the Q-Learning Agent won at 43%.

Tic Tac Toe presents an evident advantage to the first player according to analysis of game results. All games played by both Minimax variants resulted either in a victory or draw when the agents played as the first player.

Minimax techniques maintain equivalent decision quality through identical move selection and virtually the same winning performance though they execute at highly differing computational speeds. Alpha-beta pruning maintains the decision quality when the algorithm performs calculations and removes unnecessary processing steps from the solution finding process.

The Q-Learning Agent demonstrates a winning percentage of 43% through rapid learning during 5,000 training episodes although it takes less than 0.0001 seconds to process each move. Minimax delivers optimal moves through complete searches yet Q-learning achieves effective strategies solely from experiences demonstrating reinforcement learning potential even when training amounts are restricted.

The Default Agent demonstrates strong performance in Tic Tac Toe due to the game's easy nature which allows simple strategies like winning directly and stopping opponent wins to prove successful.

(a) Overall performance metrics for Tic Tac Toe (1200 games in all)

Algorithm	Win rate	Draw rate	Avg. moves	Avg. time (s)
Default Agent	53.17%	3.67%	5.95	0.0002
Minimax without Alpha-Beta	50.17%	0.17%	5.12	2.2264
Minimax with Alpha-Beta	49.83%	0.17%	5.12	0.0809
Q-Learning Agent	43.00%	3.67%	6.10	<0.0001

(b) Overall performance metrics for Connect 4 (1200 games in all with depth limit = 3)

Algorithm	Win rate	Draw rate	Avg. moves	Avg. time (s)
Default Agent	61.50%	0.00%	13.74	0.0007
Minimax without Alpha-Beta	49.83%	0.00%	10.65	0.5504
Minimax with Alpha-Beta	49.83%	0.00%	10.72	0.2726
Q-Learning Agent	38.33%	0.00%	17.39	0.0001

Table 1: Performance comparison of all algorithms across both game environments

3.3.2 Connect 4 performance

The algorithms in Connect 4 generate wider performance changes among each other compared to Tic Tac Toe. The Default Agent surpasses the Tic Tac Toe performance by winning 61.50% of its matches but the Q-Learning Agent obtains a win rate of only 38.33%.

The growing difference between the agents stands out because Connect 4 introduces enhanced levels of complexity to the board game. The simple heuristic approach used by the Default Agent generates better performance because it can overcome opponents with shallow search limits and opponents with incomplete learning pipelines.

Minimax agents perform inadequately with a depth limit of 3 because they cannot maintain complete visibility to the end of the game as Minimax without depth limitations does in Tic Tac Toe. The heuristic evaluation function assists depth frontier decision-making but fails to replace the deficiencies brought by restricted lookahead capabilities.

The Q-Learning Agent achieves fewer optimal policy solutions in Connect 4 rather than Tic Tac Toe mainly due to the game's broader state space that requires extra training attempts to develop a complete policy. Despite receiving 50,000 training episodes which amounted to ten times the number provided to the Tic Tac Toe agent the Q-Learning agent recorded a lower win rate suggesting additional training sessions or better state analysis would help larger board games.

3.4 Scalability analysis

The performance gap between Tic Tac Toe algorithms and Connect 4 algorithms demonstrates essential scalability factors to consider. Minimax algorithms proved to be computationally impossible when executed without depth constraints since they failed to finish one move within thirty minutes because of their massive state space.

The application of depth-limited search in combination with heuristic evaluation function became necessary for playing Connect 4. A depth limit of 3 enabled Minimax execution yet required it to sacrifice its ability for perfect playing. Due to its depth limitation the algorithm keeps a look ahead of just three moves that may not let it detect winning strategies that need extended move sequences.

The Q-Learning agent encountered scalability problems when applied to the Connect 4 game. The complexity of the game required a compacted state representation to ensure learning success although the program performed 50,000 training episodes still failed to grasp complete tactical mastery.

Testing different depth limits for Connect 4 revealed:

- Depth 1: 44 nodes, 0.015s - Very fast but strategically weak
- Depth 2: 449 nodes, 0.121s - Reasonable compromise for fast play

- Depth 3: 1,111 nodes, 0.290s - Chosen for our experiments as a good balance
- Depth 4: 5,866 nodes, 1.430s - Better play but significantly slower
- Depth 5: 11,618 nodes, 2.861s - Strong play but prohibitively slow for evaluation
- Unlimited: Computationally infeasible (¿30 minutes per move)

These scalability observations highlight a fundamental trade-off: as game complexity increases, algorithms must sacrifice either completeness (Minimax with depth limits) or optimality (Q-Learning with limited training) to remain computationally feasible.

4 Conclusion

I gained intriguing understanding about how strategic AI methods handle difficulties through this assignment of game-playing algorithms. The use of Alpha-beta pruning sharply decreased computational workload by more than 90 percent during Tic Tac Toe gameplay maintaining absolute decision making to demonstrate the value of intelligent approach choice over excess work persistence. The three algorithms demonstrated their unique capabilities during Connect 4 game play so that Minimax delivered optimal results even though it proved difficult to scale up while Q-learning provided adaptability despite its minimal execution time once trained and the basic Default Agent surprisingly outperformed its more advanced counterparts. The conclusions established a fundamental AI dilemma that exists throughout computer systems where performance speed must be weighed against the caliber of choices made in situations where resources are restricted and problems remain complex.

Appendices

A Default agent implementation code

```python
class DefaultAgent:
    """
    A default opponent for game playing.

    This agent is better than random but still simple:
    1. If there's a winning move, take it
    2. If opponent has a winning move, block it
    3. Otherwise, make a random valid move
    """

    def __init__(self, player_id: int):
        """
        Initialize the agent.

        Args:
            player_id: The ID of the player (1 or 2)
        """
        self.player_id = player_id
        self.opponent_id = 3 - player_id  # if player_id is 1, opponent is 2 and vice versa
        self.name = "Default Agent"

    def select_action(self, game, training: bool = False) -> Union[Tuple[int, int], int]:
        """
        Select an action based on the current game state.

        Args:
            game: The game object (TicTacToe or Connect4)

        Returns:
            action: For TicTacToe: (row, col) tuple, for Connect4: column index
        """
        valid_actions = game.get_valid_actions()
        if not valid_actions:
            raise ValueError("No valid actions available")

        # check if there's a winning move
        for action in valid_actions:
            # make a copy of the game to simulate moves
            game_copy = self._copy_game(game)

            # simulate taking the action
            state, _, done, info = game_copy.step(action)

            # if the game is over and agent won, take this action
            if done and info["winner"] == self.player_id:
                return action

        # check if the opponent has a winning move and block it
        # simulate opponent's turn first
        for action in valid_actions:
            game_copy = self._copy_game(game)

            # change current player to opponent to simulate their move
            game_copy.current_player = self.opponent_id

            # simulate opponent taking the action
            state, _, done, info = game_copy.step(action)

            # if the game would be over and opponent would win, block this action
            if done and info["winner"] == self.opponent_id:
                return action

```

```python
        # otherwise make a random valid move
        return random.choice(valid_actions)

    def _copy_game(self, game):
        """Create a deep copy of the game to simulate moves."""
        if hasattr(game, 'rows') and hasattr(game, 'cols'):
            game_copy = type(game)()
            game_copy.board = game.board.copy()
            game_copy.current_player = game.current_player
            game_copy.done = game.done
            game_copy.winner = game.winner
            game_copy.column_heights = game.column_heights.copy()
            return game_copy
        else:
            game_copy = type(game)()
            game_copy.board = game.board.copy()
            game_copy.current_player = game.current_player
            game_copy.done = game.done
            game_copy.winner = game.winner
            game_copy.valid_actions = game.valid_actions.copy() \
                if hasattr(game, 'valid_actions') else []
            return game_copy
```

B Minimax agent implementation code

```python
import numpy as np
import time
from typing import Tuple, List, Dict, Any, Union, Optional

class MinimaxAgent:
    """
    Minimax agent with optional alpha-beta pruning.

    This agent uses the minimax algorithm to select the best action.
    It can be configured to use alpha-beta pruning for efficiency.
    For Connect4, a depth limit and heuristic evaluation function are used.
    """

    def __init__(self, player_id: int, use_alpha_beta: bool = True,
                 max_depth: Optional[int] = None):
        """
        Initialize the Minimax agent.

        Args:
            player_id: The ID of the player (1 or 2)
            use_alpha_beta: Whether to use alpha-beta pruning
            max_depth: Maximum depth to search (None for unlimited)
        """
        self.player_id = player_id
        self.opponent_id = 3 - player_id
        self.use_alpha_beta = use_alpha_beta
        self.max_depth = max_depth
        self.name = f"Minimax {'with' if use_alpha_beta else 'without'} Alpha-Beta"
        if max_depth:
            self.name += f" (Depth {max_depth})"

        self.nodes_visited = 0
        self.execution_time = 0

    def select_action(self, game, training: bool = False) -> Union[Tuple[int, int], int]:
        """
        Select the best action using minimax.

        Args:
            game: The game object

        Returns:
```

```
            The best action
        """
        self.nodes_visited = 0
        start_time = time.time()

        valid_actions = game.get_valid_actions()
        if not valid_actions:
            raise ValueError("No valid actions available")

        # in case agent runs out of time/depth, play a fallback move
        best_action = valid_actions[0]

        if self.use_alpha_beta:
            best_value = float('-inf')
            alpha = float('-inf')
            beta = float('inf')

            for action in valid_actions:
                game_copy = self._copy_game(game)
                _, _, done, info = game_copy.step(action)

                # if this move wins immediately, take it
                if done and info["winner"] == self.player_id:
                    self.execution_time = time.time() - start_time
                    return action

                # otherwise evaluate the move
                if done:
                    value = 0
                else:
                    value = self._min_value(game_copy, 1, alpha, beta)

                if value > best_value:
                    best_value = value
                    best_action = action

                alpha = max(alpha, best_value)
        else:
            best_value = float('-inf')

            for action in valid_actions:
                game_copy = self._copy_game(game)
                _, _, done, info = game_copy.step(action)

                if done and info["winner"] == self.player_id:
                    self.execution_time = time.time() - start_time
                    return action

                if done:
                    value = 0
                else:
                    value = self._min_value_no_pruning(game_copy, 1)

                if value > best_value:
                    best_value = value
                    best_action = action

        self.execution_time = time.time() - start_time
        return best_action

    def _max_value(self, game, depth: int, alpha: float, beta: float) -> float:
        """Maximizing player in minimax with alpha-beta pruning."""
        self.nodes_visited += 1

        # check if we have reached the maximum depth or a terminal state
        if (self.max_depth is not None and depth >= self.max_depth) or game.is_terminal():
            return self._evaluate(game)

```

```python
        value = float('-inf')
        valid_actions = game.get_valid_actions()

        for action in valid_actions:
            game_copy = self._copy_game(game)
            _, _, done, info = game_copy.step(action)

            if done:
                if info["winner"] == self.player_id:
                    child_value = 1.0
                elif info["winner"] == 0:
                    child_value = 0
                else:
                    child_value = -1.0
            else:
                child_value = self._min_value(game_copy, depth + 1, alpha, beta)

            value = max(value, child_value)

            if value >= beta:
                return value  # return if beta cutoff

            alpha = max(alpha, value)

        return value

    def _min_value(self, game, depth: int, alpha: float, beta: float) -> float:
        """Minimizing player in minimax with alpha-beta pruning."""
        self.nodes_visited += 1

        # check if we have reached the maximum depth or a terminal state
        if (self.max_depth is not None and depth >= self.max_depth) or game.is_terminal():
            return self._evaluate(game)

        value = float('inf')
        valid_actions = game.get_valid_actions()

        for action in valid_actions:
            game_copy = self._copy_game(game)
            _, _, done, info = game_copy.step(action)

            if done:
                if info["winner"] == self.opponent_id:
                    child_value = -1.0
                elif info["winner"] == 0:
                    child_value = 0
                else:
                    child_value = 1.0
            else:
                child_value = self._max_value(game_copy, depth + 1, alpha, beta)

            value = min(value, child_value)

            if value <= alpha:
                return value # return if alpha cutoff

            beta = min(beta, value)

        return value

    def _max_value_no_pruning(self, game, depth: int) -> float:
        """Maximizing player in minimax without alpha-beta pruning."""
        self.nodes_visited += 1

        # check if we have reached the maximum depth or a terminal state
        if (self.max_depth is not None and depth >= self.max_depth) or game.is_terminal():
            return self._evaluate(game)

```

```
        value = float('-inf')
        valid_actions = game.get_valid_actions()

        for action in valid_actions:
            game_copy = self._copy_game(game)
            _, _, done, info = game_copy.step(action)

            if done:
                if info["winner"] == self.player_id:
                    child_value = 1.0  # Win
                elif info["winner"] == 0:
                    child_value = 0  # Draw
                else:
                    child_value = -1.0  # Loss
            else:
                child_value = self._min_value_no_pruning(game_copy, depth + 1)

            value = max(value, child_value)

        return value

    def _min_value_no_pruning(self, game, depth: int) -> float:
        """Minimizing player in minimax without alpha-beta pruning."""
        self.nodes_visited += 1

        # check if we have reached the maximum depth or a terminal state
        if (self.max_depth is not None and depth >= self.max_depth) or game.is_terminal():
            return self._evaluate(game)

        value = float('inf')
        valid_actions = game.get_valid_actions()

        for action in valid_actions:
            game_copy = self._copy_game(game)
            _, _, done, info = game_copy.step(action)

            if done:
                if info["winner"] == self.opponent_id:
                    child_value = -1.0  # Loss
                elif info["winner"] == 0:
                    child_value = 0  # Draw
                else:
                    child_value = 1.0  # Win
            else:
                child_value = self._max_value_no_pruning(game_copy, depth + 1)

            value = min(value, child_value)

        return value

    def _evaluate(self, game) -> float:
        """
        Evaluate the current game state.

        For terminal states:
        - Win: +1.0
        - Loss: -1.0
        - Draw: 0.0

        For non-terminal states (when using depth-limited search):
        - Heuristic evaluation based on potential winning lines
        """
        # if the game is over, return the actual outcome
        if game.is_terminal():
            winner = game.get_winner()
            if winner == self.player_id:
                return 1.0  # the agent won
            elif winner == self.opponent_id:
```

```python
                return -1.0  # the agent lost
            else:
                return 0.0  # the game is a draw

        # for depth-limited search, use a custom heuristic evaluation
        if hasattr(game, 'rows') and hasattr(game, 'cols'):
            # heuristic for Connect4
            return self._evaluate_connect4(game)
        else:
            # heuristic for TicTacToe
            return self._evaluate_tictactoe(game)

    def _evaluate_connect4(self, game) -> float:
        """
        Heuristic evaluation for Connect4.
        Counts potential winning lines for both players.
        """
        board = game.board
        score = 0

        # check horizontal, vertical, and both diagonals for wins
        # give higher scores to positions with more of the player's pieces in a row

        # for horizontal
        for row in range(game.rows):
            for col in range(game.cols - 3):
                window = board[row, col:col+4]
                score += self._evaluate_window(window)

        # for vertical
        for col in range(game.cols):
            for row in range(game.rows - 3):
                window = board[row:row+4, col]
                score += self._evaluate_window(window)

        # for diagonal (in positive slope)
        for row in range(game.rows - 3):
            for col in range(game.cols - 3):
                window = [board[row+i, col+i] for i in range(4)]
                score += self._evaluate_window(window)

        # for diagonal (in negative slope)
        for row in range(3, game.rows):
            for col in range(game.cols - 3):
                window = [board[row-i, col+i] for i in range(4)]
                score += self._evaluate_window(window)

        return score

    def _evaluate_window(self, window) -> float:
        """Evaluate a window of 4 positions."""
        player_count = np.sum(window == self.player_id)
        opponent_count = np.sum(window == self.opponent_id)
        empty_count = np.sum(window == 0)

        # if there's a mix of both players' pieces, this window isn't winnable
        if player_count > 0 and opponent_count > 0:
            return 0

        # score based on how many of player's pieces are in the window
        if player_count > 0:
            if player_count == 3 and empty_count == 1:
                return 0.8 # near to winning
            elif player_count == 2 and empty_count == 2:
                return 0.3
            elif player_count == 1 and empty_count == 3:
                return 0.1

```

```
        # score based on how many of opponent's pieces are in the window
        if opponent_count > 0:
            if opponent_count == 3 and empty_count == 1:
                return -0.8  # near to losing
            elif opponent_count == 2 and empty_count == 2:
                return -0.3
            elif opponent_count == 1 and empty_count == 3:
                return -0.1

        return 0

    def _evaluate_tictactoe(self, game) -> float:
        """
        Custom heuristic evaluation for TicTacToe.
        Simpler than Connect4 since the state space is smaller.
        """
        board = game.board
        score = 0

        # check rows, columns, and diagonals for potential wins below

        for row in range(3):
            score += self._evaluate_line(board[row, :])

        for col in range(3):
            score += self._evaluate_line(board[:, col])

        score += self._evaluate_line(np.array([board[0, 0], board[1, 1], board[2, 2]]))
        score += self._evaluate_line(np.array([board[0, 2], board[1, 1], board[2, 0]]))

        return score

    def _evaluate_line(self, line) -> float:
        """Evaluate a line of 3 positions for TicTacToe."""
        player_count = np.sum(line == self.player_id)
        opponent_count = np.sum(line == self.opponent_id)
        empty_count = np.sum(line == 0)

        # if both players have pieces in this line, it can't be won
        if player_count > 0 and opponent_count > 0:
            return 0

        # score based on how many of player's pieces are in the line
        if player_count > 0:
            if player_count == 2 and empty_count == 1:
                return 0.6  # near to winning
            elif player_count == 1 and empty_count == 2:
                return 0.2

        # score based on how many of opponent's pieces are in the line
        if opponent_count > 0:
            if opponent_count == 2 and empty_count == 1:
                return -0.6  # near to losing
            elif opponent_count == 1 and empty_count == 2:
                return -0.2

        return 0

    def _copy_game(self, game):
        """Create a deep copy of the game to simulate moves."""
        if hasattr(game, 'rows') and hasattr(game, 'cols'):
            game_copy = type(game)()
            game_copy.board = game.board.copy()
            game_copy.current_player = game.current_player
            game_copy.done = game.done
            game_copy.winner = game.winner
            game_copy.column_heights = game.column_heights.copy()
            return game_copy
```

```
        else:
            game_copy = type(game)()
            game_copy.board = game.board.copy()
            game_copy.current_player = game.current_player
            game_copy.done = game.done
            game_copy.winner = game.winner
            game_copy.valid_actions = game.valid_actions.copy() \
                if hasattr(game, 'valid_actions') else []
            return game_copy
```

C Qlearning agent implementation code

```
import numpy as np
import random
import pickle
from typing import Tuple, Dict, List, Union, Any
import os

class QLearningAgent:
    """
    Q-learning agent for playing games.

    This agent uses tabular Q-learning to learn a policy.
    State representation is simplified to make learning tractable.
    """

    def __init__(self, player_id: int, learning_rate: float = 0.1, discount_factor: float = 0.9,
                 exploration_rate: float = 0.3, exploration_decay: float = 0.995, min_exploration: float = 0.01,
                 load_qtable: bool = False, save_path: str = None):
        """
        Initialize the Q-learning agent.

        Args:
            player_id: The ID of the player (1 or 2)
            learning_rate: Alpha - learning rate
            discount_factor: Gamma - discount factor for future rewards
            exploration_rate: Epsilon - exploration rate
            exploration_decay: Factor to decay exploration rate after each episode
            min_exploration: Minimum exploration rate
            load_qtable: Whether to load an existing Q-table
            save_path: Path to save/load Q-table
        """
        self.player_id = player_id
        self.opponent_id = 3 - player_id
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.exploration_rate = exploration_rate
        self.exploration_decay = exploration_decay
        self.min_exploration = min_exploration
        self.q_table = {}
        self.name = "Q-Learning Agent"
        self.save_path = save_path or f"q_table_player{player_id}.pkl"

        # keep track of the current game for learning
        self.current_state = None
        self.current_action = None

        if load_qtable and os.path.exists(self.save_path):
            self.load_q_table()

    def state_to_tuple(self, game) -> tuple:
        """
        Convert a game state to a tuple that can be used as a dictionary key.
        Uses a simplified representation to keep the state space manageable.

```

```python
        Args:
            game: The game object

        Returns:
            A tuple representation of the state
        """
        if hasattr(game, 'rows') and hasattr(game, 'cols'):
            # use a more compact representation for Connect4
            # we can't use the raw board as a key because it's too large
            # instead, we can create a tuple of tuples where each inner tuple represents a column
            columns = []
            for col in range(game.cols):
                column_pieces = []
                for row in range(game.rows):
                    if game.board[row, col] != 0:
                        # store (row, player_id) for each piece in the column
                        column_pieces.append((row, int(game.board[row, col])))
                columns.append(tuple(column_pieces))

            # include the current player in the state
            return (tuple(columns), game.current_player)
        else:
            # use a flattened board for TicTacToe
            # convert board to tuple of tuples to hash it
            board_tuple = tuple(tuple(row) for row in game.board)
            return (board_tuple, game.current_player)

    def get_q_value(self, state: tuple, action: Union[Tuple[int, int], int]) -> float:
        """
        Get the Q-value for a state-action pair.

        Args:
            state: The state tuple
            action: The action

        Returns:
            The Q-value
        """
        if state not in self.q_table:
            self.q_table[state] = {}

        # convert action to a hashable type if it's not already
        if isinstance(action, np.ndarray):
            action = tuple(action)

        if action not in self.q_table[state]:
            self.q_table[state][action] = 0.0

        return self.q_table[state][action]

    def update_q_value(self, state: tuple, action: Union[Tuple[int, int], int],
                       reward: float, next_state: tuple, done: bool) -> None:
        """
        Update the Q-value for a state-action pair.

        Args:
            state: The current state tuple
            action: The action taken
            reward: The reward received
            next_state: The resulting state tuple
            done: Whether the episode is done
        """
        # convert action to a hashable type if it's not already
        if isinstance(action, np.ndarray):
            action = tuple(action)

        q_value = self.get_q_value(state, action)
```

```
        # if the episode is done, there is no next state
        # else, get max Q-value for next state
        if done:
            max_next_q = 0
        else:
            if next_state in self.q_table and self.q_table[next_state]:
                max_next_q = max(self.q_table[next_state].values())
            else:
                max_next_q = 0

        # update the Q-learnign formula
        new_q_value = q_value + self.learning_rate * ( \
            reward + self.discount_factor * max_next_q - q_value)

        if state not in self.q_table:
            self.q_table[state] = {}

        self.q_table[state][action] = new_q_value

    def select_action(self, game, training: bool = False) -> Union[Tuple[int, int], int]:
        """
        Select an action based on the current game state.

        Args:
            game: The game object
            training: Whether the agent is in training mode

        Returns:
            The selected action
        """
        state = self.state_to_tuple(game)
        valid_actions = game.get_valid_actions()

        if not valid_actions:
            raise ValueError("No valid actions available")

        # save current state for learning update
        if training:
            self.current_state = state

        # let agent explore with random action
        if training and random.random() < self.exploration_rate:
            action = random.choice(valid_actions)
            if training:
                self.current_action = action
            return action

        # let agent take best known action
        q_values = {action: self.get_q_value(state, action) for action in valid_actions}

        # find actions with the maximum Q-value
        max_q = max(q_values.values()) if q_values else 0
        best_actions = [action for action, q_value in q_values.items() if q_value == max_q]

        # if multiple actions have the same Q-value, choose randomly among them
        action = random.choice(best_actions) if best_actions else random.choice(
    valid_actions)

        if training:
            self.current_action = action

        return action

    def learn(self, game, reward: float, done: bool) -> None:
        """
        Learn from the most recent action.

```

```
        Args:
            game: The game object after the action was taken
            reward: The reward received
            done: Whether the episode is done
        """
        if self.current_state is None or self.current_action is None:
            return

        next_state = self.state_to_tuple(game) if not done else None

        self.update_q_value(self.current_state, self.current_action, reward, next_state, done)

        self.current_state = next_state
        self.current_action = None

        if done:
            self.exploration_rate = max(self.min_exploration,
                                        self.exploration_rate * self.exploration_decay)

    def save_q_table(self, path: str = None) -> None:
        """Save the Q-table to a file."""
        save_path = path or self.save_path
        with open(save_path, 'wb') as f:
            pickle.dump(self.q_table, f)
        print(f"Q-table saved to {save_path}")
        print(f"Q-table size: {len(self.q_table)} states")

    def load_q_table(self, path: str = None) -> None:
        """Load the Q-table from a file."""
        load_path = path or self.save_path
        if os.path.exists(load_path):
            with open(load_path, 'rb') as f:
                self.q_table = pickle.load(f)
            print(f"Q-table loaded from {load_path}")
            print(f"Q-table size: {len(self.q_table)} states")
        else:
            print(f"No Q-table found at {load_path}")

    def reset(self) -> None:
        """Reset agent state for a new episode."""
        self.current_state = None
        self.current_action = None
```