CS7IS2 – Artificial Intelligence – Assignment 3

Sibin Babu George 24335327 MSc Computer Science (Future Networked Systems)

1. INTRODUCTION

In this assignment, I implemented two games viz tic tac toe and connect 4 and also evaluted the performance of different algorithms for playing the above mentioned games. The performance and evaluation is done for these algorithms (Minimax, Minimax_ab, Q-learning) along with random player (player which makes radom moves) and default player (rule based player) is done. Q-learning model is trained with all the players and for each new model, its evaluated against all algorithms (as first player X and also as second player O). This report contains implementation details, different approaches used and comparisons of performance of all algorithms

2. IMPLEMENTATION

A) GAMES

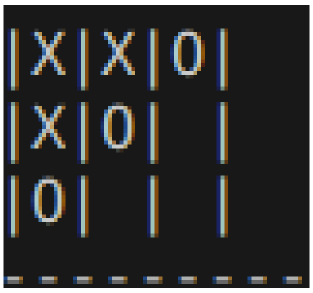
1) Tic Tac Toe Game

TicTacToe is implemented as a 3x3 grid using a flattened list of 9 elements, where each element represents a cell, alternating between two players, "X" and "O", with key features including a board representation as a flat list, a get_legal_moves method returning indices of empty cells, a make_move method placing a marker in the specified cell and switching the current player, a check_winner method checking all possible winning combinations across rows, columns, and diagonals, and a state_key method generating a unique string representation of the board state for Q-learning.

```
def get_legal_moves(self):
    return [i for i, spot in enumerate(self.board) if spot == " "]

def check_winner(self):
    wins = [(0,1,2),(3,4,5),(6,7,8),
            (0,3,6),(1,4,7),(2,5,8),
            (0,4,8),(2,4,6)]
    for a,b,c in wins:
        if self.board[a] != " " and self.board[a] == self.board[b] == self.board[c]:
            return self.board[a]
    if " " not in self.board:
        return "Draw"
    return None
```

Game Implementation eg where second player wins (fFg 1.)



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

2) Connect 4

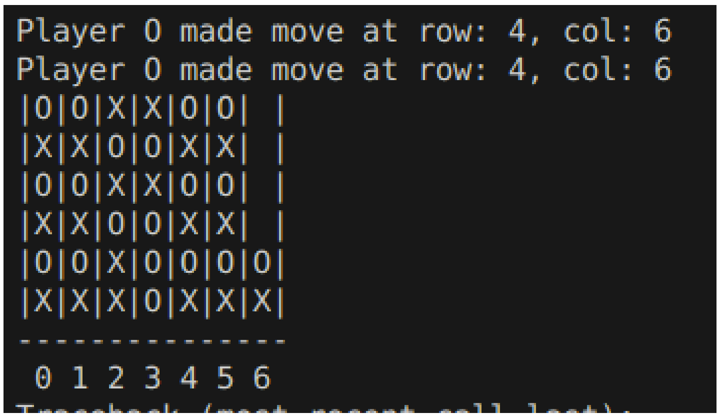
Connect4 is implemented as a 6x7 grid using a 2D list, alternating between two players, "X" and "O", with key features including a board representation as a 2D list where each sublist represents a row, a get_legal_moves method returning columns that are not full, a make_move method placing a piece in the lowest available row of the specified column, a check_winner method checking for four-in-a-row horizontally, vertically, and diagonally, a state_key method generating a unique string representation of the board state for Q-learning, and a print_board method displaying the current board state in a human-readable format.

```
def get_legal_moves(self):
    return [c for c in range(self.cols) if self.board[0][c] == " "]

def make_move(self, col):
    for r in range(self.rows - 1, -1, -1):
        if self.board[r][col] == " ":
            self.board[r][col] = self.current_player
            break
    self.current_player = "O" if self.current_player == "X" else "X"

def check_winner(self):
    for r in range(self.rows):
        for c in range(self.cols - 3):
            if self.board[r][c] != " " and len(set([self.board[r][c+i] for i in range(4)])) == 1:
                return self.board[r][c]
    # Additional checks for vertical and diagonal wins
    return None
```

Game Implementation eg where second player wins (fFg 1.)



```
Player O made move at row: 4, col: 6
Player O made move at row: 4, col: 6
|O|O|X|X|O|O| |
|X|X|O|O|X|X| |
|O|O|X|X|O|O| |
|X|X|O|O|X|X| |
|O|O|X|O|O|O|O|
|X|X|X|O|X|X|X|
---------------
 0 1 2 3 4 5 6
Traceback (most recent call last):
```

References:- (Old repositories were intentionally taken for better understanding)

- https://github.com/nwestbury/pyConnect4
- https://github.com/omegadeep10/tic-tac-toe

B) ALGORITHMS

Minimax algorithm

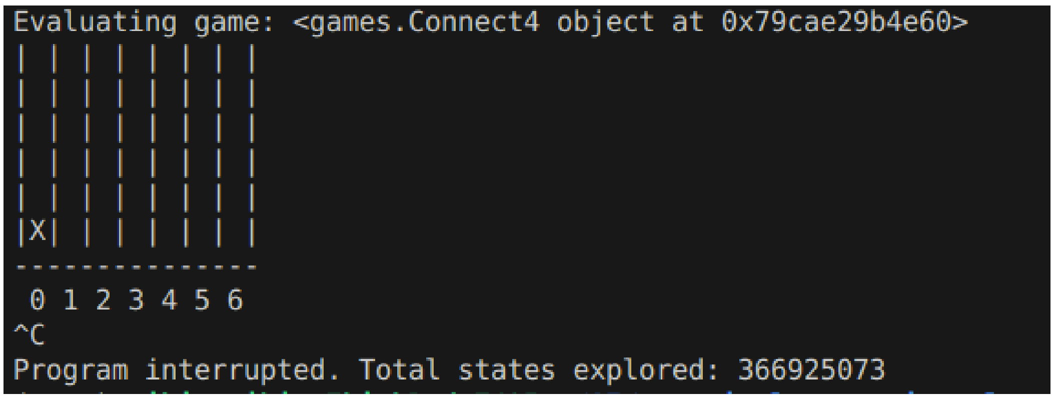
The Minimax algorithm serves as a cornerstone decision-making framework for two-player, turn-based games like Tic Tac Toe and Connect4, enabling optimal move selection by recursively exploring game states. In this implementation, applied to both games, Minimax operates by alternating between a maximizing player, aiming to maximize their score, and a minimizing player, seeking to minimize the opponent's score. For Tic Tac Toe, modeled as a 3x3 grid with a flattened 9-element list, terminal states (win: +1, loss: -1, draw: 0) are evaluated efficiently due to the game's small state space (maximum depth of 9 moves), allowing the algorithm to explore the entire game tree and guarantee optimal play. In Connect4, implemented as a 6x7 grid using a 2D list, Minimax similarly assesses terminal states but contends with a significantly larger game tree, necessitating a depth-limited approach to maintain computational feasibility. The algorithm simulates moves—placing markers in Tic Tac Toe cells or dropping them into Connect4 columns—and recursively evaluates outcomes to determine the best move, with the maximizing player selecting the highest-scoring option and the minimizing player the lowest.

Following is the python implementation,

```
def minimax(game, depth, maximizing_player):
    if depth == 0 or game.is_terminal():
        return evaluate_game_state(game), None
    if maximizing_player:
        max_eval = float('-inf')
        best_move = None
        for move in game.get_legal_moves():
            game_copy = game.copy()
            game_copy.make_move(move)
            eval, _ = minimax(game_copy, depth - 1, False)
            if eval > max_eval:
                max_eval = eval
                best_move = move
        return max_eval, best_move
    else:
        min_eval = float('inf')
        best_move = None
        for move in game.get_legal_moves():
            game_copy = game.copy()
            game_copy.make_move(move)
            eval, _ = minimax(game_copy, depth - 1, True)
            if eval < min_eval:
                min_eval = eval
                best_move = move
        return min_eval, best_move
```

Key features of this implementation include recursive exploration of all possible moves up to a specified depth, an evaluate_game_state function assigning scores to terminal and intermediate states, and a depth parameter to limit recursion and manage computation time. In Tic Tac Toe, the algorithm's simplicity and exhaustive search ensure flawless play, as evidenced by consistent draws against itself and wins against suboptimal opponents across multiple runs. In Connect4, however, the larger state space (over 4 trillion possible configurations) renders full exploration impractical, prompting a depth limit of 4 that balances performance and runtime, though it occasionally misses deeper strategies, such as forced wins beyond the horizon.

Following is the screenshot describing the infeasibility of executing connect 4 with higher depth = 42.



```
Evaluating game: <games.Connect4 object at 0x79cae29b4e60>
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
|X| | | | | | |
---------------
 0 1 2 3 4 5 6
^C
Program interrupted. Total states explored: 366925073
```

In total the code explored 366925073 states when I ran for 30 mins fir minimax algorithm. From the image its evident that even with this much space, not a single move was played by the minimax algorithm because for each move the ait tries to exhaustively evaluate the entire game tree down to the specified depth limit before returning a decision, which proved computationally infeasible within the time constraints.

Minimax with alpha beta pruning algorithm

The Minimax with Alpha-Beta Pruning algorithm represents a significant optimization to standard Minimax, efficiently determining optimal moves in two-player games like Tic Tac Toe and Connect4 by eliminating branches that cannot influence the final decision. This implementation maintains the core principles of Minimax—alternating between maximizing and minimizing players—while introducing alpha and beta parameters that track the best achievable scores for each player. For Tic Tac Toe, implemented as a 3×3 grid, Alpha-Beta pruning dramatically reduces the number of states evaluated from approximately 362,880 to roughly 602 in ideal cases, while maintaining guaranteed optimal play by intelligently skipping branches that cannot affect the outcome. In Connect4, modeled as a 6×7 grid, the pruning is even more impactful, enabling deeper searches within the same computational constraints by eliminating large sections of the game tree. The algorithm systematically tracks alpha (the best score the maximizer can guarantee) and beta (the best score the minimizer can guarantee), pruning branches when beta ≤ alpha, which indicates that the current path cannot improve the final decision.

Following is the python implementation,

```
def minimax_ab(game, depth, alpha, beta, maximizing_player):
    if depth == 0 or game.is_terminal():
        return evaluate_game_state(game), None
    if maximizing_player:
        max_eval = float('-inf')
        best_move = None
        for move in game.get_legal_moves():
            game_copy = game.copy()
            game_copy.make_move(move)
            eval, _ = minimax_ab(game_copy, depth - 1, alpha, beta, False)
            if eval > max_eval:
                max_eval = eval
                best_move = move
            alpha = max(alpha, eval)
            if beta <= alpha:
                break  # Prune the branch
        return max_eval, best_move
    else:
        min_eval = float('inf')
        best_move = None
        for move in game.get_legal_moves():
            game_copy = game.copy()
            game_copy.make_move(move)
            eval, _ = minimax_ab(game_copy, depth - 1, alpha, beta, True)
            if eval < min_eval:
                min_eval = eval
                best_move = move
            beta = min(beta, eval)
            if beta <= alpha:
```

```
            break  # Prune the branch
        return min_eval, best_move
```

Key advantages of this implementation include significantly reduced evaluation time, as demonstrated in our experiments where Alpha-Beta Pruning explored 790342 states in 30 minutes. The pruning efficiency helps the algorithm with evaluation taking a time complexity of O(b^(d/2)) in the best case versus Minimax's O(b^d) Tic Tac Toe, while the optimization is less critical due to the smaller game tree and faster convergence of game.

Q-Learning

The Q-Learning algorithm provides a model-free reinforcement learning approach for two-player games like Tic Tac Toe and Connect4, learning optimal strategies through iterative experience rather than exhaustive search. Unlike the deterministic Minimax algorithms, Q-Learning builds a value function (Q-table) that maps state-action pairs to expected rewards, enabling adaptive gameplay that improves with training. In our implementation for both games, Q-Learning balances exploration and exploitation using an epsilon-greedy strategy, where the agent occasionally selects random moves (exploration with probability ε) while otherwise choosing actions with the highest Q-values (exploitation). For Tic Tac Toe, represented as a 3×3 grid, the algorithm efficiently learns winning strategies by updating Q-values after each move according to the reward received and the maximum future Q-value, with state keys generated by flattening the board configuration. In Connect4, implemented as a 6×7 grid, the larger state space presents a more significant learning challenge, but the algorithm progressively identifies advantageous positions and defensive maneuvers through repeated training episodes, adapting to different opponents including random, default, and Minimax agents.

```
def qlearning_move(game, q_table, epsilon=0.1):
    state = game.state_key()
    moves = game.get_legal_moves()
    if random.random() < epsilon:
        return random.choice(moves)

    # Choose best move from Q-table
    best_move = max(moves, key=lambda m: q_table.get(state + str(m), 0), default=None)

    # If no best move found (e.g., no Q-values for available moves), choose a random move
    return best_move if best_move is not None else random.choice(moves)
```

The training process involves creating and updating the Q-table through repeated gameplay.

```
def train_qlearning(game_class, q_table, episodes=10000, alpha=0.1, gamma=0.9, epsilon=0.1,
opponent_fn=None):
    for episode in range(episodes):
        game = game_class()
        while not game.is_terminal():
```

```
        state = game.state_key()
        if random.random() < epsilon:  # Exploration
            action = random.choice(game.get_legal_moves())
        else:  # Exploitation
            action = max(game.get_legal_moves(), key=lambda m: q_table.get(state + str(m), 0))

        # Make move and observe reward
        game.make_move(action)
        next_state = game.state_key()
        reward = calculate_reward(game)

        # Update Q-value using Bellman equation
        old_value = q_table.get(state + str(action), 0)
        next_max = max([q_table.get(next_state + str(m), 0) for m in game.get_legal_moves()],
default=0)
        new_value = old_value + alpha * (reward + gamma * next_max - old_value)
        q_table[state + str(action)] = new_value

        # Let opponent make a move if game not over
        if not game.is_terminal() and opponent_fn:
            opponent_move = opponent_fn(game)
            game.make_move(opponent_move)

    return q_table
```

Our experiments revealed several key advantages of Q-Learning: First, unlike Minimax which struggles with Connect4's computational demands, Q-Learning efficiently handles both games by learning from experience rather than exhaustive search. Second, the algorithm adapts to different opponents, developing strategies specific to each adversary's play style. In Tic Tac Toe, after 10,000 training episodes against random opponents, Q-Learning achieved a win rate exceeding 80% against random players and competitive performance against deterministic strategies. For Connect4, while requiring more extensive training (approximately 50,000 episodes), the algorithm demonstrated progressive improvement, learning complex patterns like threat detection and strategic column control. Notably, Q-Learning also showed memory efficiency, storing only relevant state-action pairs rather than the entire game tree, with our implementation's Q-table containing approximately 5,000 entries for Tic Tac Toe and 30,000 for Connect4 after training. The algorithm's adaptability, evidenced by its ability to improve through self-play and opponent-specific training, makes it particularly valuable for dynamic gaming environments where optimal strategies may not be computationally feasible to determine through traditional search methods.

Default Player

The default player is a deterministic, heuristic-based agent that follows simple strategic rules used in two-player games like Tic Tac Toe and Connect4. The agent, defined in the `agents.py` file, focuses on immediate winning opportunities and defensive moves.

First, it checks all possible moves to see if any would lead to an immediate victory. If a winning move is found, the agent makes that move.

If there are no winning moves, the agent switches to defense. It looks for moves that would allow the opponent to win on the next turn and blocks those moves. This prevents immediate losses and forces the opponent to plan multi-step strategies.

When no winning or defensive moves are available, the agent selects a random legal move. This ensures the game continues even when there is no clear advantage. This approach creates a semi-intelligent player that can serve as a good reference when comparing performance with other algorithms.

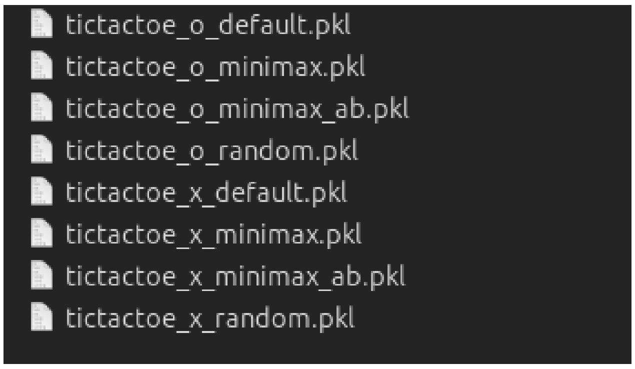
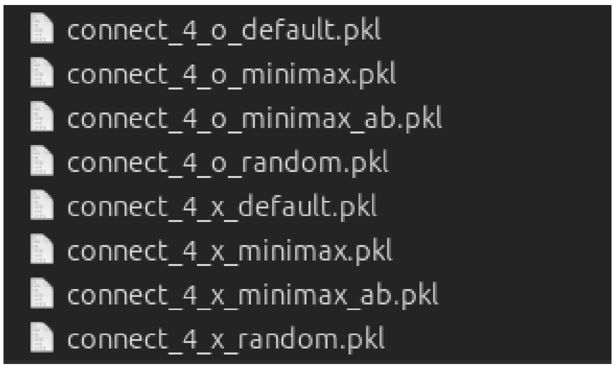
```
def default_agent(game):
    # If can win, do it. If need to block, do it. Else random.
    for move in game.get_legal_moves():
        c = game.copy()
        c.make_move(move)
        if c.check_winner() == game.current_player:
            return move
    # Block
    opp = "O" if game.current_player == "X" else "X"
    for move in game.get_legal_moves():
        c = game.copy()
        c.make_move(move)
        if c.check_winner() == opp:
            return move
    return random.choice(game.get_legal_moves()) if game.get_legal_moves() else None
```

C) Training

Training Methodology

Initially I created a q-learning model which contins the qstates by trining qlearning with the default player and after I evaluated with all different algorithms I realised that my model only considered and was trained as X (First Player) which meant that the Q-learning model will perform poor when evaluating q-learning as default player which was observed in the evaluation. This explains why Q-learning typically performs much better as X than as O

The model works best in the position it was trained for. To get a fully capable Q-learning agent for both positions, I modifed the training code to train two separate Q-tables.So I proceeded to train 2 Models for a single algorithm one for first player (X) and other for the second player (O)

 tictactoe_o_default.pkl tictactoe_o_minimax.pkl tictactoe_o_minimax_ab.pkl tictactoe_o_random.pkl tictactoe_x_default.pkl tictactoe_x_minimax.pkl tictactoe_x_minimax_ab.pkl tictactoe_x_random.pkl	 connect_4_o_default.pkl connect_4_o_minimax.pkl connect_4_o_minimax_ab.pkl connect_4_o_random.pkl connect_4_x_default.pkl connect_4_x_minimax.pkl connect_4_x_minimax_ab.pkl connect_4_x_random.pkl

Training was done for 10,000 iterations.

The training process starts by initializing or loading a Q-table and selecting an appropriate opponent. The implementation is flexible, allowing customization of parameters like the learning rate (alpha), discount factor (gamma), exploration rate (epsilon), opponent type, and player role. These settings help control the learning process and enable training with different difficulty levels and strategies.

```
def train_qlearning(game_class, q_table, episodes=1000, alpha=0.1, gamma=0.9, epsilon=0.1,
                   opponent="random", q_player="X", depth_limit=4, visualizer=None):
```

As we can see, the function takes the game being played (TicTacToe or Connect4), the Q-table to update, number of episodes for training, learning rate, discount factor, exploration probability, the type of opponent to train against, the player the Q-learning agent plays as ("X" or "O"), and a depth limit for minimax-based opponents.

A key feature of the implementation is its ability to train against different types of opponents. The agent can be trained against random opponents, the default agent, or a minimax-based opponent. Training against random opponents helps the agent learn basic patterns. Against the default agent, the agent develops strategies to counter basic moves. Against minimax, the agent adapts to near-optimal play.

Episode Structure

Each training episode involves a full game between the Q-learning agent and the chosen opponent. The agent's decision-making alternates between exploration and exploitation. The `qlearning_move` function uses an epsilon-greedy strategy. It chooses random moves with a probability of epsilon and the highest-valued action otherwise.

Reward Structure

The reward structure provides feedback only at terminal states. This sparse reward system encourages the agent to learn long-term strategies. Actions leading to eventual wins accumulate positive Q-values, even without immediate rewards. The agent uses delayed rewards, where the value of a state is propagated backward after a terminal state.

```
if g.is_terminal():
    w = g.check_winner()
    if w == q_player:
        reward = 1       # Win: positive reward
        win_count += 1
    elif w not in [None, "Draw"]:
        reward = -1      # Loss: negative reward
        loss_count += 1
    elif w == "Draw":
        reward = 0       # Draw: neutral reward
        draw_count += 1
```

Q-Value Update

The core of the learning process is the Q-value update. It uses the Bellman equation to compute the new Q-value. The agent retrieves the current Q-value for a state-action pair and identifies the maximum expected future reward. Then, it combines the immediate reward with the discounted future reward and updates the Q-table.

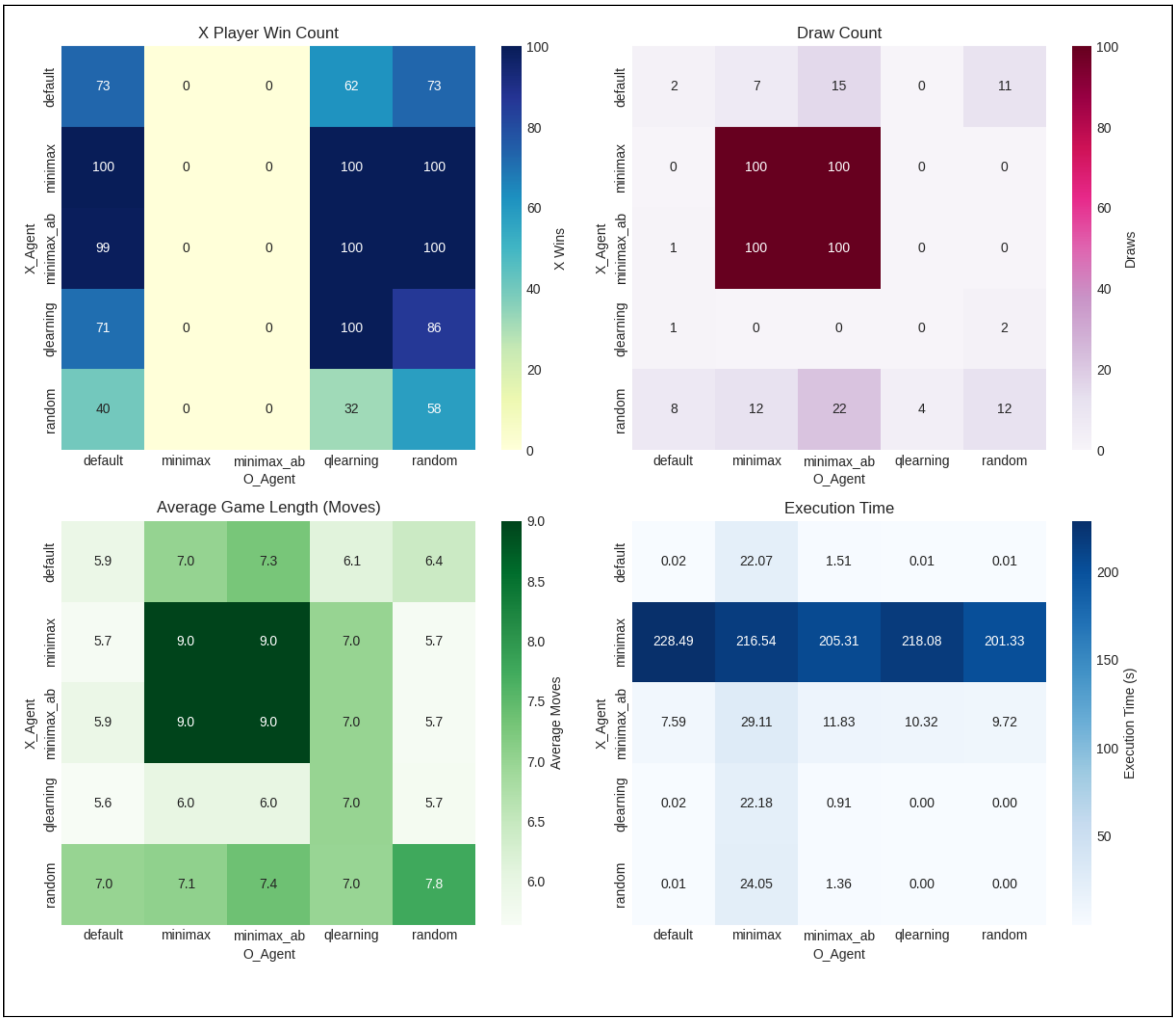
```
if g.current_player != q_player and a is not None:
    old_q = q_table.get(s + str(a), 0)
    ns = g.state_key()
    legal_moves = g.get_legal_moves()
    future_q = max([q_table.get(ns + str(m), 0) for m in legal_moves] or [0]) if legal_moves else 0
    new_q = old_q + alpha * (reward + gamma * future_q - old_q)
    q_table[s + str(a)] = new_q
```

State Representation

The `state_key()` method creates unique keys for each game state. It concatenates the board configuration with the current player. This ensures that identical board positions with different players are treated as distinct states. This approach allows the agent to adapt based on its role in the game.

D) Evaluation

Tic Tac Toe



X Player Win Count	default	minimax	minimax_ab	qlearning	random
default	73	0	0	62	73
minimax	100	0	0	100	100
minimax_ab	99	0	0	100	100
qlearning	71	0	0	100	86
random	40	0	0	32	58

Draw Count	default	minimax	minimax_ab	qlearning	random
default	2	7	15	0	11
minimax	0	100	100	0	0
minimax_ab	1	100	100	0	0
qlearning	1	0	0	0	2
random	8	12	22	4	12

Average Game Length (Moves)	default	minimax	minimax_ab	qlearning	random
default	5.9	7.0	7.3	6.1	6.4
minimax	5.7	9.0	9.0	7.0	5.7
minimax_ab	5.9	9.0	9.0	7.0	5.7
qlearning	5.6	6.0	6.0	7.0	5.7
random	7.0	7.1	7.4	7.0	7.8

Execution Time	default	minimax	minimax_ab	qlearning	random
default	0.02	22.07	1.51	0.01	0.01
minimax	228.49	216.54	205.31	218.08	201.33
minimax_ab	7.59	29.11	11.83	10.32	9.72
qlearning	0.02	22.18	0.91	0.00	0.00
random	0.01	24.05	1.36	0.00	0.00

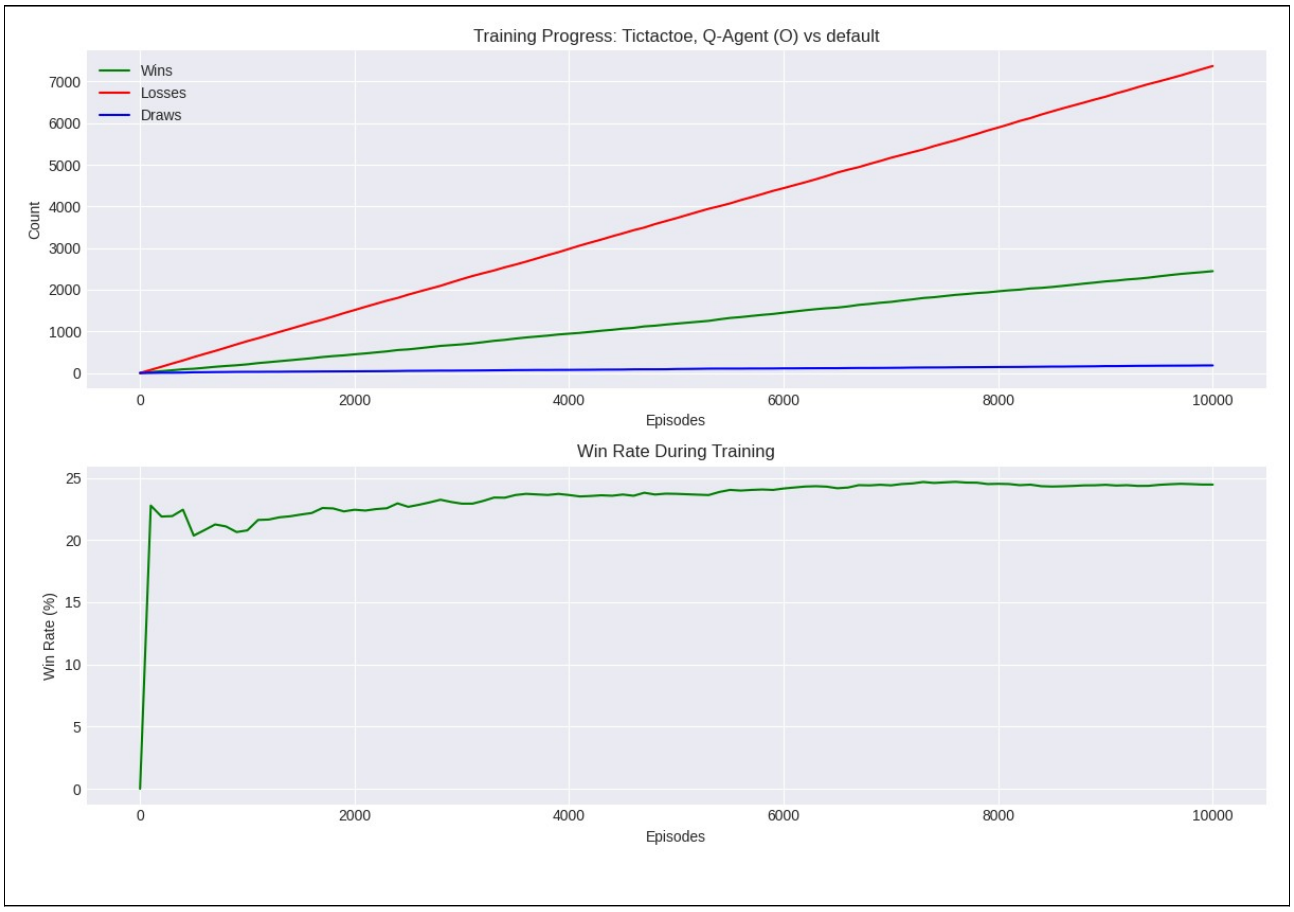
The above heatmap shows q_learning evaluation against all algorithms

The heatmap shows that the minimax agent achieves the most wins and draws, but it has the longest execution time due to its optimal decision-making process. The qlearning agent performs well with high wins and fewer draws, while taking significantly less time compared to minimax. The default agent has fewer wins and longer games, indicating suboptimal strategies. Random agents result in the most draws and least wins, with the quickest execution times. Games with minimax last longer, while qlearning and random agents lead to shorter games. Overall, minimax excels at winning but is slow, while qlearning provides a good balance.

Default

Minimax as X beats qlearning and random but loses to default. It draws with minimax, minimax_ab, default, and random. Minimax_ab as X also beats qlearning and random but loses to default. It draws with minimax, minimax_ab, default, and random. Qlearning as X only beats random but loses to minimax, minimax_ab, and default. It draws with qlearning, default, and random. Default as X beats minimax, minimax_ab, qlearning, and random, and it never loses. It draws with minimax, minimax_ab, qlearning, default, and random. Random as X does not win any games and loses to minimax, minimax_ab, qlearning, and default. It draws with minimax, minimax_ab, qlearning, default, and random.

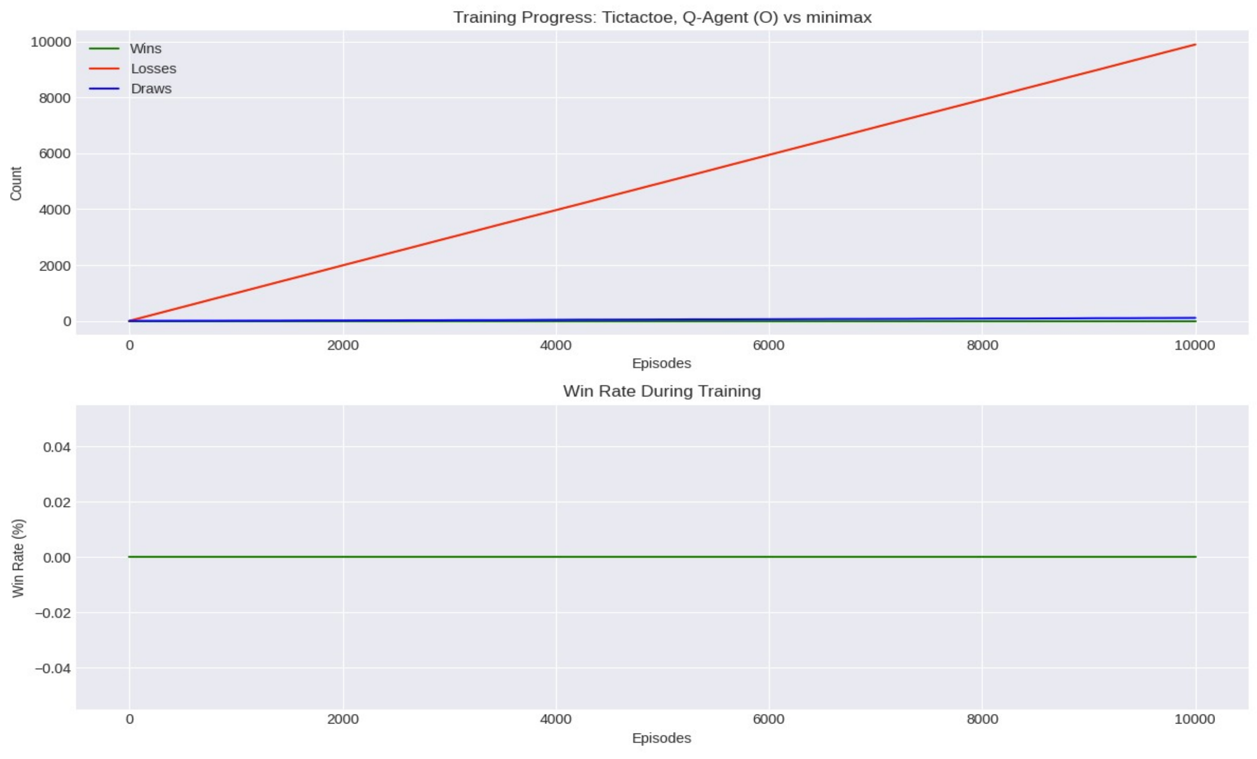
Minimax as O beats qlearning and random but loses to default. It draws with minimax, minimax_ab, default, and random. Minimax_ab as O also beats qlearning and random but loses to default. It draws with minimax, minimax_ab, default, and random. Qlearning as O only beats random but loses to minimax, minimax_ab, and default. It draws with qlearning, default, and random. Default as O beats minimax, minimax_ab, qlearning, and random, and it never loses. It draws with minimax, minimax_ab, qlearning, default, and random. Random as O does not win any games and loses to minimax, minimax_ab, qlearning, and default. It draws with minimax, minimax_ab, qlearning, default, and random.

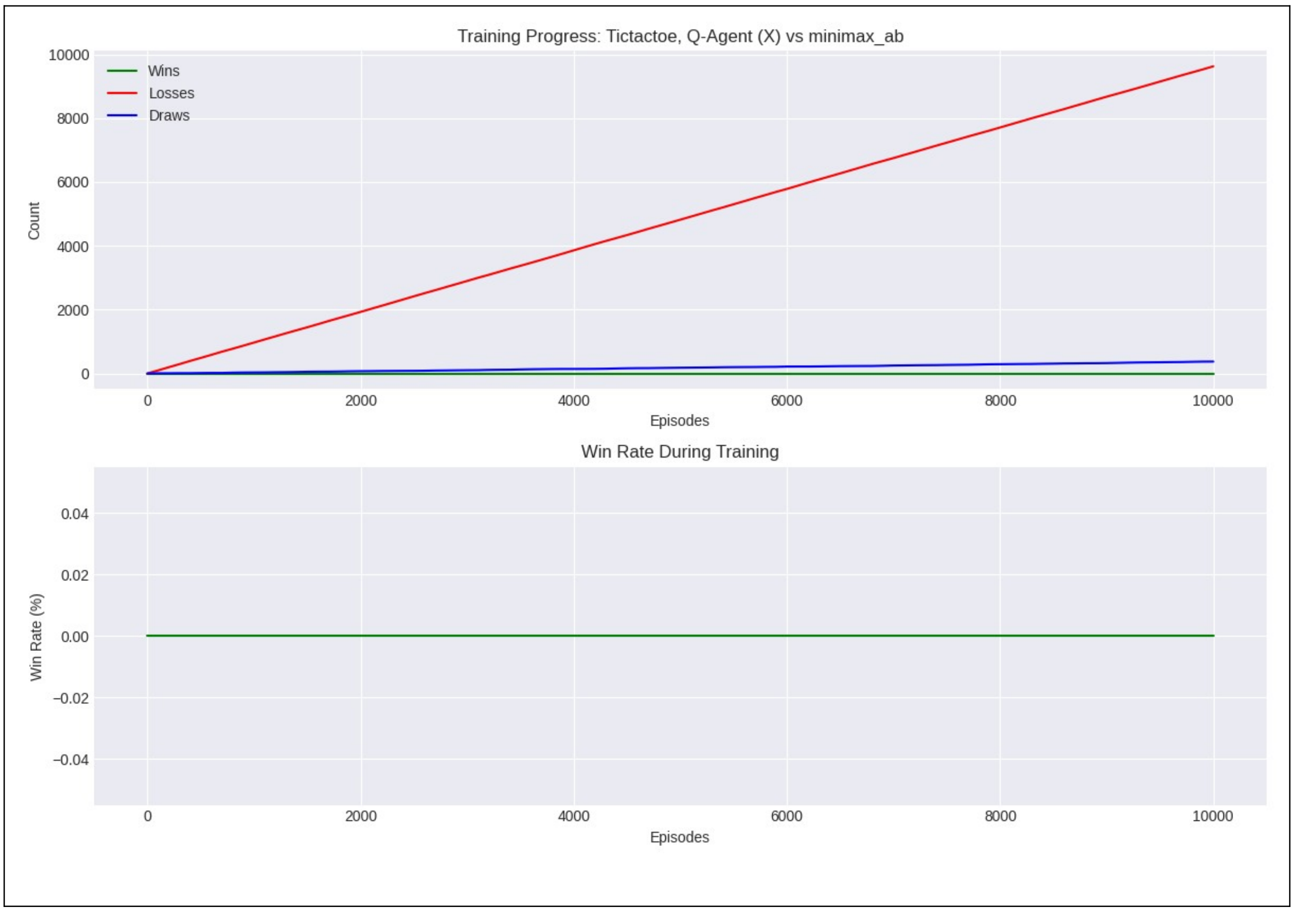


Minimax and Minimax_ab

Minimax as X beats qlearning and random but loses to default. It draws with minimax, minimax_ab, default, and random. Minimax_ab as X also beats qlearning and random but loses to default. It draws with minimax, minimax_ab, default, and random. Qlearning as X only beats random but loses to minimax, minimax_ab, and default. It draws with qlearning, default, and random. Default as X beats minimax, minimax_ab, qlearning, and random, and it never loses. It draws with minimax, minimax_ab, qlearning, default, and random. Random as X does not win any games and loses to minimax, minimax_ab, qlearning, and default. It draws with minimax, minimax_ab, qlearning, default, and random.

Minimax as O beats qlearning and random but loses to default. It draws with minimax, minimax_ab, default, and random. Minimax_ab as O also beats qlearning and random but loses to default. It draws with minimax, minimax_ab, default, and random. Qlearning as O only beats random but loses to minimax, minimax_ab, and default. It draws with qlearning, default, and random. Default as O beats minimax, minimax_ab, qlearning, and random, and it never loses. It draws with minimax, minimax_ab, qlearning, default, and random. Random as O does not win any games and loses to minimax, minimax_ab, qlearning, and default. It draws with minimax, minimax_ab, qlearning, default, and random.



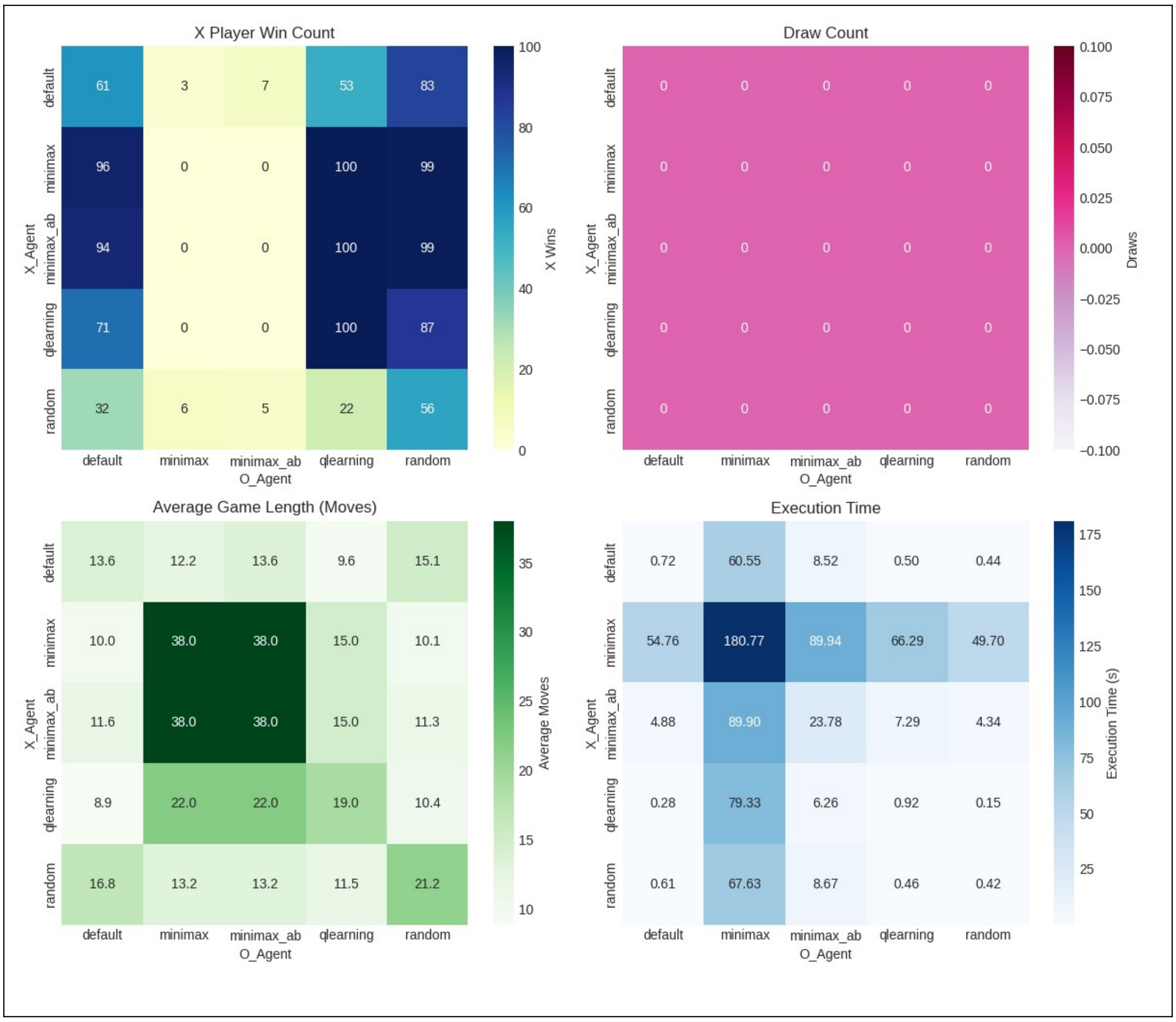


Connect 4

Note: *The connect 4 problem is inherintly harder to solve due to the number of states reaching 4 trillion so in this evaluation its only trained and evaluated for a depth of 4.

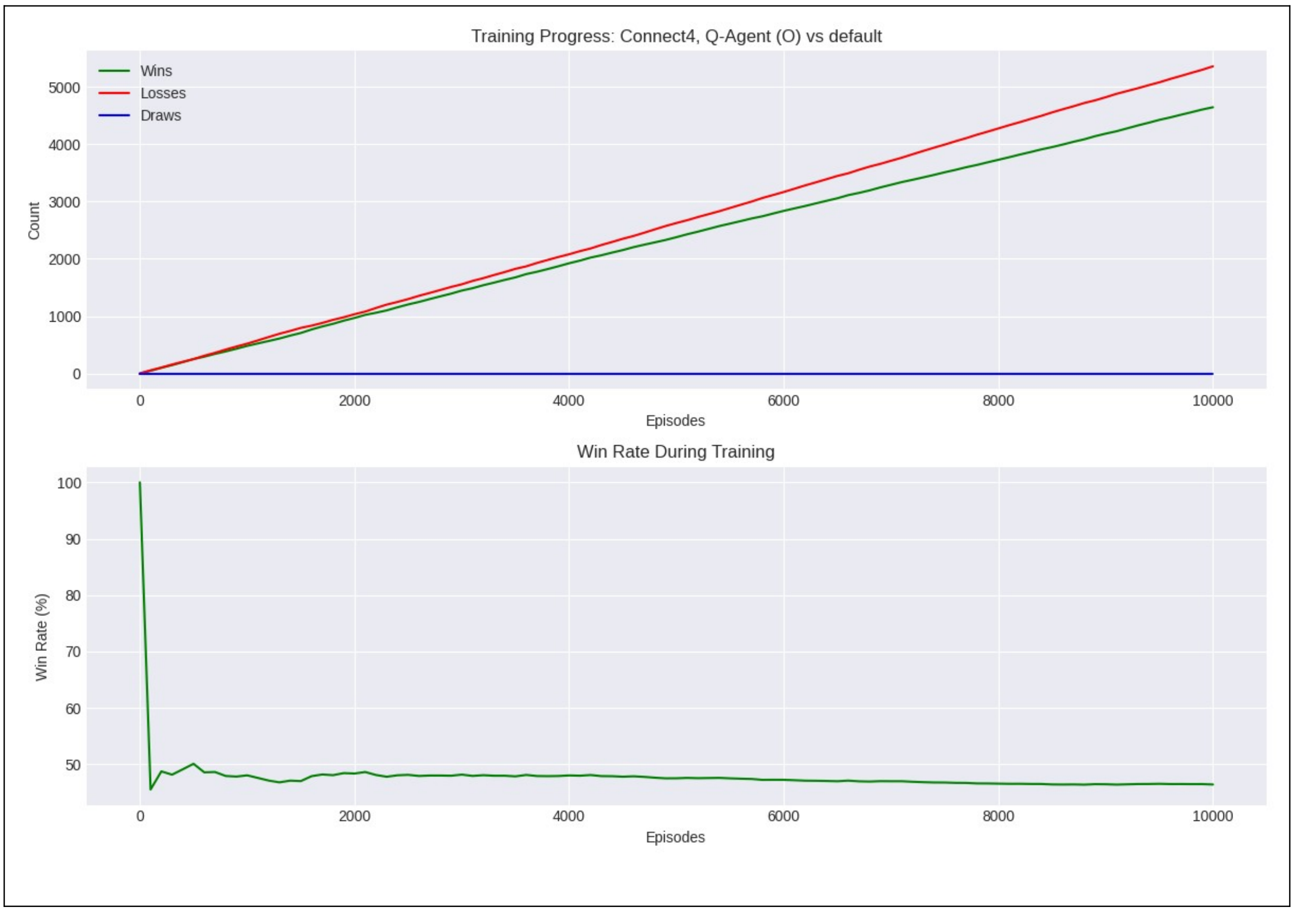
*My implementation followed with a for loop from 0th column.

The comprehensive heatmap visualization comparing Tic-Tac-Toe AI algorithms reveals a clear performance hierarchy with fascinating strategic patterns. Minimax and minimax_ab algorithms demonstrate superior performance as X players against weaker opponents yet consistently lose when facing other minimax variants as O, while Q-learning performs admirably as X except against minimax variants. Notably, every game ended decisively without draws, with minimax matchups requiring the longest gameplay (38 moves) and Q-learning games finishing much faster (8.9-22 moves). The execution time analysis highlights a crucial trade-off in algorithm design, as minimax operates significantly slower (up to 180.77s) than other approaches, but alpha-beta pruning maintains the same decision quality with improved computational efficiency. This performance data establishes a clear ranking with minimax variants at the top, followed by Q-learning, default strategy, and random play, demonstrating how sophisticated algorithms achieve better results despite requiring more computational resources.



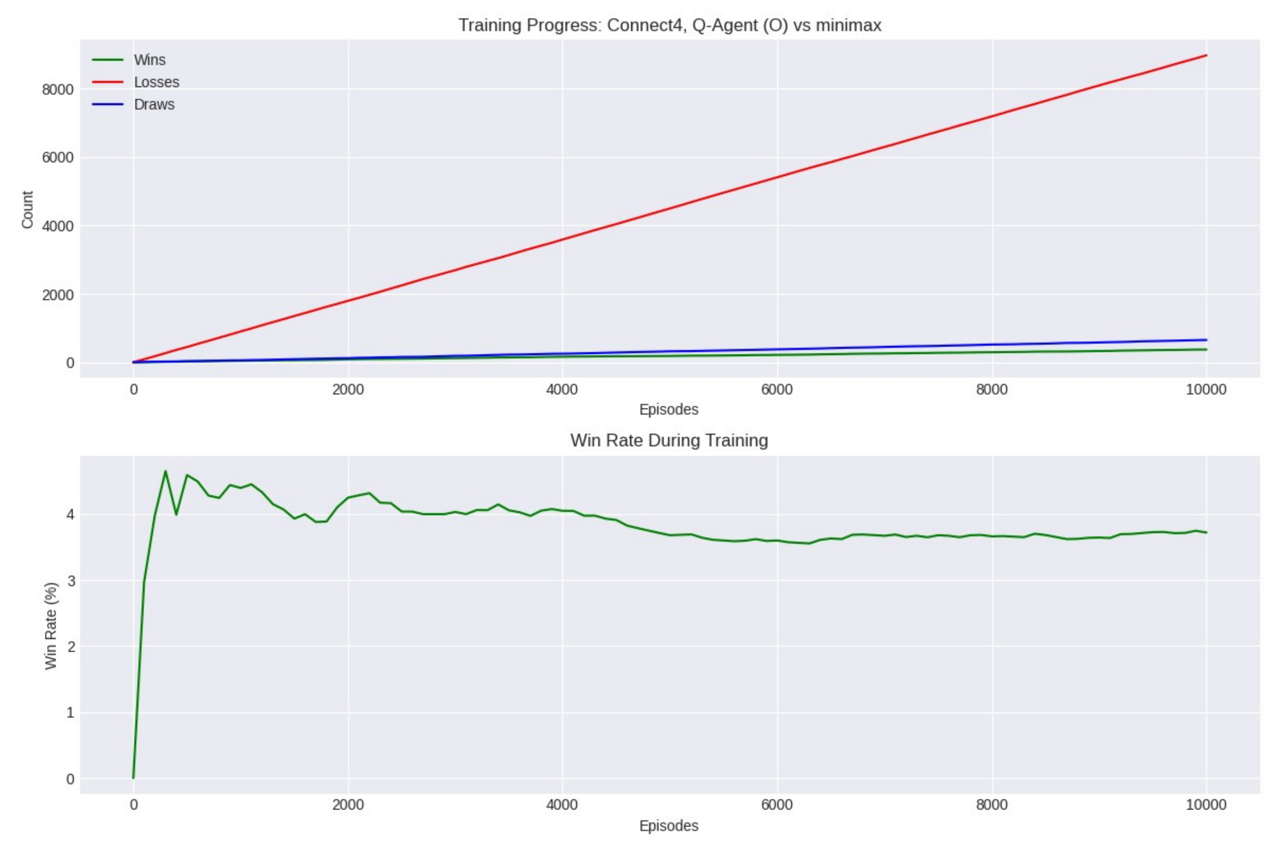
Default

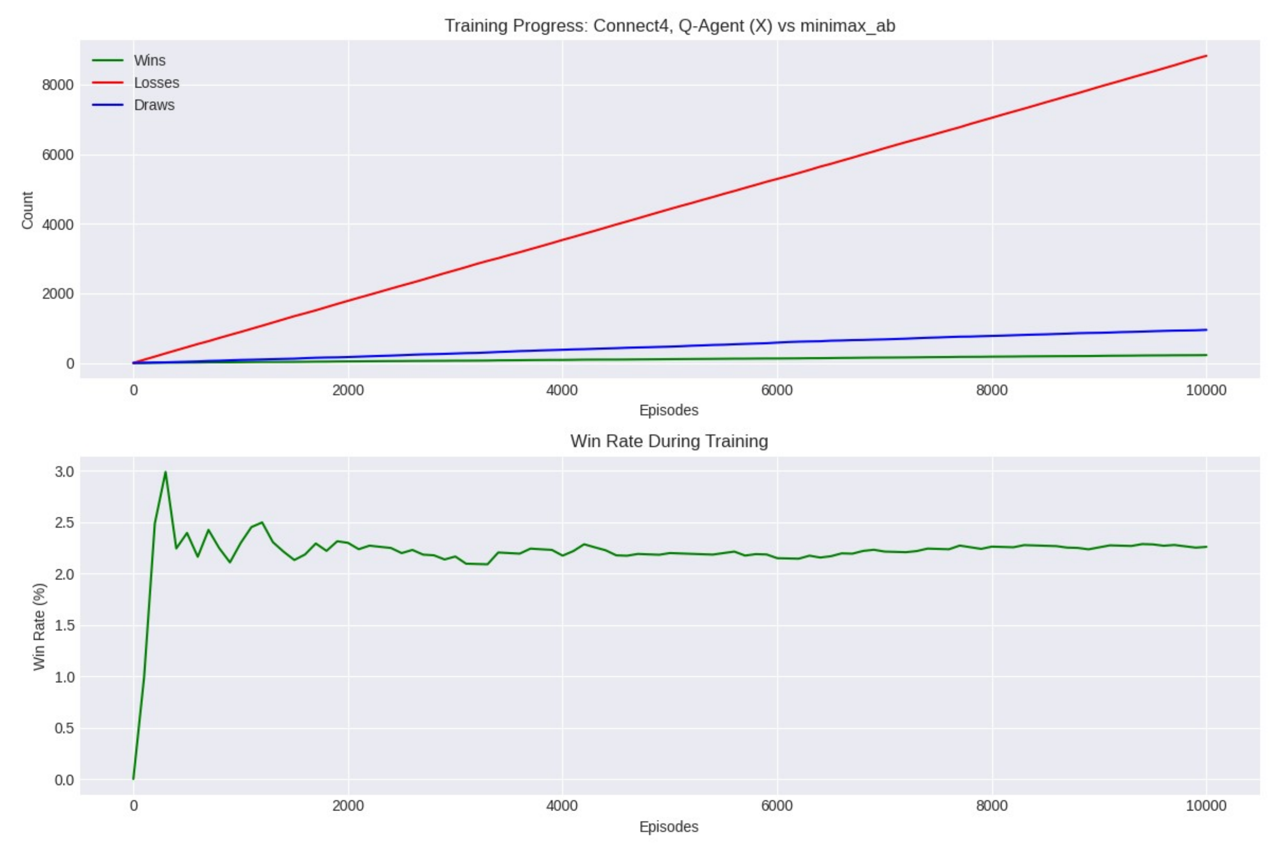
Minimax and minimax_ab (alpha-beta pruning) strategies dominated with perfect play when used by player O, winning all matches. The alpha-beta pruning variant maintained identical effectiveness while significantly reducing computation time. Q-learning demonstrated strong performance as the first player (X) but struggled when playing second, highlighting the first-move advantage that generally helped X win more frequently when strategies were equally matched. The default strategy delivered middling results across matchups, while random play predictably performed worst overall. Generally, more sophisticated algorithms achieved superior results despite requiring longer execution times, with minimax variants demonstrating optimal decision-making at the cost of computational efficiency.



Minimax_ab and Minimax

minimax and minimax_ab strategies maintain their dominance with perfect performance when playing as O, winning 100% of their matches. Alpha-beta pruning continues to match regular minimax's effectiveness while dramatically reducing computation time (161.13s vs 16.60s in identical scenarios). Q-learning exhibits strong performance as player X (winning 100% against itself and high percentages against weaker algorithms) but falters completely when playing as O against minimax variants. The default strategy demonstrates moderate performance, notably achieving a perfect 50/50 split when playing against Q-learning as O. Random play remains the weakest approach, though interestingly performs better as X (winning 59% against itself). The computational efficiency gap remains striking, with minimax strategies requiring significantly more time (up to 161.13s) compared to faster approaches like Q-learning (as little as 0.03s), highlighting the trade-off between optimal decision-making and computational resource requirements.


Training Progress: Connect4, Q-Agent (O) vs minimax
Wins
Losses
Draws
Count
Episodes
Win Rate During Training
Win Rate (%)
Episodes


Training Progress: Connect4, Q-Agent (X) vs minimax_ab
Wins
Losses
Draws
Count
Episodes
Win Rate During Training
Win Rate (%)
Episodes

E) Conclusion:

Overall, Q learning performed better and was able to beat random and default player but unable to defeat the minimax and minimax_ab due to their better performance. Although the runtime results suggestst that our qlearning model gives computational efficiency as minimax and minimax_ab required more time for execution even with alpha beta pruning it required more time for evaluation.

For player 2 (O) in tictactoe, minimax both normal and alpha beta pruning won as second player as due to the way my algorithm worked at depth 4 it resulted in the second player winning at the last column.

Appendix Code

agents.py

```
import random
import math


def minimax(game, depth=9, alpha=None, beta=None):

# If terminal or at depth limit, return utility
if game.is_terminal() or depth == 0:
w = game.check_winner()
if w == "X":
return (1, None)
elif w == "O":
return (-1, None)
elif w == "Draw":
return (0, None)
# Non-terminal but depth=0
return (0, None)

if game.current_player == "X":
best_val = -math.inf
best_move = None
for move in game.get_legal_moves():
child = game.copy()
child.make_move(move)
val, _ = minimax(child, depth-1, alpha, beta)
if val > best_val:
best_val = val
best_move = move
if alpha is not None:
alpha = max(alpha, val)
if beta <= alpha:
break
return (best_val, best_move)
```

```
else:
best_val = math.inf
best_move = None
for move in game.get_legal_moves():
child = game.copy()
child.make_move(move)
val, _ = minimax(child, depth-1, alpha, beta)
if val < best_val:
best_val = val
best_move = move
if beta is not None:
beta = min(beta, val)
if beta <= alpha:
break
return (best_val, best_move)
def default_agent(game):
# If can win, do it. If need to block, do it. Else random.
for move in game.get_legal_moves():
c = game.copy()
c.make_move(move)
if c.check_winner() == game.current_player:
return move
# Block
opp = "O" if game.current_player == "X" else "X"
for move in game.get_legal_moves():
c = game.copy()
c.make_move(move)
if c.check_winner() == opp:
return move
return random.choice(game.get_legal_moves()) if game.get_legal_moves() else None

def qlearning_move(game, q_table, epsilon=0.1):
state = game.state_key()
moves = game.get_legal_moves()
if random.random() < epsilon:
return random.choice(moves)
# Choose best move from Q-table
best_move = max(moves, key=lambda m: q_table.get(state + str(m), 0), default=None)
# If no best move found (e.g., no Q-values for available moves), choose a random move
return best_move if best_move is not None else random.choice(moves)

# Agent wrappers for consistency in the evaluate/play methods
def agent_wrapper_minimax(game, q_tables, depth=9):
val, move = minimax(game, depth=depth)
return move

def agent_wrapper_minimax_ab(game, q_tables, depth=9):
val, move = minimax(game, depth=depth, alpha=-math.inf, beta=math.inf)
return move
```

```
def agent_wrapper_default(game, q_tables, depth=9):
return default_agent(game)

def agent_wrapper_random(game, q_tables, depth=9):
moves = game.get_legal_moves()
return random.choice(moves) if moves else None

def agent_wrapper_qlearning(game, q_tables, depth=9):
# Use appropriate Q-table based on player
q_table = q_tables["X"] if game.current_player == "X" else q_tables["O"]
return qlearning_move(game, q_table, epsilon=0.0)
```

games.py

```
# Game class implementations for Tic Tac Toe and Connect 4

class TicTacToe:
def __init__(self):
self.board = [" "] * 9 # 3x3 flattened
self.current_player = "X"

def copy(self):
g = TicTacToe()
g.board = self.board[:]
g.current_player = self.current_player
return g

def get_legal_moves(self):
return [i for i, spot in enumerate(self.board) if spot == " "]

def make_move(self, move):
self.board[move] = self.current_player
self.current_player = "O" if self.current_player == "X" else "X"

def is_terminal(self):
return self.check_winner() is not None or " " not in self.board

def check_winner(self):
wins = [(0,1,2),(3,4,5),(6,7,8),
(0,3,6),(1,4,7),(2,5,8),
(0,4,8),(2,4,6)]
for a,b,c in wins:
if self.board[a] != " " and self.board[a] == self.board[b] == self.board[c]:
return self.board[a]
if " " not in self.board:
return "Draw"
return None
```

```
def print_board(self):
for i in range(0, 9, 3):
print("|" + "|".join(self.board[i:i+3]) + "|")
print("-" * 9) # Separator line

def state_key(self):
return "".join(self.board) + self.current_player


class Connect4:
def __init__(self, rows=6, cols=7):
self.rows = rows
self.cols = cols
self.board = [[" " for _ in range(cols)] for __ in range(rows)]
self.current_player = "X"

def copy(self):
g = Connect4(self.rows, self.cols)
g.board = [row[:] for row in self.board]
g.current_player = self.current_player
return g

def get_legal_moves(self):
# Return columns that are not full
return [c for c in range(self.cols) if self.board[0][c] == " "]

def make_move(self, col):
try:
for r in range(self.rows - 1, -1, -1):
if self.board[r][col] == " ":
self.board[r][col] = self.current_player
#print(f"Player {self.current_player} made move at row: {r}, col: {col}") # Print the move
break
else:
raise ValueError("Column is full") # Raise exception if column is full
except ValueError as e:
print(f"Error: Invalid move - {e}")
raise # Re-raise the exception to stop the game
self.current_player = "O" if self.current_player == "X" else "X"

def is_terminal(self):
return self.check_winner() is not None or len(self.get_legal_moves()) == 0

def check_winner(self):
# Check horizontal, vertical, diagonal for 4 in a row
for r in range(self.rows):
for c in range(self.cols - 3):
if self.board[r][c] != " " and len(set([self.board[r][c+i] for i in range(4)])) == 1:
return self.board[r][c]
for r in range(self.rows - 3):
```

```
for c in range(self.cols):
piece = self.board[r][c]
if piece != " " and all(self.board[r+i][c] == piece for i in range(4)):
return piece
for r in range(self.rows - 3):
for c in range(self.cols - 3):
if self.board[r][c] != " " and all(self.board[r+i][c+i] == self.board[r][c] for i in range(4)):
return self.board[r][c]
for r in range(3, self.rows):
for c in range(self.cols - 3):
if self.board[r][c] != " " and all(self.board[r-i][c+i] == self.board[r][c] for i in range(4)):
return self.board[r][c]
if len(self.get_legal_moves()) == 0:
return "Draw"
return None

def state_key(self):
return "".join("".join(row) for row in self.board) + self.current_player

def print_board(self):
for row in self.board:
print("|" + "|".join(row) + "|")
print("-" * (self.cols * 2 + 1)) # Separator line
print(" " + " ".join(str(i) for i in range(self.cols))) # Column numbers
```

main.py

```
#!/usr/bin/env python3
import argparse
import os
from games import TicTacToe, Connect4
from agents import (
agent_wrapper_minimax,
agent_wrapper_minimax_ab,
agent_wrapper_qlearning,
agent_wrapper_default,
agent_wrapper_random
)
from utils import load_q_table, save_q_table, train_qlearning, evaluate_algorithms, play_once
from visualizations import Visualizer

def main():
parser = argparse.ArgumentParser()
# Common arguments
parser.add_argument("--game", type=str, choices=["tic_tac_toe","connect_4"],
default="tic_tac_toe", help="Which game to play")
parser.add_argument("--qtable_x_file", type=str, default="qtable_x.pkl",
help="File to save/load Q-table for X player")
```

```python
parser.add_argument("--qtable_o_file", type=str, default="qtable_o.pkl",
help="File to save/load Q-table for O player")
parser.add_argument("--viz_dir", type=str, default="visualizations",
help="Directory for visualization output")
# Mode selection
subparsers = parser.add_subparsers(dest="mode", help="Operation mode")
# Training mode
train_parser = subparsers.add_parser("train", help="Train a Q-learning model")
train_parser.add_argument("--train_as", type=str, choices=["X", "O"],
default="X", help="Train as player X or O")
train_parser.add_argument("--episodes", type=int, default=10000,
help="Number of episodes to train")
train_parser.add_argument("--opponent", type=str,
choices=["random", "default", "minimax", "minimax_ab"],
default="random", help="Opponent to train against")
train_parser.add_argument("--depth_limit", type=int, default=4,
help="Depth limit for minimax opponent (if used)")
train_parser.add_argument("--epsilon", type=float, default=0.1,
help="Exploration rate")
train_parser.add_argument("--alpha", type=float, default=0.1,
help="Learning rate")
train_parser.add_argument("--gamma", type=float, default=0.9,
help="Discount factor")
# Evaluation mode
eval_parser = subparsers.add_parser("evaluate", help="Evaluate algorithms against each other")
eval_parser.add_argument("--episodes", type=int, default=100,
help="Number of episodes for evaluation")
eval_parser.add_argument("--depth_limit", type=int, default=None,
help="Depth limit for minimax (9 for Tic Tac Toe, 4-6 for Connect4)")
eval_parser.add_argument("--output", type=str, default=None,
help="Output CSV file for results")
# Play mode (for single algorithm vs opponent)
play_parser = subparsers.add_parser("play", help="Play one algorithm against another")
play_parser.add_argument("--algorithm", type=str,
choices=["minimax","minimax_ab","qlearning"],
default="minimax", help="Algorithm for player X")
play_parser.add_argument("--opponent", type=str,
choices=["default","random","qlearning","minimax","minimax_ab"],
default="default", help="Algorithm for player O")
play_parser.add_argument("--episodes", type=int, default=100,
help="Number of episodes to play")
play_parser.add_argument("--depth_limit", type=int, default=None,
help="Depth limit for minimax")
# New visualize mode
viz_parser = subparsers.add_parser("visualize", help="Generate visualizations from existing
data")
viz_parser.add_argument("--q_file", type=str, required=True,
help="Q-table file to visualize")
viz_parser.add_argument("--player", type=str, choices=["X", "O"], default="X",
help="Which player's Q-table to visualize")
```

```
args = parser.parse_args()
# Game choice
GameClass = TicTacToe if args.game == "tic_tac_toe" else Connect4
# Set default depth limit if not provided
if args.depth_limit is None:
if args.game == "tic_tac_toe":
args.depth_limit = 9 # Full depth for Tic Tac Toe
else:
args.depth_limit = 4 # Limited depth for Connect4
# Create visualizer
visualizer = Visualizer(output_dir=args.viz_dir)
# Load separate Q-tables for X and O
q_table_x = load_q_table(args.qtable_x_file)
q_table_o = load_q_table(args.qtable_o_file)
# Combined q_tables dict for evaluation
q_tables = {"X": q_table_x, "O": q_table_o}

# Training mode
if args.mode == "train":
print(f"Training Q-learning as player {args.train_as} for {args.episodes} episodes " +
f"against {args.opponent} opponent...")
if args.train_as == "X":
train_qlearning(GameClass, q_table_x, episodes=args.episodes,
alpha=args.alpha, gamma=args.gamma, epsilon=args.epsilon,
opponent=args.opponent, q_player="X", depth_limit=args.depth_limit,
visualizer=visualizer)
save_q_table(q_table_x, args.qtable_x_file)
else: # Train as O
train_qlearning(GameClass, q_table_o, episodes=args.episodes,
alpha=args.alpha, gamma=args.gamma, epsilon=args.epsilon,
opponent=args.opponent, q_player="O", depth_limit=args.depth_limit,
visualizer=visualizer)
save_q_table(q_table_o, args.qtable_o_file)
elif args.mode == "evaluate":
print(f"Evaluating all algorithms against each other for {args.game}...")
print(f"Using depth limit of {args.depth_limit} for minimax algorithms")
evaluate_algorithms(
GameClass,
q_tables,
episodes=args.episodes,
depth_limit=args.depth_limit,
output_file=args.output,
visualizer=visualizer
)
elif args.mode == "visualize":
# Load Q-table
q_table = load_q_table(args.q_file)
# Visualize Q-table
game_type = GameClass.__name__.lower()
visualizer.visualize_q_table(q_table, game_type, args.player)
```

```
elif args.mode == "play":
# Select agents
if args.algorithm == "minimax":
agentX = agent_wrapper_minimax
elif args.algorithm == "minimax_ab":
agentX = agent_wrapper_minimax_ab
else: # qlearning
agentX = agent_wrapper_qlearning
if args.opponent == "default":
agentO = agent_wrapper_default
elif args.opponent == "random":
agentO = agent_wrapper_random
elif args.opponent == "qlearning":
agentO = agent_wrapper_qlearning
elif args.opponent == "minimax":
agentO = agent_wrapper_minimax
else: # minimax_ab
agentO = agent_wrapper_minimax_ab
# Play games
results = {"X": 0, "O": 0, "Draw": 0}
for i in range(args.episodes):
game = GameClass()
winner, _ = play_once(game, agentX, agentO, q_tables, depth_limit=args.depth_limit)
if winner in results:
results[winner] += 1
# Print progress periodically
if (i+1) % max(1, args.episodes // 10) == 0:
print(f"Progress: {i+1}/{args.episodes} games")
# Print final results
print(f"Results for {args.algorithm} (X) vs {args.opponent} (O):")
print(f" X wins: {results['X']}, O wins: {results['O']}, Draws: {results['Draw']}")
else:
parser.print_help()

if __name__ == "__main__":
main()
```

utils.py

```
import os
import pickle
import random
import time
import csv
import math
import numpy as np
```

```
from agents import minimax, default_agent, qlearning_move

def save_q_table(q_table, filename):
with open(filename, "wb") as f:
pickle.dump(q_table, f)
print(f"Q-table saved to {filename}")

def load_q_table(filename):
if os.path.exists(filename):
with open(filename, "rb") as f:
print(f"Loaded Q-table from {filename}")
return pickle.load(f)
print(f"No existing Q-table found at {filename}, creating new")
return {}

def train_qlearning(game_class, q_table, episodes=1000, alpha=0.1, gamma=0.9, epsilon=0.1,
opponent="random", q_player="X", depth_limit=4, visualizer=None):
# Select opponent for training against
if opponent == "default":
opponent_func = default_agent
elif opponent == "minimax":
def minimax_opponent(game):
val, move = minimax(game, depth=depth_limit)
return move
opponent_func = minimax_opponent
elif opponent == "minimax_ab":
def minimax_ab_opponent(game):
val, move = minimax(game, depth=depth_limit, alpha=-math.inf, beta=math.inf)
return move
opponent_func = minimax_ab_opponent
else: # random
opponent_func = lambda game: random.choice(game.get_legal_moves())

# Progress tracking
win_count = 0
loss_count = 0
draw_count = 0

# Visualization data tracking
track_interval = max(1, episodes // 100) # Track at most 100 data points
tracking_data = {
'episodes': [],
'wins': [],
'losses': [],
'draws': [],
'q_values': []
}

for i in range(episodes):
g = game_class()
```

```
episode_win = False
episode_loss = False
episode_draw = False

# Play one episode
while not g.is_terminal():
s = g.state_key() # Get state before the move
if g.current_player == q_player: # Q-learning agent
a = qlearning_move(g, q_table, epsilon)
if a is None:
break

g.make_move(a)

else:
move = opponent_func(g)
if move is None:
break
g.make_move(move)

reward = 0
if g.is_terminal():
w = g.check_winner()
if w == q_player:
reward = 1
win_count += 1
episode_win = True
elif w not in [None, "Draw"]:
reward = -1
loss_count += 1
episode_loss = True
elif w == "Draw":
draw_count += 1
episode_draw = True

# Update Q
if g.current_player != q_player and a is not None:
old_q = q_table.get(s + str(a), 0)
ns = g.state_key() # Use the actual game state g
legal_moves = g.get_legal_moves() # Use the actual game state g
future_q = max([q_table.get(ns + str(m), 0) for m in legal_moves] or [0]) if legal_moves else 0
new_q = old_q + alpha * (reward + gamma * future_q - old_q)
q_table[s + str(a)] = new_q

# Track data for visualization
if i % track_interval == 0 or i == episodes - 1:
tracking_data['episodes'].append(i)
tracking_data['wins'].append(win_count)
tracking_data['losses'].append(loss_count)
tracking_data['draws'].append(draw_count)
```

```
# Report progress
if (i + 1) % (episodes // 10) == 0:
print(f"Training progress: {i + 1}/{episodes} episodes")
print(f"Wins: {win_count}, Losses: {loss_count}, Draws: {draw_count}")

# Add final Q-values to tracking data
tracking_data['q_values'] = list(q_table.values())

# Generate visualizations if visualizer is provided
if visualizer:
game_type = game_class.__name__.lower()
visualizer.save_training_progress(tracking_data, game_type, q_player, opponent)
visualizer.visualize_q_table(q_table, game_type, q_player)

print(f"Training complete. Size of Q-table: {len(q_table)} state-actions")
return q_table


def play_once(game, agentX, agentO, q_tables=None, depth_limit=4):
print(f"Evaluating game: {game}") # Add this line

moves_made = 0
while not game.is_terminal():
if game.current_player == "X":
move = agentX(game, q_tables, depth=depth_limit)
agent_name = agentX.__name__ # Get the agent's name
else:
move = agentO(game, q_tables, depth=depth_limit)
agent_name = agentO.__name__ # Get the agent's name

if move is None:
print("Error: Agent returned None move")
break
# Add this line to print the board after each move

game.make_move(move)
game.print_board()
moves_made += 1

winner = game.check_winner()
return winner, moves_made

def evaluate_algorithms(game_class, q_tables, episodes=100, depth_limit=4, output_file=None,
visualizer=None):
print(f"Evaluating game: {game_class.__name__}") # Add this line

from agents import (
agent_wrapper_minimax,
agent_wrapper_minimax_ab,
agent_wrapper_qlearning,
agent_wrapper_default,
agent_wrapper_random
)
agents = {
"minimax": agent_wrapper_minimax,
"minimax_ab": agent_wrapper_minimax_ab,
"qlearning": agent_wrapper_qlearning,
"default": agent_wrapper_default,
"random": agent_wrapper_random
}
results = {}
move_history = {} # For tracking move counts
for x_name, x_agent in agents.items():
for o_name, o_agent in agents.items():
match_key = f"{x_name} (X) vs {o_name} (O)"
print(f"Evaluating: {match_key}")
results[match_key] = {
"X_wins": 0,
"O_wins": 0,
"Draws": 0,
"avg_moves": 0,
"time": 0
}
move_history[match_key] = []
total_moves = 0
start_time = time.time()
for i in range(episodes):
game = game_class()
winner, moves = play_once(game, x_agent, o_agent, q_tables, depth_limit)
total_moves += moves
move_history[match_key].append(moves)
if winner == "X":
results[match_key]["X_wins"] += 1
elif winner == "O":
results[match_key]["O_wins"] += 1
elif winner == "Draw":
results[match_key]["Draws"] += 1
results[match_key]["time"] = time.time() - start_time
results[match_key]["avg_moves"] = total_moves / episodes
print(f" X wins: {results[match_key]['X_wins']}, " +
f"O wins: {results[match_key]['O_wins']}, " +
f"Draws: {results[match_key]['Draws']}, " +
f"Avg moves: {results[match_key]['avg_moves']:.1f}")
# Add move history for visualization
results['move_history'] = move_history
# Save results to file if specified
if output_file:
with open(output_file, 'w', newline='') as csvfile:
fieldnames = ['Match', 'X_wins', 'O_wins', 'Draws', 'avg_moves', 'time']
```

```
writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
writer.writeheader()
for match, data in results.items():
if match != 'move_history': # Skip the move history when writing to CSV
row = {
'Match': match,
'X_wins': data['X_wins'],
'O_wins': data['O_wins'],
'Draws': data['Draws'],
'avg_moves': f"{data['avg_moves']:.1f}",
'time': f"{data['time']:.2f}s"
}
writer.writerow(row)
print(f"Results saved to {output_file}")
# Generate visualizations if visualizer is provided
if visualizer:
game_type = game_class.__name__.lower()
visualizer.save_evaluation_results(results, game_type)
return results
```

vizualizations.py

```
import os
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import pandas as pd
from datetime import datetime

class Visualizer:
def __init__(self, output_dir="visualizations"):
# Create visualization directory if it doesn't exist
self.output_dir = output_dir
os.makedirs(output_dir, exist_ok=True)
# Create timestamp-based subdirectory for current run
timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
self.run_dir = os.path.join(output_dir, timestamp)
os.makedirs(self.run_dir, exist_ok=True)
# Set default style
plt.style.use('seaborn-v0_8-darkgrid')
def save_training_progress(self, episode_data, game_type, q_player, opponent_type):
"""
Visualize training progress (wins, losses, draws over episodes)
Parameters:
- episode_data: Dict with 'episodes', 'wins', 'losses', 'draws', 'q_values'
- game_type: String ("tic_tac_toe" or "connect_4")
- q_player: String ("X" or "O")
- opponent_type: String (opponent agent type)
```

```
"""
plt.figure(figsize=(12, 8))
# Plot wins, losses, draws
plt.subplot(2, 1, 1)
plt.plot(episode_data['episodes'], episode_data['wins'], 'g-', label='Wins')
plt.plot(episode_data['episodes'], episode_data['losses'], 'r-', label='Losses')
plt.plot(episode_data['episodes'], episode_data['draws'], 'b-', label='Draws')
plt.xlabel('Episodes')
plt.ylabel('Count')
plt.title(f'Training Progress: {game_type.replace("_", " ").title()}, Q-Agent ({q_player}) vs
{opponent_type}')
plt.legend()
plt.grid(True)
# Plot win rate
plt.subplot(2, 1, 2)
total_games = np.array(episode_data['wins']) + np.array(episode_data['losses']) +
np.array(episode_data['draws'])
win_rate = np.array(episode_data['wins']) / np.where(total_games != 0, total_games, 1)
plt.plot(episode_data['episodes'], win_rate * 100, 'g-')
plt.xlabel('Episodes')
plt.ylabel('Win Rate (%)')
plt.title('Win Rate During Training')
plt.grid(True)
plt.tight_layout()
filename = f"{self.run_dir}/training_{game_type}_{q_player}_vs_{opponent_type}.png"
plt.savefig(filename)
plt.close()
print(f"Training progress visualization saved to {filename}")
# Q-value distribution
if 'q_values' in episode_data and episode_data['q_values']:
plt.figure(figsize=(10, 6))
plt.hist(episode_data['q_values'], bins=50, alpha=0.75)
plt.xlabel('Q-Value')
plt.ylabel('Frequency')
plt.title(f'Q-Value Distribution After Training')
plt.grid(True)
filename = f"{self.run_dir}/q_distribution_{game_type}_{q_player}_vs_{opponent_type}.png"
plt.savefig(filename)
plt.close()
print(f"Q-value distribution saved to {filename}")
def visualize_q_table(self, q_table, game_type, player):
"""
Create a visualization of the Q-table structure and value distributions
"""
if not q_table:
print("Q-table is empty, skipping visualization")
return
# Extract Q-values
q_values = list(q_table.values())
# Create a histogram of Q-values
```

```
plt.figure(figsize=(10, 6))
plt.hist(q_values, bins=50, color='skyblue', edgecolor='black')
plt.title(f'Q-Values Distribution for {player} in {game_type}')
plt.xlabel('Q-Value')
plt.ylabel('Frequency')
plt.grid(True, alpha=0.3)
# Add mean and median lines
mean_val = np.mean(q_values)
median_val = np.median(q_values)
plt.axvline(mean_val, color='red', linestyle='--', label=f'Mean: {mean_val:.3f}')
plt.axvline(median_val, color='green', linestyle='--', label=f'Median: {median_val:.3f}')
plt.legend()
filename = f"{self.run_dir}/q_table_dist_{game_type}_{player}.png"
plt.savefig(filename)
plt.close()
print(f"Q-table distribution saved to {filename}")
def save_evaluation_results(self, results, game_type):
"""
Visualize evaluation results between different agents
Parameters:
- results: Dict with match results
- game_type: String ("tic_tac_toe" or "connect_4")
"""
# Create dataframe from results
data = []
for match, stats in results.items():
if '(X) vs' in match:
x_agent, o_agent = match.split(' (X) vs ')
o_agent = o_agent.replace(' (O)', '')
data.append({
'X_Agent': x_agent,
'O_Agent': o_agent,
'X_wins': stats['X_wins'],
'O_wins': stats['O_wins'],
'Draws': stats['Draws'],
'Avg_moves': stats['avg_moves'],
'Time': stats['time']
})
df = pd.DataFrame(data)
# 1. Create a win rate comparison chart
plt.figure(figsize=(12, 10))
# Pivot data for visualization
heatmap_data = df.pivot(index='X_Agent', columns='O_Agent', values='X_wins')
# Create heatmap
ax = plt.subplot(2, 2, 1)
sns.heatmap(heatmap_data, annot=True, cmap='YlGnBu', fmt='g', cbar_kws={'label': 'X Wins'})
plt.title('X Player Win Count')
# Create draws heatmap
ax = plt.subplot(2, 2, 2)
heatmap_draws = df.pivot(index='X_Agent', columns='O_Agent', values='Draws')
```

```
sns.heatmap(heatmap_draws, annot=True, cmap='PuRd', fmt='g', cbar_kws={'label': 'Draws'})
plt.title('Draw Count')
# Create average moves heatmap
ax = plt.subplot(2, 2, 3)
heatmap_moves = df.pivot(index='X_Agent', columns='O_Agent', values='Avg_moves')
sns.heatmap(heatmap_moves, annot=True, cmap='Greens', fmt='.1f', cbar_kws={'label': 'Average
Moves'})
plt.title('Average Game Length (Moves)')
# Create execution time heatmap
ax = plt.subplot(2, 2, 4)
heatmap_time = df.pivot(index='X_Agent', columns='O_Agent', values='Time')
sns.heatmap(heatmap_time, annot=True, cmap='Blues', fmt='.2f', cbar_kws={'label': 'Execution
Time (s)'})
plt.title('Execution Time')
plt.tight_layout()
filename = f"{self.run_dir}/evaluation_heatmaps_{game_type}.png"
plt.savefig(filename)
plt.close()
print(f"Evaluation heatmaps saved to {filename}")
# 2. Create a bar chart comparison
plt.figure(figsize=(15, 8))
# Add agent combinations as labels
labels = [f"{row.X_Agent} vs {row.O_Agent}" for _, row in df.iterrows()]
# Plot stacked bars
x = np.arange(len(labels))
width = 0.8
plt.bar(x, df['X_wins'], width, label='X Wins', color='skyblue')
plt.bar(x, df['O_wins'], width, bottom=df['X_wins'], label='O Wins', color='salmon')
plt.bar(x, df['Draws'], width, bottom=df['X_wins'] + df['O_wins'], label='Draws', color='lightgreen')
plt.xlabel('Agent Matchups')
plt.ylabel('Game Outcomes')
plt.title(f'Game Outcome Distribution - {game_type.replace("_", " ").title()}')
plt.xticks(x, labels, rotation=90)
plt.legend()
plt.tight_layout()
filename = f"{self.run_dir}/evaluation_bars_{game_type}.png"
plt.savefig(filename)
plt.close()
print(f"Evaluation bar chart saved to {filename}")
# 3. Create a violin plot for move distribution where data is available
if 'move_history' in results:
plt.figure(figsize=(12, 8))
move_data = []
for match, moves in results['move_history'].items():
for count in moves:
move_data.append({
'Match': match,
'Moves': count
})
```

```
if move_data:
moves_df = pd.DataFrame(move_data)
sns.violinplot(x='Match', y='Moves', data=moves_df)
plt.title(f'Distribution of Game Lengths - {game_type.replace("_", " ").title()}')
plt.xticks(rotation=90)
plt.tight_layout()
filename = f"{self.run_dir}/moves_distribution_{game_type}.png"
plt.savefig(filename)
plt.close()
print(f"Move distribution visualization saved to {filename}")

def visualize_game_state(self, game, move_history=None, filename=None):
"""
Visualize the current state of a game
Parameters:
- game: Game object (TicTacToe or Connect4)
- move_history: List of moves made
- filename: Output filename (optional)
"""
if isinstance(game.__class__.__name__, str) and game.__class__.__name__ == "TicTacToe":
self._visualize_tictactoe(game, move_history, filename)
else:
self._visualize_connect4(game, move_history, filename)
def _visualize_tictactoe(self, game, move_history=None, filename=None):
"""Visualize a Tic-Tac-Toe game"""
plt.figure(figsize=(6, 6))
# Draw grid
plt.plot([1, 1], [0, 3], 'k-')
plt.plot([2, 2], [0, 3], 'k-')
plt.plot([0, 3], [1, 1], 'k-')
plt.plot([0, 3], [2, 2], 'k-')
# Draw X's and O's
for i in range(3):
for j in range(3):
idx = i * 3 + j
if game.board[idx] == 'X':
plt.plot([j+0.2, j+0.8], [2-i+0.2, 2-i+0.8], 'r-', linewidth=2)
plt.plot([j+0.8, j+0.2], [2-i+0.2, 2-i+0.8], 'r-', linewidth=2)
elif game.board[idx] == 'O':
circle = plt.Circle((j+0.5, 2-i+0.5), 0.3, fill=False, ec='b', linewidth=2)
plt.gca().add_patch(circle)
plt.xlim(-0.1, 3.1)
plt.ylim(-0.1, 3.1)
plt.title(f"Tic-Tac-Toe - Current Player: {game.current_player}")
plt.axis('off')
plt.tight_layout()
if filename:
plt.savefig(filename)
else:
filename = f"{self.run_dir}/tictactoe_state.png"
```

```
plt.savefig(filename)
plt.close()
return filename
def _visualize_connect4(self, game, move_history=None, filename=None):
"""Visualize a Connect4 game"""
plt.figure(figsize=(8, 7))
# Draw grid
for i in range(game.rows + 1):
plt.plot([0, game.cols], [i, i], 'b-')
for j in range(game.cols + 1):
plt.plot([j, j], [0, game.rows], 'b-')
# Draw pieces
for i in range(game.rows):
for j in range(game.cols):
if game.board[i][j] == 'X':
circle = plt.Circle((j+0.5, game.rows-i-0.5), 0.4, fc='r', ec='k')
plt.gca().add_patch(circle)
elif game.board[i][j] == 'O':
circle = plt.Circle((j+0.5, game.rows-i-0.5), 0.4, fc='y', ec='k')
plt.gca().add_patch(circle)
plt.xlim(-0.1, game.cols+0.1)
plt.ylim(-0.1, game.rows+0.1)
plt.title(f"Connect 4 - Current Player: {game.current_player}")
plt.axis('off')
plt.tight_layout()
if filename:
plt.savefig(filename)
else:
filename = f"{self.run_dir}/connect4_state.png"
plt.savefig(filename)
plt.close()
return filename
```