Connect 4 Game Report

AI ASSIGNMENT 2

AHMED ABDELMONEIM – 7673
MARIAM GAADALLAH – 7751
FARIDA SHERIF – 7720

1. Introduction

This report details our implementation of an AI agent for the Connect 4 game using various minimax algorithms as required in Assignment 2. Connect 4 is a two-player game where players take turns dropping colored discs into a vertically suspended grid, aiming to connect four of their discs horizontally, vertically, or diagonally. The winner is determined by who has the greater number of connected-fours once the board is full.

Our implementation supports three AI algorithms:

- Minimax without alpha-beta pruning
- Minimax with alpha-beta pruning
- Expectiminimax

2. System Architecture

2.1 Overview

Our system is structured using the Model-View-Controller (MVC) pattern:

- **Model**: Game state representation and algorithms
- **View**: Graphical user interface components
- **Controller**: Game flow management

2.2 Main Components

- **pop_up_menu.py**: Configuration interface for game settings
- **game_controller.py**: Main game loop and event handling
- **minimax.py**: Minimax algorithm with alpha-beta pruning
- **minimax_noprune.py**: Standard minimax algorithm without pruning
- **expectiminimax.py**: Expectiminimax algorithm for probabilistic outcomes
- **heuristics.py**: Evaluation functions for board states

3. Algorithms Implementation

3.1 Minimax without Alpha-Beta Pruning

```
def minimax_noprune(board, depth, maximizingPlayer, piece=AI_PIECE,
                    visualize=False, strategy="combined", graph=None,
                    id_counter=None, node_id=None):
```

This implementation explores the entire search space to the specified depth, evaluating each possible move sequence. Key features include:

- Complete exploration of the game tree to the specified depth

- Move ordering for better performance (though all branches are still explored)
- Transposition table for avoiding redundant calculations

3.2 Minimax with Alpha-Beta Pruning

```
def minimax(board, depth, alpha, beta, maximizingPlayer, piece=AI_PIECE,
            visualize=False, strategy="combined", graph=None,
            id_counter=None, node_id=None):
```

This implementation improves efficiency by pruning branches that cannot influence the final decision:

- Alpha-beta bounds track the best achievable scores for each player
- Branches are pruned when they cannot improve the current best decision
- Move ordering enhances pruning efficiency by exploring promising moves first
- Transposition table caches previously evaluated positions

3.3 Expectiminimax

```
def expectiminimax(board, depth, alpha, beta, maximizing, piece=AI_PIECE,
                   visualize=False, graph=None, id_counter=None,
                   node_id=None, strategy="combined", prune_threshold=0):
```

This algorithm handles the probabilistic nature of piece placement with:

- Probability distribution (0.6 for chosen column, 0.2 for adjacent columns)
- Weighted expectation calculation for each possible outcome
- Alpha-beta pruning adapted for probabilistic scenarios
- Zobrist hashing for efficient state representation
- Advanced pruning with heuristic thresholds

4. Heuristic Function

4.1 Design Philosophy

Our heuristic function evaluates board states based on multiple strategic factors important in Connect 4:

```
def combined_heuristic(board, piece):
    # Implementation details...
```

The function returns higher values for positions favorable to the AI and lower values for positions favorable to the human player.

4.2 Components

The heuristic weights different strategic elements:

```
WEIGHTS = {
    "center_control": 6,
    "reward_4": 10000,
    "reward_3": 100,
    "reward_2": 10,
    "reward_1": 1,
    "block_3": 10000000,
    "block_2": 100,
    "trap_bonus": 1500,
    "isolation_penalty": 50,
}
```

These weights reflect the relative importance of:

1. **Center Control**: Pieces in the center column are more valuable as they provide more connection opportunities
2. **Pattern Recognition**: Evaluating developing connections (1, 2, 3, or 4 pieces in a row)
3. **Defensive Awareness**: Heavily penalizing states where the opponent is close to winning
4. **Trap Creation**: Rewarding positions that create multiple simultaneous threats
5. **Piece Connectivity**: Discouraging isolated pieces that don't contribute to potential connections

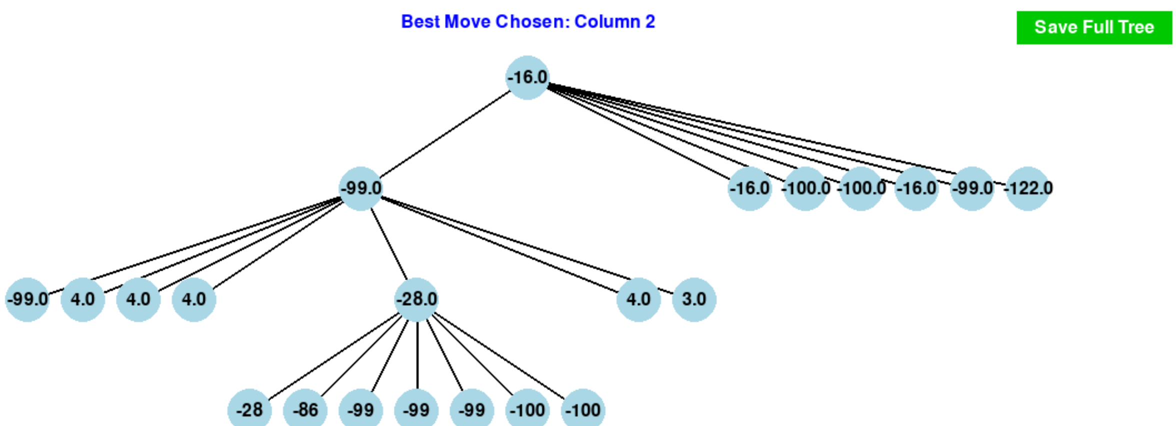
The function analyzes all possible 4-cell windows on the board, tallying scores for offensive patterns, defensive needs, and strategic positioning.

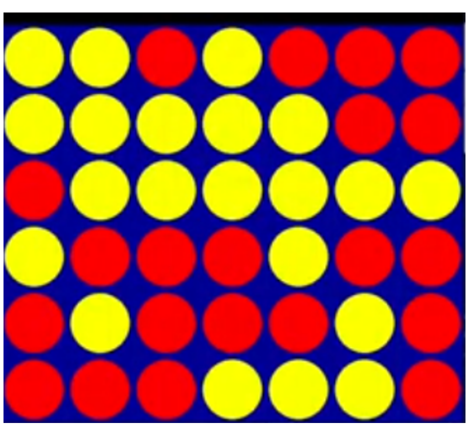
Hashing (Zobrist Hash):
The board's string representation is hashed using MD5 to produce a unique identifier. This hash is used as a key in the transposition table (a dictionary) to cache results from the expectiminimax function, which avoids recalculating already-evaluated board states.
Heuristic Weights:
A dictionary of weights defines various strategic factors such as center control, rewards for having 2, 3, or 4 pieces aligned, blocking opponent moves, and penalties for isolation. These weights are used in the evaluation function to score the board by iterating over precomputed 4-cell "windows" that could result in wins or threats. The specific numerical values (for example, a reward of 10,000 for a win condition or a very high penalty for a likely opponent win) are assumptions that guide the AI's decision-making and balance offensive versus defensive moves.
--- Heuristic Weights ---
WEIGHTS = {
"center_control": 6,
"reward_4": 10000,
"reward_3": 100,
"reward_2": 10,
"reward_1": 1,
"block_3": 10000000,
"block_2": 100,
"trap_bonus": 1500,
"isolation_penalty": 50,
}

Sample Game Execution

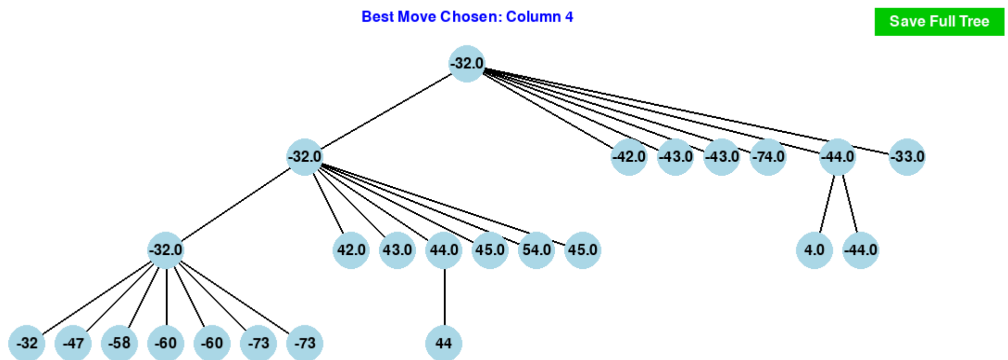
minimax no prune - depth = 3

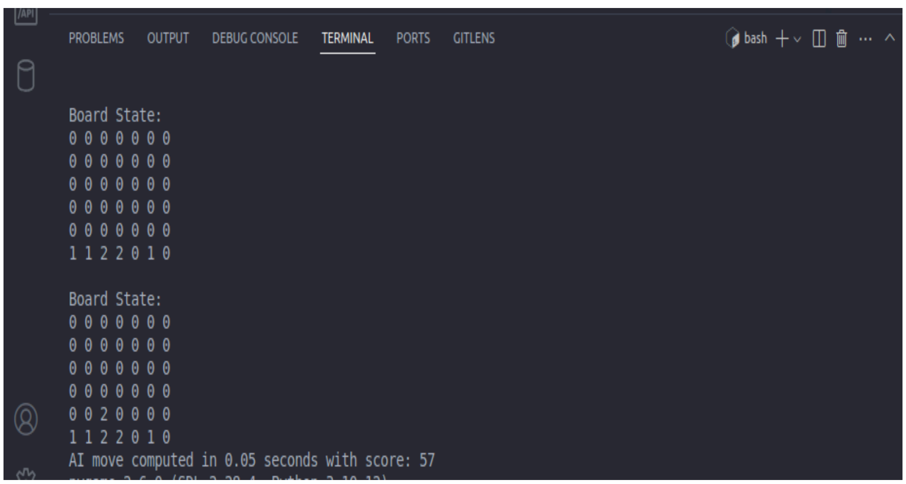
```
Board State:
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 2 1 0 0 0 0
0 2 1 1 0 0 0

Board State:
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 2 0 0 0 0 0
0 2 1 0 0 0 0
0 2 1 1 0 0 0
AI move computed in 0.18 seconds with score: -42
```





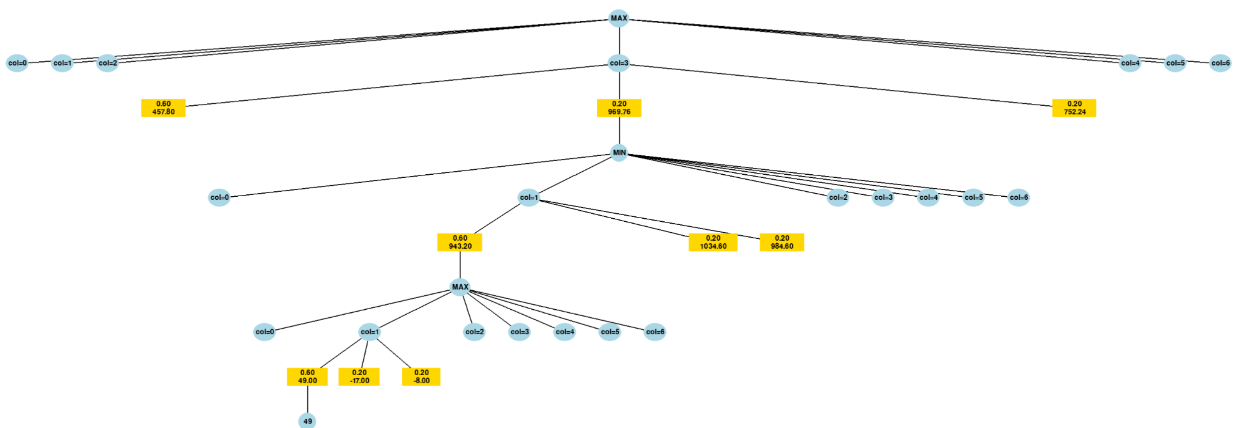
minimax with pruning - depth = 3

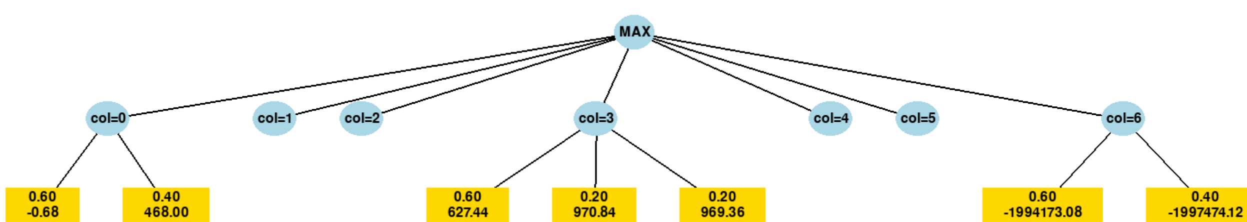




Expectiminimax - depth = 3

```
Board State:
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 1 2 0 0
AI move computed in 0.89s with score: 165.01600000000002
```





Minimax no pruning - depth 7

```
Board State:
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 2 0 1 0 0 0
AI move computed in 46.39s with score: -237
```

Minimax with pruning - depth 8

```
AI suggests column 6 with score -574

Board State:
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 1 0 0 2 0
AI move computed in 12.04s with score: -574
```

Expectiminimax - depth 5

```
Board State:
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 1 0 2 0 0 0
AI move computed in 16.50s with score: 1807.34976
```

7. Data Structures

7.1 Game Board Representation

The board state is stored as a flat list of strings, where each element represents a cell on the board. This simple representation allows for easy conversion into a string (by concatenation) that is then hashed.

- Empty cells are represented by the EMPTY constant
- Player's pieces are represented by the PLAYER_PIECE constant
- AI's pieces are represented by the AI_PIECE constant

This representation allows for efficient state copying and manipulation.

7.2 Transposition Tables

We implemented transposition tables as Python dictionaries to cache previously evaluated positions:

```
_transposition_table_ab = {}  # For minimax with alpha-beta
_transposition_table = {}     # For minimax without pruning
_trans_table_em = {}          # For expectiminimax
```

For expectiminimax, we use Zobrist hashing to generate unique identifiers for board states:

```
def zobrist_hash(board_str):
    return hashlib.md5(board_str.encode('utf-8')).hexdigest()
```

7.3 Search Tree Visualization

For visualization, we use NetworkX to build a directed graph representation of the search tree:

```
graph = nx.DiGraph()
```

This graph captures the decision-making process of the AI, with nodes representing board states and edges representing moves.

8. Optimizations

8.1 Move Ordering

To improve alpha-beta pruning efficiency, we implemented heuristic-based move ordering:

```
children.sort(key=lambda x: x[2], reverse=maximizingPlayer)
```

This ensures that promising moves are explored first, potentially increasing pruning opportunities.

8.2 Precomputed Windows

We precompute all possible 4-cell windows that could form a winning connection:

```
WINDOWS = generate_windows()
```

This avoids redundant calculation during heuristic evaluation.

8.3 Pruning Threshold

In the expectiminimax implementation, we added a heuristic pruning threshold to further reduce the search space:

```
if prune_threshold > 0:
    approx_score = evaluate_board(sub_board, piece, strategy)
    if maximizing and approx_score < alpha - prune_threshold:
        continue
    if not maximizing and approx_score > beta + prune_threshold:
        continue
```

8.4 Parallel Visualization

To avoid slowing down gameplay, the tree visualization runs in a separate process:

```
p = multiprocessing.Process(
    target=draw_graph_process, args=(graph, best_move)
)
p.daemon = True
p.start()
```

9. Findings and Observations

1. **Search Depth Trade-offs**: Increasing search depth significantly improves play quality but exponentially increases computation time. For practical gameplay, depth=4 with alpha-beta pruning provides the best balance of performance and strong play.
2. **Heuristic Importance**: The quality of the heuristic function is more important than extra search depth. Our evaluation function's emphasis on defensive play and trap creation enables strong performance even at limited depths.
3. **Algorithm Comparison**:
    - Alpha-beta pruning is significantly more efficient than standard minimax
    - Expectiminimax produces more conservative play due to accounting for uncertainty
    - At equal depths, minimax with pruning makes decisions faster but expectiminimax better handles the probabilistic nature of the game

4. **Position Evaluation**: Center control and blocking imminent threats proved to be the most critical factors in effective evaluation.

10. Conclusion

Our implementation successfully meets all the project requirements, providing a Connect 4 game with three different AI algorithms and visualizations of the decision-making process. The most effective variant is minimax with alpha-beta pruning, which achieves the best balance of computation efficiency and strong gameplay.

The comparison between algorithms demonstrates the trade-offs between thoroughness, speed, and handling uncertainty in game-playing AI systems. Our heuristic function effectively captures the strategic elements of Connect 4, enabling strong play even with limited search depth.

Future improvements could include learning-based heuristic weights, further optimization of the expectiminimax implementation, and parallel search for deeper look-ahead.