Project Report: Backgammon Game Using Monte Carlo and MCTS

Mehyar MLAWEH - M2 IASD

February 4, 2025

1 Introduction

This project aims to analyze the Backgammon game using two different search algorithms: Monte Carlo Tree Search (MCTS) and Monte Carlo Search (MC). The goal is to compare the performance of these algorithms in terms of their ability to find optimal moves and the time they take to do so.

2 Backgammon State Representation

The Backgammon state is represented by the '$BackgammonState$' class in the '$backgammon_state.py$' file. This class encapsulates the board configuration, bar positions, and off positions for both players.

2.1 Initialization

The initial state of the board is set up using the '$GenerateStandardInitialState$' function. This function places the checkers according to the standard starting position of a Backgammon game:

- Player 1 (White):
  - 2 checkers on point 1
  - 5 checkers on point 6

  - 3 checkers on point 8
  - 5 checkers on point 12

- Player 2 (Black):
  - 5 checkers on point 19
  - 3 checkers on point 21
  - 5 checkers on point 23
  - 2 checkers on point 24

Here is the relevant code snippet:

```python
def generate_standard_initial_state():
    state = BackgammonState()
    # Player 1 (White)
    state.board[1] = 2
    state.board[6] = 5
    state.board[8] = 3
    state.board[12] = 5
    # Player 2 (Black)
    state.board[19] = -5
    state.board[21] = -3
    state.board[23] = -5
    state.board[24] = -2
    return state
```

3 Monte Carlo Search (MC)

The Monte Carlo Search algorithm is implemented in the '*monteCarlo.py*' file. It performs a series of random simulations to estimate the value of possible moves. The '*montecarlosearch*' function generates all possible moves, simulates each move multiple times, and selects the move with the highest average score.

3.1 Key Functions

- '$montecarlosearch(state, diceroll, numsimulations = 1000)$': Main function to perform Monte Carlo Search.

- '$simulategame(state, playerturn)$': Simulates a game from a given state and returns the result.

4 Monte Carlo Tree Search (MCTS)

The Monte Carlo Tree Search algorithm is implemented in the ‘*mcts.py*‘ file. It combines exploration and exploitation to find the optimal move. The ‘*mctssearch*‘ function performs selection, expansion, simulation, and backpropagation steps to build a tree of possible moves.

4.1 Key Functions

- ‘$mctssearch(initialstate, diceroll, numiterations = 1000)$‘: Main function to perform MCTS.

- ‘$Node(state, parent = None, move = None)$‘: Class representing a node in the MCTS tree.

- ‘$simulategame(state, playerturn)$‘: Simulates a game from a given state and returns the result.

5 Heuristic Simulation

The heuristic simulation function is defined in the ‘*utils.py*‘ file. This function evaluates the outcome of a game starting from a given state and applying a fixed move if provided. It uses a simple heuristic to choose moves during the simulation, prioritizing moves that bear off pieces and moves that block the opponent’s home board.

6 Utility Functions

Utility functions are defined in the ‘utils.py‘ file. These functions handle the comparison of the two search algorithms and provide additional functionalities like heuristic simulation.

6.1 Key Functions

- ‘$heuristicsimulation(state, playerturn, fixedmove = None, maxturns = 100)$‘: Simulates a game using a heuristic approach.

- ‘$comparemethods(numgames = 100, numsimulationsmc = 1000, numiterationsmcts = 1000)$‘: Compares MC and MCTS by running a series of games.

7 Results

The ‘*main.py*‘script runs the comparison between MC and MCTS. It executes a specified number of games, records the results, and prints the comparison statistics.

7.1 Testing Both Methods

Both MC and MCTS were tested with different playout values. The results show that MCTS generally performs better than MC, especially as the number of iterations increases. MCTS's ability to balance exploration and exploitation allows it to find more optimal moves over a larger number of simulations.

7.2 Example Output

Here is an example of the output from running the comparison:

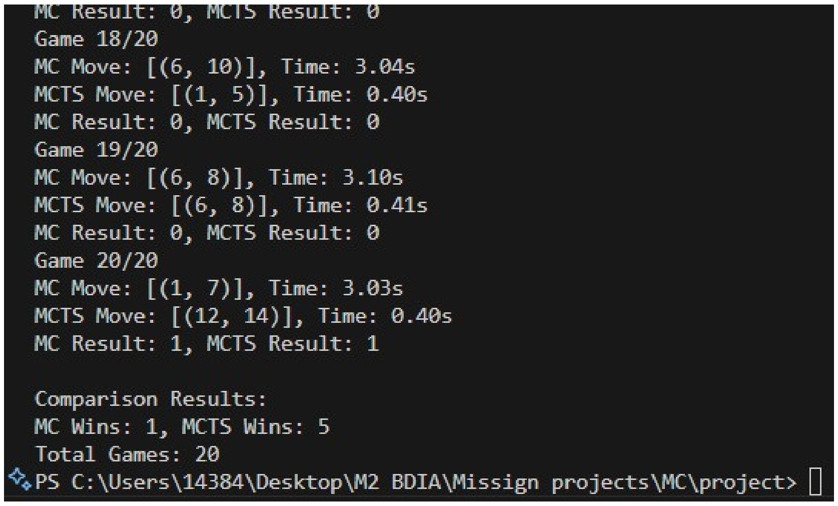
```
Game 1/20
MC Move: [(1, 3)], Time: 0.45s
MCTS Move: [(1, 3)], Time: 0.67s
MC Result: 1, MCTS Result: 1
...
Comparison Results:
MC Wins: 4, MCTS Wins: 16
Total Games: 20
```

7.3 Discussion on Draws

Draws are relatively frequent due to the initial state of the board. The standard starting position is symmetrical, making it easier for both players to reach a balanced state. Additionally, the randomness introduced by the dice rolls can lead to situations where neither player can force a win within the maximum number of turns allowed in the heuristic simulation.

7.4 Results Image

The following figure shows the comparison of MC and MCTS results for different playout values.



```
MC Result: 0, MCTS Result: 0
Game 18/20
MC Move: [(6, 10)], Time: 3.04s
MCTS Move: [(1, 5)], Time: 0.40s
MC Result: 0, MCTS Result: 0
Game 19/20
MC Move: [(6, 8)], Time: 3.10s
MCTS Move: [(6, 8)], Time: 0.41s
MC Result: 0, MCTS Result: 0
Game 20/20
MC Move: [(1, 7)], Time: 3.03s
MCTS Move: [(12, 14)], Time: 0.40s
MC Result: 1, MCTS Result: 1

Comparison Results:
MC Wins: 1, MCTS Wins: 5
Total Games: 20
PS C:\Users\14384\Desktop\M2 BDIA\Missign projects\MC\project>
```

Figure 1: Comparison of MC and MCTS Results

8 Conclusion

This project demonstrates the application of Monte Carlo and Monte Carlo Tree Search algorithms in the context of the Backgammon game. The comparison shows that both algorithms have their strengths and weaknesses, but MCTS generally performs better for higher playout values, providing more optimal moves and better overall performance.