Pacman Multi-Agent Search

*Comparing Reflex, Minimax, AlphaBeta, and MonteCarloTreeSearch Agents

Md Omar Faruque Md Zaman

***Abstract*—In this project, we embark on a comparative analysis of four multi-agent search algorithms applied to the classic game of Pacman. The algorithms under investigation are Reflex Agent, Minimax Agent, Alpha-Beta Agent, and Monte Carlo Tree Search (MCTS) Agent. We implement each algorithm and evaluate its performance based on average score, win rate, and average time per run across three game layouts of varying complexity. The results shed light on the strengths and weaknesses of each algorithm in the context of autonomous Pacman gameplay. The MCTS Agent emerges as the top-performing algorithm, achieving the highest average score and win rate across all layouts. Our findings emphasize the importance of selecting the appropriate algorithm based on the specific requirements and constraints of the game scenario, taking into account factors such as layout complexity, available computational resources, and the desired balance between performance and efficiency.**

***Index Terms*—Pacman, Multi-Agent, Adversarial, Search**

I. Introduction

Pac-Man, the iconic arcade game, has captivated players for decades. The game's premise is simple yet engaging: guide Pac-Man through a maze, gobbling up food pellets while avoiding the relentless pursuit of ghosts. In this project, we delved into the fascinating world of AI algorithms and their application to autonomous Pac-Man gameplay. We explored four distinct algorithms, each with its own unique decision-making process. Our motivation was threefold:

- **Understanding Principles:** To comprehend the fundamental principles underlying multi-agent search algorithms.
- **Application in Real-World Scenarios:** To apply these algorithms within a real-world game scenario, using the Pacman game as a testbed.
- **Performance Analysis:** To analyze and compare the performance of the algorithms to evaluate their effectiveness and efficiency.

This paper builds on the **UC Berkeley Pacman AI project** [1], implementing advanced AI agents to navigate and solve the game's challenges.

Through this project, we made several key contributions:

- **Enhanced AI Agents:** We expanded the multiAgents.py to include new AI agents like ReflexAgent, MinimaxAgent, AlphaBetaAgent, and MonteCarloTreeSearchAgent, enabling a comprehensive evaluation of these search algorithms.
- **Automated Performance Tracking:** We developed automated scripts to systematically collect performance data such as win rates, scores, and computation times, facilitating consistent and reliable analysis.
- **Visual Performance Comparisons:** We used matplotlib for creating visual representations, such as graphs, assists in the comparative analysis of each agent's performance across different game layouts.

The remainder of this paper is structured as follows. Section II describes the methodology, including the UCB Project repository structure and detailed explanations of each algorithm's implementation. Section III presents the results and discussion, evaluating the performance of each algorithm and discussing their implications. Finally, Section IV concludes the paper, summarizing the key findings and outlining future directions in this project.

II. Methodology

A. UCB Project repository structure

To conduct our experiments, we utilized the Pacman AI project framework developed by UC Berkeley. This framework provides a robust and flexible environment for implementing and testing various AI algorithms in the context of the Pacman game. The framework includes several key components that we leveraged in our project:

1) **Game Engine:** The Pacman game engine, implemented in `pacman.py`, handles the game logic, state transitions, and rendering. It provides a `GameState` class that encapsulates the current state of the game, including the positions of Pacman, ghosts, and food pellets, as well as the game score and other relevant information.
2) **Agent Classes:** The framework defines a hierarchy of agent classes, with the base class `Agent` in `game.py`. We implemented our multi-agent search algorithms by extending the `MultiAgentSearchAgent` class in `multiAgents.py`. This class provides hooks for implementing the `getAction` method, which returns the chosen action for an agent given the current game state.
3) **Game Layouts:** The framework includes a set of predefined game layouts in the `layouts` directory, representing different levels of complexity and difficulty. These layouts are loaded by the game engine and used to initialize the game state. In our experiments, we focused on three specific layouts: `testClassic`, `smallClassic`, and `mediumClassic`, which vary in grid size, number of ghosts, wall density, and open space.

4) **Evaluation Functions:** The framework allows for the specification of evaluation functions, which assign scores to game states. These evaluation functions guide the decision-making process of the agents.

To integrate our multi-agent search algorithms into the Pacman framework, we focused our implementation efforts on the `multiAgents.py` file. We used the `MultiAgentSearchAgent` class to define our `ReflexAgent`, `MinimaxAgent`, `AlphaBetaAgent`, and `MonteCarloTreeSearchAgent`. These agents override the `getAction` method to implement their respective decision-making algorithms.

B. Algorithm Implementations

1. Reflex Agent:: The Reflex Agent implementation can be found in the `ReflexAgent` class in `multiAgents.py`. The key components of the implementation are:

- The `getAction` method, which collects legal moves, evaluates each move using the `evaluationFunction`, and chooses the best action based on the highest score.
- The `evaluationFunction` method, which evaluates a game state by considering factors such as the successor game state's score, food proximity, and ghost proximity. It assigns higher scores to states with closer food pellets and scared ghosts, while penalizing states with close active ghosts.

The pseudo algorithm for the Reflex Agent is given in Algorithm 1.

Algorithm 1 Reflex Agent - getAction

```
legalMoves ← getLegalActions(currentState)
scores ← []
for each action in legalMoves do
    successorState ← generateSuccessor(currentState, action)
    score ← evaluationFunction(successorState)
    scores.append(score)
end for
bestScore ← max(scores)
bestIndices ← [index for index, score in enumerate(scores) if score == bestScore]
chosenIndex ← random.choice(bestIndices)
return legalMoves[chosenIndex]
```

2. Minimax Agent:: The Minimax Agent is implemented in the `MinimaxAgent` class. The main parts of the implementation are:

- The `getAction` method, which initiates the minimax search from Pacman's perspective at the current game state and depth 0.
- The `minimax` function, which recursively computes the minimax value for each agent at each depth level. It maximizes the value for Pacman and minimizes the value for ghosts. The recursion terminates when the game ends or the depth limit is reached, at which point the state is evaluated using the `evaluationFunction`.

The pseudo algorithm for the Minimax Agent is given in Algorithm 2.

Algorithm 2 Minimax Agent - minimax

```
if gameEnds(currentState) or currentDepth == maxDepth then
    return evaluationFunction(currentState)
end if
if agentIndex == 0 then
    bestValue ← −∞
    for each action in getLegalActions(currentState, agentIndex) do
        successorState ← generateSuccessor(currentState, agentIndex, action)
        value ← minimax(successorState, (agentIndex + 1)
        bestValue ← max(bestValue, value)
    end for
    return bestValue
else
    bestValue ← +∞
    for each action in getLegalActions(currentState, agentIndex) do
        successorState ← generateSuccessor(currentState, agentIndex, action)
        if agentIndex == numAgents - 1 then
            value ← minimax(successorState, 0, currentDepth + 1)
        else
            value ← minimax(successorState, agentIndex + 1, currentDepth)
        end if
        bestValue ← min(bestValue, value)
    end for
    return bestValue
end if
```

3. Alpha-Beta Agent:: The Alpha-Beta Agent implementation is located in the `AlphaBetaAgent` class. The key components are:

- The `getAction` method, which starts the alpha-beta search with initial alpha and beta values.
- The `maxValue` function, which computes the maximum value for Pacman's moves while updating the alpha value and pruning branches when the value exceeds beta.
- The `minValue` function, which computes the minimum value for the ghosts' moves while updating the beta value and pruning branches when the value is less than alpha.

The pseudo algorithm for the Alpha-Beta Agent is given in Algorithm 3.

4. Monte Carlo Tree Search (MCTS) Agent:: The MCTS Agent is implemented in the `MonteCarloTreeSearchAgent` class, with the core

Algorithm 3 Alpha-Beta Agent - getAction

```
if gameEnds(currentState) or currentDepth == 0 then
    return evaluationFunction(currentState)
end if
if agentIndex == 0 then
    bestValue ← −∞
    for each action in getLegalActions(currentState, agentIndex) do
        successorState ← generateSuccessor(currentState, agentIndex, action)
        value ← minValue(successorState, currentDepth, alpha, beta, 1)
        bestValue ← max(bestValue, value)
        if bestValue > beta then
            return bestValue
        end if
        alpha ← max(alpha, bestValue)
    end for
    return bestValue
else
    bestValue ← +∞
    for each action in getLegalActions(currentState, agentIndex) do
        successorState ← generateSuccessor(currentState, agentIndex, action)
        if agentIndex == numAgents - 1 then
            value ← maxValue(successorState, currentDepth - 1, alpha, beta)
        else
            value ← minValue(successorState, currentDepth, alpha, beta, agentIndex + 1)
        end if
        bestValue ← min(bestValue, value)
        if bestValue < alpha then
            return bestValue
        end if
        beta ← min(beta, bestValue)
    end for
    return bestValue
end if
```

functionality in the `MCTSNode` class. The main components of the implementation are[1]:

- The `MCTSNode` class, which represents a node in the MCTS tree. It contains methods for node initialization, expansion, selection, simulation (rollout), and backpropagation.
- The `getAction` method in the `MonteCarloTreeSearchAgent` class, which initializes the MCTS root node and performs a specified number of MCTS iterations before selecting the best action.
- The `MCTSAction` method, which executes the MCTS iterations and returns the best action based on the average values and visit counts of the child nodes.

The pseudo algorithm for the MCTS Agent is given in Algorithm 4.

Algorithm 4 MCTS Agent - getAction

```
rootNode ← initializeNode(currentState)
for iteration = 1 to maxIterations do
    selectedNode ← treePolicy(rootNode)
    reward ← defaultPolicy(selectedNode.state)
    backpropagate(selectedNode, reward)
end for
bestChild ← bestChild(rootNode)
return bestChild.action
```

These pseudo algorithms provide a high-level overview of how each algorithm is implemented in our project. The Reflex Agent relies on predefined rules and heuristics, while the Minimax and Alpha-Beta Agents utilize recursive search with value maximization and minimization. The MCTS Agent builds a search tree incrementally, balancing exploration and exploitation to make informed decisions.

III. Results

A. Experimental Setup

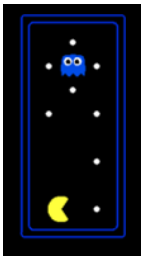
The algorithms were tested on three different game layouts: `testClassic`, `smallClassic`, and `mediumClassic`. These layouts varied in terms of grid size, number of ghosts, wall density, and open space, representing different levels of difficulty. Table I summarizes the features of each layout.

TABLE I: Layout Features

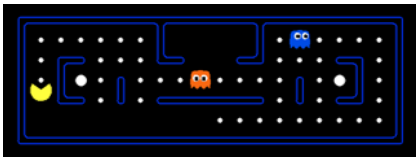
Feature	testClassic	smallClassic	mediumClassic
Grid Size	Small	Medium	Large
Number of Ghosts	1	2	2
Wall Density	Low	Medium	High
Open Space	High	Medium	Low
Estimated Difficulty	Easy	Medium	Hard

We ran each algorithm 10 times on each layout, and the measured performance based on three metrics: average score, win rate, and average time per run. We developed automated script (`testing.py`) to systematically collect performance data such as win rates, scores, and computation times, facilitating consistent and reliable analysis. The algorithms were run using the command-line interface provided by the framework, with the `--no-graphics` to disable graphical output and speed up the execution. Additionally, we used matplotlib for creating visual representations, such as graphs, which assist in the comparative analysis of each agent's performance across different game layouts. Figure 1 shows the visual representation of the three layouts used in the experiments.

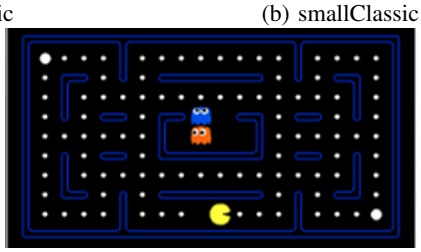
[1]The MCTS agent implementation was inspired by the work of Prabin Rath, available at https://github.com/prabinrath/Monte-Carlo-Tree-Search-Pac-Man/tree/main.



(a) testClassic



(b) smallClassic



(c) mediumClassic

Fig. 1: Game Layouts

B. ReflexAgent Performance

The ReflexAgent demonstrated good performance on the `testClassic` layout, achieving a perfect win rate of 100% and an average score of 563.20. However, its performance decreased on the more complex layouts, with a win rate of 50% on both `smallClassic` and `mediumClassic`. The average scores for these layouts were 537.00 and 1069.40, respectively. The average time per run for the ReflexAgent was relatively low, ranging from 0.09s to 0.39s across the layouts. Figure 2 shows the detailed performance metrics for the ReflexAgent on each layout.

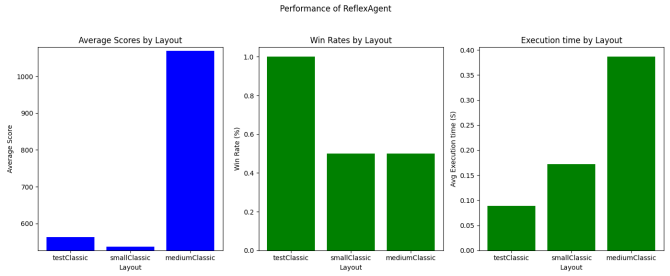


Fig. 2: ReflexAgent Performance

C. MinimaxAgent Performance

The MinimaxAgent achieved a perfect win rate of 100% on the `testClassic` layout, with an average score of 527.00. However, its performance significantly deteriorated on the `smallClassic` and `mediumClassic` layouts, with win rates of only 10% and negative average scores of -130.60 and -356.40, respectively. The average time per run for the MinimaxAgent increased with the complexity of the layouts, ranging from 0.31s to 6.93s. Figure 3 illustrates the performance metrics for the MinimaxAgent on each layout.

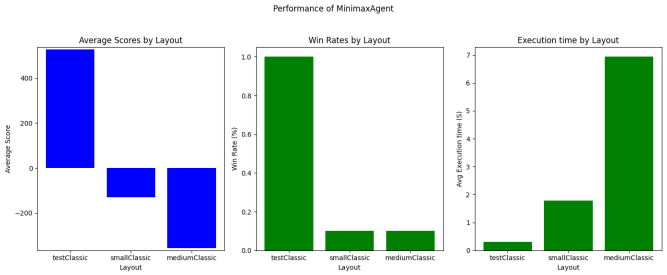


Fig. 3: MinimaxAgent Performance

D. AlphaBetaAgent Performance

The AlphaBetaAgent exhibited a similar performance pattern to the MinimaxAgent. It achieved a perfect win rate of 100% on the `testClassic` layout, with an average score of 503.20. However, it struggled on the `smallClassic` and `mediumClassic` layouts, with win rates of 0% and negative average scores of -72.10 and -740.80, respectively. The average time per run for the AlphaBetaAgent also increased with the complexity of the layouts, ranging from 0.35s to 8.61s. Figure 4 presents the performance metrics for the AlphaBetaAgent on each layout.

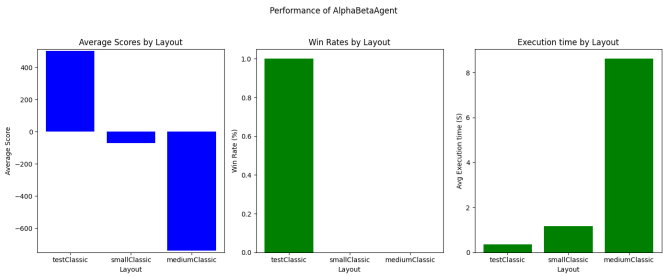


Fig. 4: AlphaBetaAgent Performance

E. MonteCarloTreeSearchAgent Performance

The MonteCarloTreeSearchAgent demonstrated the best overall performance among the evaluated algorithms. It achieved perfect win rates of 100% on both the `testClassic` and `smallClassic` layouts, with average scores of 554.20 and 1187.70, respectively. On the `mediumClassic` layout, it achieved a win rate of 50% and an average score of 655.30. However, the MonteCarloTreeSearchAgent also had the highest average time per run, ranging from 1.44s to 27.91s, indicating a trade-off between performance and computational efficiency. Figure 5 displays the performance metrics for the MonteCarloTreeSearchAgent on each layout.

F. Performance Comparison

Figure 6 presents a comprehensive comparison of the algorithms' performance on the three layouts. The MonteCarloTreeSearchAgent most of the time outperformed the other

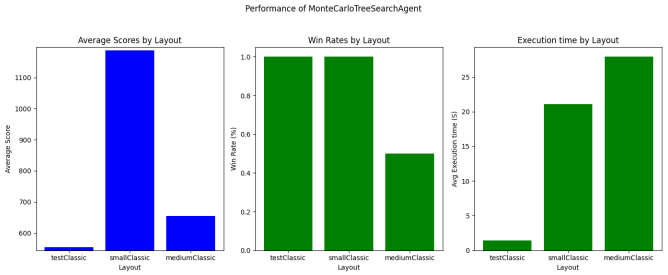
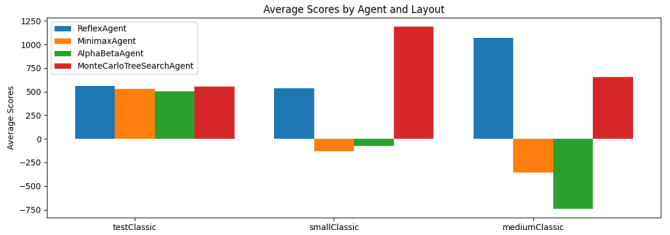


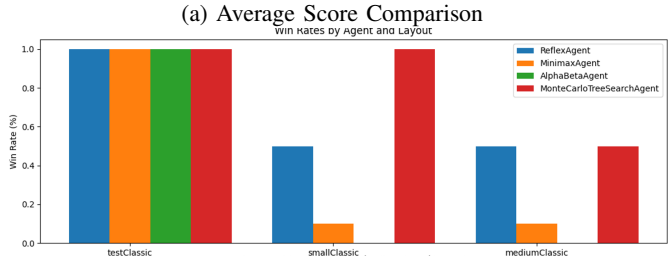
Fig. 5: MonteCarloTreeSearchAgent Performance

algorithms in terms of average score and win rate, demonstrating its effectiveness in the Pac-Man domain. However, it also had the highest execution time, which may be a consideration in real-time decision-making scenarios.

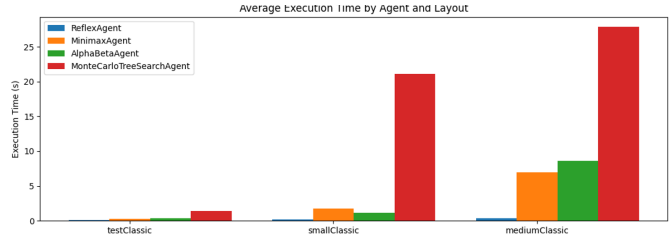
The ReflexAgent showed good performance on the simpler `testClassic` layout but struggled on the more complex layouts. The MinimaxAgent and AlphaBetaAgent exhibited similar performance patterns, achieving perfect win rates on the `testClassic` layout but failing to perform well on the more challenging layouts.



(a) Average Score Comparison



(b) Win Rate Comparison



(c) Execution Time Comparison

Fig. 6: Performance Comparison

Table II summarizes the overall performance of the agents across all test runs and all layouts.

It is evident from table II that the MonteCarloTreeSearchAgent not only scored the highest on average but also maintained the highest win rate, indicating its robustness in complex game scenarios.

TABLE II: Overall Performance of Agents

Agent	Avg Score	Win Rate (%)	Time (s)
ReflexAgent	723.20	67	0.22
MinimaxAgent	13.33	40	3.01
AlphaBetaAgent	-103.23	33	3.37
MonteCarloTreeSearch	799.07	83	16.81

Overall, the results highlight the trade-off between performance and computational efficiency among the agents. The MonteCarloTreeSearchAgent achieves the best performance but at the cost of higher computational time, while the ReflexAgent provides a good balance between performance and efficiency. The MinimaxAgent and AlphaBetaAgent struggle to perform well in complex layouts and have moderate computational requirements

IV. Conclusion

In our project, we evaluated the efficacy of four multi-agent search algorithms through autonomous Pacman gameplay. We highlighted the distinct advantages and limitations of each algorithm, emphasizing the importance of choosing the right algorithm tailored to the unique demands and constraints of the gameplay scenario. Our findings showed that the Monte Carlo Tree Search (MCTS) Agent was the most proficient, illustrating the advantages of employing simulation-based decision-making within intricate gaming environments.

For future work, we could focus on developing more advanced evaluation functions for the Minimax and Alpha-Beta Agents to enhance their effectiveness in complex scenarios. Additionally, incorporating domain-specific knowledge and heuristics into these algorithms might further improve their decision-making capabilities.

References

[1] UC Berkeley Pacman AI Project. https://inst.eecs.berkeley.edu/~cs188/sp22/project2/

[2] Russell, S. J., & Norvig, P. (2021). Artificial Intelligence: A Modern Approach (4th ed.). Pearson.

[3] Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., ... & Colton, S. (2012). A survey of monte carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in Games, 4(1), 1-43.