Reflex-MCTS: A Hybrid Agent for Pacman

Prabin Kumar Rath, Rahil Hastu, Sai Nikhil Guntur, and Austin Porter

Abstract—**This report presents an implementation of the Monte-Carlo Tree Search (MCTS) method customized for playing the Pacman game with random ghosts. We present an agent that utilizes the effectiveness of MCTS for exploration and exploitation but switches to choosing customized reflex actions at critical zones. Throughout this report, we demonstrate that our agent yields consistent win and performance results as opposed to conventional tree-based methods, such as Minimax, Expectimax, and Alpha-Beta Pruning. The report analyzes different layouts where our agent succeeds and identifies possible layouts where our agent may perform sub-optimally.**

I. INTRODUCTION

There are several tree-based search algorithms for designing agents to play in different game domains. Most of these methods allow an agent to look ahead in the game and determine utilities at terminal states or evaluations at preempted states. The agent chooses actions that would yield high utility and score based on these evaluations.

A Minimax agent assumes that the adversary, or opponent, will act optimally. Hence, the agent chooses actions that lead to the highest possible utility. Since the ghosts in our Pacman domain act randomly, this agent will often play too cautiously and struggle to act optimistically. Another common bottleneck with Minimax is that the tree grows exponentially for games with large state spaces. In contrast to Minimax, an Alpha-Beta agent utilizes pruning strategies for handling large state spaces. The agent never expands certain tree branches, assuming that the opponent acts optimally. This reduction in width exploration buys additional compute for exploring higher depths compared to the standard Minimax algorithm. On the other hand, an Expectimax agent accounts for the fact that the adversary may act sub-optimally. The agent models the likelihood of the adversary taking different actions and attempts to maximize its expected utility.

All three of these conventional methods have a commonality with their inability to handle large state spaces. Real-world search problems are rarely low-dimensional. Handling problems with huge state space was a bottleneck for AI research until 2006 when the Monte-Carlo Tree Search (MCTS) method was proposed by Chaslot et al.[1]. For any given state, MCTS uses rollouts, and the game is simulated from that state onwards using random actions till termination or timeout. This helps to evaluate actions over a short horizon by expanding the tree in the most promising direction.

We found that a vanilla MCTS implementation results in interesting behaviors of the agent, and intuitive strategies emerge within a few iterations of the algorithm. However, we also found it ineffective when ghosts were within the agent's proximity. After thorough analysis, we combined reflex behavior with the MCTS algorithm to overcome this challenge and designed a hybrid agent that performs significantly better than conventional search methods and vanilla MCTS.

The exact algorithm used by our agent will be discussed in detail in Section II. We extensively evaluate our agent on 90 different layouts categorized based on grid dimension into small, medium, and large. Each category has variations in wall structures, number of ghosts, and number of power pallets. For this analysis, we compare the performance of our agent with the Minimax, Alpha-Beta, and Expectimax algorithms. Section III presents our analysis results in terms of win percentages, compute time, and average scores on the three layout categories.

II. Technical Approach

MCTS is a search algorithm that explores new states and, at the same time, exploits its past experiences to meticulously select its next move. The algorithm initializes a tree with the root node representing the current state. It has four steps that are iterated a specified number of times. Once the iterations are over, the best action is selected from the root node that yields the highest utility from the current state.

1. *Selection*: A function is used to select the next action among all feasible actions from any given tree node. This function determines whether to exploit a path that has been explored before or explore a new path to find a better strategy. We use the Upper Confidence Bound (UCB1) [2] as the selection function in our implementation, where the node with the highest UCB1 score is chosen by the algorithm. In the equation presented below, V_i is the average utility of the child node, N is the number of visits for the current node, and n_i is the number of visits for the child node.

$$UCB1 = V_i + c\sqrt{\frac{lnN}{n_i}}$$

2. *Expansion*: If the selected node is a leaf node and has been visited at least once, then successor states for the node are obtained for each legal action at that state, and new nodes are added to the tree.
3. *Rollout*: If the selected node is a leaf node and has

never been visited earlier, then the game is simulated with random actions until termination or timeout. Timeout occurs when the number of simulation steps exceeds a specified threshold.

4. *Backpropagation*: An evaluation function is used to determine the utility of the rollout state. This utility is added to the value of all the nodes that are part of the trajectory from the root to the state. For each node, the visit count is also incremented by a unit.

With successive iterations of MCTS, the agent should find actions most likely to end in a winning scenario. However, our implementation of the vanilla MCTS always lost in the Pacman world. After careful observation, we found that for states where the agent is in close proximity to the ghosts, almost all of the random action rollouts lead to terminal states where the agent dies. Our rollout evaluation function assigns a fixed negative utility to losing terminal states; hence, by the innate property of the UCB1 function, all children of the root get the same utility. There is no distinction between actions for the root node; thus, MCTS fails to determine the best action. For the remaining report, we use the alias “critical zone” for states where ghosts are nearby the agent.

After sufficient gameplays with vanilla MCTS, we found that our agent's priority at critical zones should be to run away from the ghosts as soon as possible. Given that we observe similar utilities for multiple actions, we override the final selection function to choose a reflex action from the set of actions with the highest utility. The successor state for this action has the highest sum of manhattan distance from active ghosts and hence helps the agent to escape critical zones. A few other strategies were utilized to obtain the best possible performance with our hybrid MCTS agent.

1. *Rollout Evaluation*: Utility is initialized with the reciprocal of manhattan distance to the nearest food. It is scaled down by a multiplier proportional to the reciprocal of the layout’s perimeter. +1 and -1 are added to the utility for the win and lose states, respectively.
2. *Thrashing Avoidance*: For non-critical zones, the set of actions with the highest utility is prioritized using the alignment of the action with the path to the nearest food, as generated by BFS search. As a result, reflex behavior is observed in huge layouts where MCTS returns similar utilities for all actions. This can be attributed to the fact that most rollouts are timed out for huge layouts.
3. *Stop Action Prevention*: For non-critical zones, it was observed that the agent tried to stop and wait for the ghosts to leave with the expectation that ghosts would randomly move away from the agent. This behavior is similar to the ones observed in conventional search algorithms such as Minimax, which results in severe thrashing until the ghosts corner the agent and kill it. To avoid this scenario, stop actions were removed from the choices of available actions for the agent, and it was never allowed to stop while in pursuit of the food.

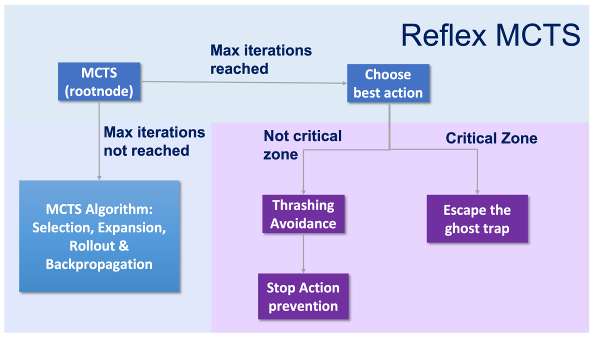


Fig. 1. High-level flowchart of Reflex-MCTS.

Figure 1 depicts a high-level flow chart of our hybrid MCTS implementation. Our agent first seeks the best action using MCTS. It switches to reflex mode for states where multiple actions have similar high utilities. It must be noted that the MCTS algorithm dictates the agent’s choices for reflex actions. Interesting behaviors emerge when Reflex-MCTS assigns higher utilities to actions leading toward power pallets. We observed a peculiar and rather interesting behavior that the agent started exhibiting. Whenever it finds itself in a situation where it has to choose between a path containing power pallets and food on the other, it chooses the path towards the power pallet. Additionally, after eating a power pallet, if it finds a ghost nearby, it eats the ghost, as shown in Figure 2.

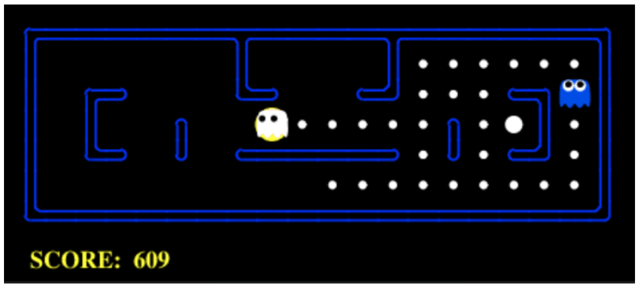


Fig. 2. Picture depicting Pacman eating a ghost

III. Result Analyses

Different agents were compared based on computing times, average scores, and win percentages. We chose ANOVA Tukey HSD for the tests, which involves comparing sample results of different groups by evaluating their mean and variance. Testing layouts were broadly divided into three categories small, medium, and large, each with 30 unique environments. We generate these layouts using an automation script parameterized by size, food density, and ghost count. These variations in layouts ensure a comprehensive performance evaluation of our Reflex-MCTS agent. Compared to conventional agents, Reflex-MCTS was able to perform reasonably well in each category, winning consistently and achieving the highest scores, thus showcasing the efficacy of our implementation. The layout details can be found in Table 1. Parameters that affect the performance of Reflex-MCTS are shown in Table 2.

It should be noted that the vanilla MCTS agent, under real-time time bounds, did not perform well due to its inability to differentiate good and bad actions at critical zones. After we combined MCTS with the reflex properties, our agent could break out of critical zones, resulting in consistent winning streaks.

Properties	Small	Medium	Big
Rows	Between 5 to 10	Between 13 to 20	Between 23 to 35
Columns	Between 1 to 12	Between 8 to 15	Between 10 to 20
#Food	60-80% of the Layout Area		
#Ghost	{1,2,3} with probability (0.19, 0.57, 0.23)	{1,2,3} with probability (0.23, 0.38, 0.38)	{1,2,3,4} with probability (0.11, 0.29, 0.29, 0.29)

Table 1. Layout details for analysis, each column represents a different layout category

Parameter	Description	Value
ucb_param	Exploration constant	2
n_itr	Updates for MCTS	50
max_sim_steps	Rollout timeout	50
ghost_proximity	Run away distance	3

Table 2. Algorithm parameters for the Reflex-MCTS agent

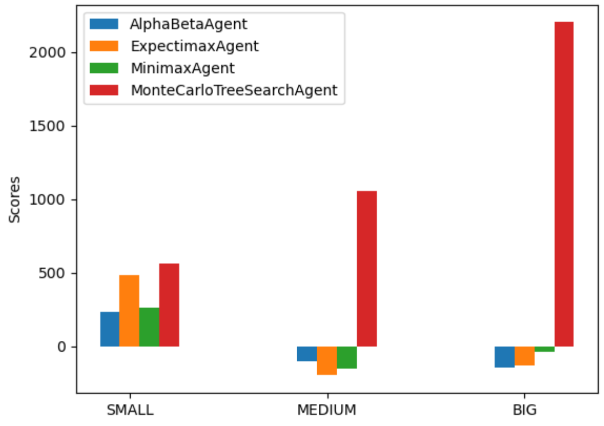


Fig. 3. Comparison of average scores between agents for each layout class

In Figure 3, the results are visibly apparent and in favor of our Reflex-MCTS agent. Our agent outperforms conventional algorithms in all layout categories. Trends from Figure 3 indicate that conventional agents suffer from severe thrashing while our Reflex-MCTS agent successfully avoids such behaviors.

The win percentage result in Figure 4 further confirms the efficacy of our agent, outperforming other agents in all the different categories, especially in medium and big, where it was the only one able to win. Even in the small layout category, where all the agents performed fairly well, Relex-MCTS still had almost twice the win rate compared to any other agents. Expectimax performs better than Minimax as it accounts for stochastic ghosts. Expectimax falls short compared to the Reflex-MCTS agent as it cannot figure out locally optimal strategies that lead to winning situations. Other agents could not win in bigger layouts due to their ineptness in exploring the complete game tree subject to real-time compute bounds. MCTS algorithm efficiently handles this issue using rollouts, which is why Reflex-MCTS performs significantly better in larger layouts.

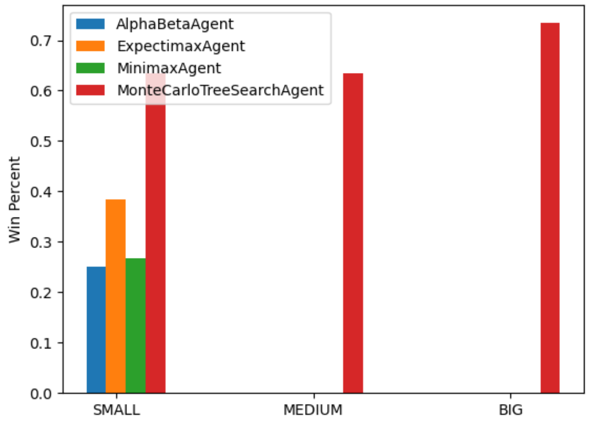


Fig. 4. Comparison of win percentages between agents for each layout category

Figure 5 shows the average compute time for different agents. Decision time depends on two hyperparameters, i.e., the number of iterations for MCTS updates and max number of simulation steps during rollout. Time taken by the Reflex-MCTS agent is often higher than conventional algorithms. The only exception to this trend is in small layouts, where our agent takes the lowest average time out of all the agents. This is because conventional agents usually suffer from thrashing. Decision time could be reduced if parallel computation strategies were utilized or a feature-state approach was used to train a model that could later be applied to other layouts and scenarios. Rollouts can be executed as separate jobs on multiple CPU cores, which can significantly improve the runtime performance of our algorithm.

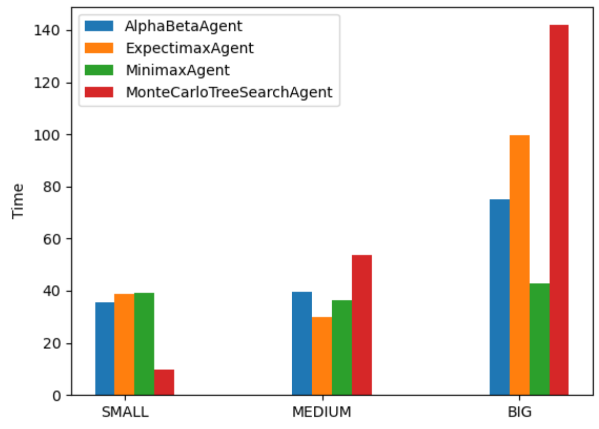
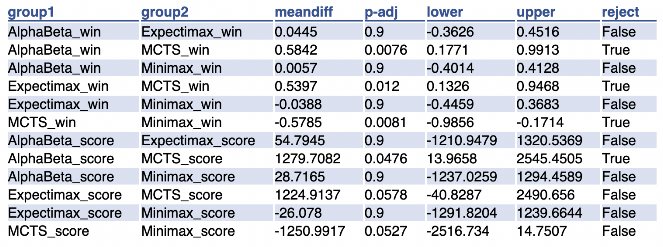


Fig. 5. Comparison of computing time between agents for each layout category

To statistically prove that Reflex-MCTS outperformed conventional agents, we executed Anova and Tukey range testing [3]. ANOVA stands for Analysis of Variance, a test in which two or more groups are compared to prove/disprove the null hypothesis based on a variable. This helps us to determine the distribution among the groups. A simple ANOVA test was not helpful for us as we were looking for the exact group affecting the distribution. Thus, we utilized the Tukey Range Test, which generates pair-wise results.

Figure 6, shown below, presents the Tukey Test results indicating pairwise p-values. Note that for rows having Reflex-MCTS, the p-value is less than 0.05, indicating that the variance in winning percentage and scores is mainly contributed by our agent. These results confirm that our Reflex-MCTS agent is significantly better than other agents.



group1	group2	meandiff	p-adj	lower	upper	reject
AlphaBeta_win	Expectimax_win	0.0445	0.9	-0.3626	0.4516	False
AlphaBeta_win	MCTS_win	0.5842	0.0076	0.1771	0.9913	True
AlphaBeta_win	Minimax_win	0.0057	0.9	-0.4014	0.4128	False
Expectimax_win	MCTS_win	0.5397	0.012	0.1326	0.9468	True
Expectimax_win	Minimax_win	-0.0388	0.9	-0.4459	0.3683	False
MCTS_win	Minimax_win	-0.5785	0.0081	-0.9856	-0.1714	True
AlphaBeta_score	Expectimax_score	54.7945	0.9	-1210.9479	1320.5369	False
AlphaBeta_score	MCTS_score	1279.7082	0.0476	13.9658	2545.4505	True
AlphaBeta_score	Minimax_score	28.7165	0.9	-1237.0259	1294.4589	False
Expectimax_score	MCTS_score	1224.9137	0.0578	-40.8287	2490.656	False
Expectimax_score	Minimax_score	-26.078	0.9	-1291.8204	1239.6644	False
MCTS_score	Minimax_score	-1250.9917	0.0527	-2516.734	14.7507	False

Fig. 6. ANOVA Tukey Range test results

Tukey test results show that the two considerable downsides against Reflex-MCTS are the compute times and the corridor layouts. The latter is likely because our agent still has trouble when trapped in a corridor. The solution for this problem would be to never go into a corridor in the first place, but there are cases where it is either necessary to take the risk or it is too far ahead of time for the agent to foresee it.

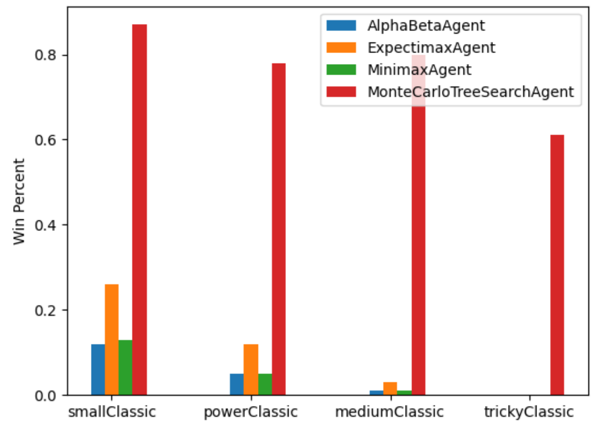


Fig. 7. Win percentages of Reflex-MCTS on standard layouts

Figure 7 shows the performance of our agent on standard Pacman layouts. It can be seen that the results are well aligned with the results of our statistical analysis.

IV. Conclusion and Discussion

This report described a hybrid Monte-Carlo Tree Search (MCTS) agent successfully playing Pacman with random ghosts. Our algorithm utilized reflex decision-making for critical states, thriving in environments where conventional agents often failed. The agent used MCTS for evaluating actions over a short horizon, and intelligent strategies such as ghost busting naturally emerged at runtime. We observed that reflex behaviors helped the agent stay alive in tricky situations. Our comparison results show that our agent performs better in every layout and consistently wins the game. Our Reflex-MCTS agent has a few downsides, which can most likely be resolved with parallel computation strategies and feature-based state generalization methods. In the future, we plan to combine this method with a model-based approach that will treat each state with features for improving real-time performance and pattern recognition abilities. The code repo for this project can be found at https://github.com/prabinrath/Monte-Carlo-Tree-Search-ExMachina

References

[1] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck, “Monte-Carlo Tree Search: A Framework for Game AI,” MICC, Maastricht, The Netherlands, 2008.
[2] Auer, Peter, Nicolo Cesa-Bianchi, and Paul Fischer. "Finite-time analysis of the multiarmed bandit problem." Machine learning 47.2 (2002): 235-256.
[3] Ostertagova, Eva & Ostertag, Oskar. (2013). Methodology and Application of One-way ANOVA. American Journal of Mechanical Engineering. 1. 256-261. 10.12691/ajme-1-7-21.