Monte Carlo Tree Search On Pacman

TEAM 19

Meesum Ali Khan
Arizona State University
Tempe, AZ, USA
makhan35@asu.edu

Siva Nagi Reddy Munaganuru
Arizona State University
Tempe, AZ, USA
smunagan@asu.edu

Satya Shreya Sri Reddy
Arizona State University
Tempe, AZ, USA
sreddy27@asu.edu

Kesava Prakash Thatavarthi
Arizona State University
Tempe, AZ, USA
kthatav1@asu.edu

***Abstract*—In this project, we implement the Monte Carlo Tree Search algorithm for the Pacman Game scenario. We compare the performance of our implementation against the Alpha-Beta Agent and Expectimax Agent we had programmed in Project 2. We use the ANOVA test and consequently the T-test to observe the difference in performance with respect to score and resources utilized by these agents. From our implementation, we observe that the Monte Carlo Tree Search Agent performs much better than the Alpha Beta Agent, and performs at least as well as the Expectimax Agent, while using significantly lesser resources. Therefore, we claim that it is a much faster and more efficient solution.**

I. Introduction

Pacman is an example of an adversarial game, where the pacman agent is trying to eat up all the food and the agents are trying to eat the pacman agent. Depending on the game type, as we'll discuss later, the ghosts may act more randomly, or act in a more adversarial manner, which would increase the difficulty for Pacman.

In our multiagent project, we had developed an Alpha-Beta agent and an Expectimax agent [2] and seen its performance in the Pacman environment. While these are powerful algorithms, they become exponentially resource-hungry as our state space grows, making them inefficient for larger and more complex environments. Also, while we can generate a complete state tree for really small environments, and alpha-beta and Expectimax would perform optimally in these situations, the same cannot be done for larger state spaces. We need to limit the depth at some level, and use some sort of heuristic function to evaluate our states. Thus, our two agents will be completely at the mercy of the heuristic function, which will not be accurate all the time.

This is where Monte Carlo Tree Search comes in. Monte Carlo Tree Search is built on four fundamental steps:

- Selection
- Expansion
- Simulation
- Back-propagation

These steps will be explained in detail in the technical approach section, but they give us a significant resource advantage over the previously discussed methods. Firstly, we do not generate a complete state tree to find the best next move, but rather a smaller one with promising paths that we evaluate through multiple iterations. Secondly, by using methods to balance exploration and exploitation, we are able to cover a large chunk of the promising paths. Thirdly, we are not wasting important space on storing those paths that yield no value. Thus, Monte Carlo Tree Search gives us a more efficient solution for larger state spaces, while performing at least as well as the Expectimax agent.

II. Technical Approach

Monte Carlo Tree Search(MCTS) algorithm requires a given game to be represented as a tree. Where each node represents a game state and the edges represent all the possible moves for a given state. Consequently, the children represent the game state that can be achieved by taking one legal action from a state. Also, MCTS requires each node to carry certain statistical values like the Number of wins, explorations, and scores. These statistical values are evaluated by purely running many simulations and looking at their outcome. The MCTS algorithm consists of four steps which are repeated an arbitrary number of times to improve the result.

A. Selection:

It begins at the root node for which the best move needs to be determined initially. As already stated, the MCTS method constructs the game's statistics tree by iteratively traversing over it. Therefore, it requires a selection mechanism to choose the path that should be evaluated next. The algorithm we used puts more priority on the unexplored nodes over the explored nodes for a given state. Once all the immediate children have been traversed, the upper confidence bound(UCB) applied to trees is used to select the path.

$$A_n = argmax_a(\frac{W}{E} + c\sqrt{\frac{logn}{N_n(a)}})$$

Where W/E stands for No. of Wins and Total Explored

UCB equation tries to find a balance between exploration and exploitation. By varying the C value in the above equation we can change the weight on exploration. For every iteration, we started with C=250 and then reduced the C value by a fixed amount after each iteration thereby shifting more focus on promising nodes. At a given depth d, a node with the highest UCB value is used is selected.

B. Expansion:

The selected node is expanded unless it's a leaf node, and all its children will be appended to the search tree. Then randomly one of the children will be selected for the simulation.

C. Simulation:

The biggest advantage of the MCTS compared to other search algorithms is the Simulation step. Generally, in the simulation step, the selected node is expanded by taking random actions until you reach the game end and then backpropagate the final results. Since Pacman can move in any direction, simulating till the end of the game is not an optimal solution. So, we have put a cap on the simulation depth and used a heuristic function to evaluate a state. To further improve the simulation, we have used the evaluation function of the reflex agent as the rollout policy for the pacman. Accordingly, a ghost agent is developed with a combination of randomized and directional behavior [4].

D. Backpropagation:

The value from the simulation is used to update all the nodes that lead to the selected leaf node. The explored number of every node on the path increases by 1 and the number of wins gets increased by 1 based on the simulation results. These updated values are used UCB function in the next iteration to select a path.

These four steps are repeatedly performed to build the statistical tree. With every iteration, we are exploring a new path and by repeating these steps we learn more about the game, like which path has a higher chance of leading to the winning state. We are also using an early stop parameter to stop the algorithm if the tree does not change much and before completely stopping the algorithm we are first increasing the exploration parameter. After every game step, the MCTS that has been developed so far is saved in a pickle file and we reused the tree again in the next step. This has significantly improved the performance of the algorithm.

The hyper-parameters involved in this approach [1] are the number of iterations, simulation depth, optimism, and heuristic function. The algorithm is very sensitive to the Heuristic function. As the whole statistical values updated in the tree are dependent on the heuristic function, a change in the heuristic function significantly impacted the game result. We have developed a heuristic function that puts more importance on eating food while trying to stay away from the ghosts. The other hyper-parameters are also tuned to improve the performance of the Monte Carlo Tree Search Algorithm. The number of iterations parameter limits the repetition of MCTS algorithm to find one best action. The simulation depth limits the depth in the simulation step.

III. Testing

To test our Monte Carlo Agent against Alpha Beta and Expectimax, we used a number of different environments and metrics. For the metrics, we considered score, wins, memory used, and time taken for one move. For different environments, we took different map sizes and ghost behavior. The three map sizes we considered across all of our tests were smallClassic, mediumClassic, and originalClassic. It is worth noting that small and medium maps only support up to two ghosts, and the original map supports 4 ghosts. For ghost behavior, we varied between random and directional, where random ghosts would make more random moves, whereas directional ghosts would make more moves toward pacman, thus making it harder for him to win.

For the score performance metric, we collected scores across the different environments we've described for all agents, for 10 games each. Then, we used the one-way ANNOVA test for the collected scores to see if there was any agent that was outperforming the others. If we observed that this was indeed the case, we applied the independent T-test between the 3 agents to see which one was yielding the best performance. In the following section, we will outline our results through plots and scores.

IV. Results and Analysis

A. Score Comparison:

For the score comparison, we have taken only the small and medium map, as the Alpha-beta and Expectimax agents are too slow on the original map due to the size. We have compared with both random and directional ghosts, and the results are shown in the figures 1 and 2. In the small map layout, we see that MCTS and Expectimax perform relatively well in the random ghost scenario, but with the directional ghost environment, all agents struggle. This is because it's very hard to escape the directional ghosts in such a small map.

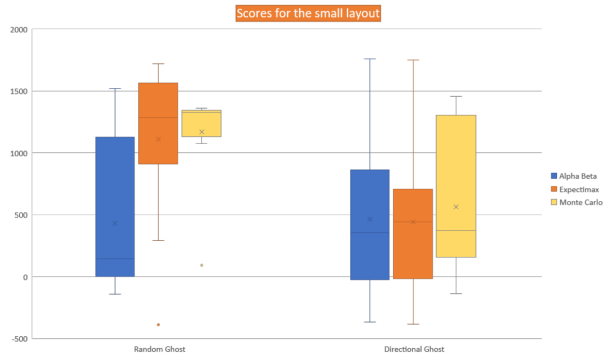

Fig. 1. Small Map Results

In the medium map, we see in the random ghost scenario that our MCTS implementation performs much better. However, with directional ghosts, it performs at the same level as the Expectimax agent.

From the scores and wins, we see that MCTS performs at least as well as the other two agents while doing much better in some layouts. To verify this, we performed the one-way ANOVA test to check whether some agents is outperforming others, and if this was validated, we also performed the individual T-test for that environment to find the best agent.

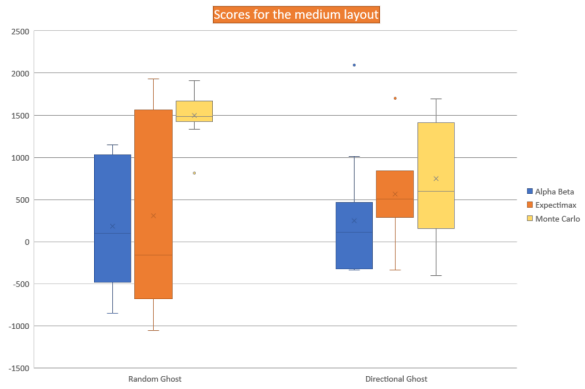

Fig. 2. Medium Map Results

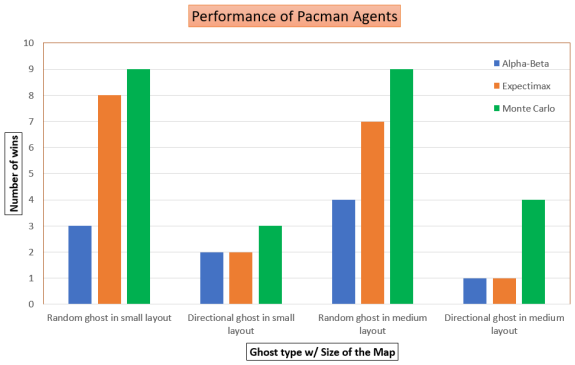

Fig. 3. Performance Comparision of Pacman Agents

Figure 4 shows the P-scores for ANOVA with all agents, and the results of the individual T-test for MCTS with the other two agents can be seen in figure 5.

Configuration	P-Score
Small Map Random Ghost	0.0110653
Small Map Directional Ghost	0.902017
Medium Map Random Ghost	0.0015978
Medium Map Directional Ghost	0.270368

Fig. 4. ANOVA test results

Configuration/Score	Expectimax	AlphaBeta
Small Map Random Ghost	0.809781	0.00412272
Small Map Directional Ghost	0.657954	0.736386
Medium Map Random Ghost	0.00457663	6.42753e-05
Medium Map Directional Ghost	0.523168	0.151981

Fig. 5. T-test results

B. Time and Space Comparison:

As we've seen from the score comparison, our MCTS implementation is performing adequately, and at least as well as the Expectimax agent. However, the main advantage of MCTS is observed in time and space optimization. While Expectimax builds a complete tree up to a certain depth, and Alpha Beta uses pruning to improve on that a little, MCTS builds a much smaller asymmetric tree with the most promising paths, which it iterates over for the given number of iterations. In the following plots, we see how with larger environments, Expectimax blows up exponentially with respect to space and time requirements, and even Alpha Beta with pruning uses much more resources than MCTS. For this test, we have used all three maps and limited the depth of all agent trees to 2. We have measured the time and space taken for one step of each agent and the results are shown in figures 6 and 7.

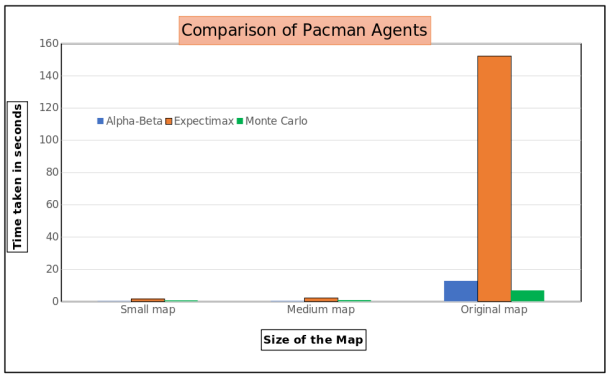

Fig. 6. Time Comparison across all agents

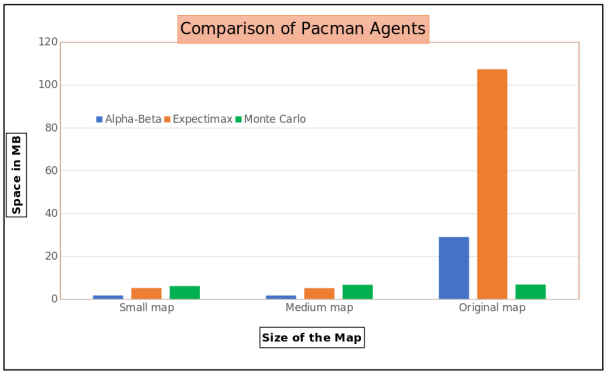

Fig. 7. Space Comparison across all agents

V. Conclusion

In this project, we have implemented a Monte Carlo Tree Search agent for Pacman. In this implementation, we have experimented with different hyperparameters to see what would work best, and have optimized the agent. We have then compared our MCTS agent against the Alpha Beta agent and Expectimax agent we had developed in an earlier course project.

Our results show that our MCTS implementation is comparatively better than the other two agents in terms of score and wins, but the difference is not that high. However, from our

time and space analysis, we can conclude that MCTS is far better than the other two agents. With a growing state space, the other two agents are rendered useless without large time and space resources to utilize. However, MCTS can still work with larger state spaces, and its requirements grow at a much slower rate.

REFERENCES

[1] T. Pepels, M. H. M. Winands and M. Lanctot, "Real-Time Monte Carlo Tree Search in Ms Pac-Man," in IEEE Transactions on Computational Intelligence and AI in Games, vol. 6, no. 3, pp. 245-257, Sept. 2014, doi: 10.1109/TCIAIG.2013.2291577.

[2] M. Sravya, B Tejashwini, V Sanjana, " Expectimax Algorithm to solve Multiplayer Stochastic Game" in Proceedings of the International Journal of Innovations in Engineering Research and Technology, Jan. 2020.

[3] Dienstknecht, M. Enhancing Monte Carlo Tree Search by Using Deep Learning Techniques in Video Games. Diss. PhD thesis. Maastricht University, 2018.

[4] Kien Quang Nguyen and Ruck Thawonmas, "Monte Carlo Tree Search for Collaboration Control of Ghosts" in Proceedings of the IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES,, March. 2013.

[5] https://ai-boson.github.io/mcts/