Project topic 3: Games (Monte Carlo Tree Search)

CSE 571: Artificial Intelligence

Sahith Reddy Gaddam
1218622587
sgaddam7@asu.edu

***Abstract*—Over the past few decades there were a lot of algorithms developed to solve search problems using different evaluation functions. But they have found to be ineffective in cases where the branching factor is huge and decision making is difficult. Monte Carlo Tree Search is one such algorithm which can solve search problems with ease irrespective of the branching factor.**

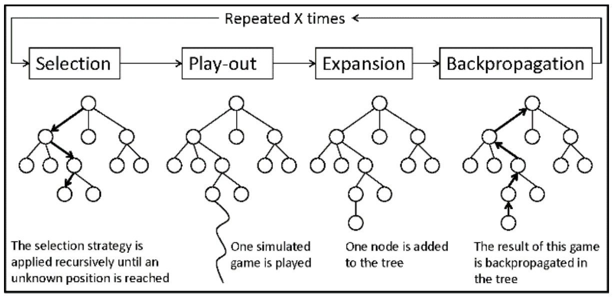
Keywords—component, formatting, style, styling, insert (*key words*)

I. Introduction (*Monte Carlo Tree SEarch*)

Games like tic-tac-toe and rubics cube can be solved by minimax algorithm or AlphaBetaAgents algorithm. The real challenge arises when the agent tries to get to solve the games like Chess or Go. Minimax or Alphabeta algorithms try to explore every branch and node of the game states to get to the accurate output. In certain complex situations when the branching factor is high these algorithms find it difficult to solve the search problems. Monte Carlo Tree Search algorithm is one such algorithm which effectively this problem. MCTS unlike other algorithms is aheuristic which means the MCTS doesn't need any heuristics or the domain information to solve the complex problems. MCTS is faster because of the policy MCTS uses which is to give more importance to some states leaving other states with less importance and allowing those states to be explored most often. In my project I have implemented the pure MCTS algorithm on pacman domain to clear the layouts which the minimax and aplhabeta algorithms couldn't clear. Minimax and alphabeta agents performed well in smaller layouts but in medium and big layouts these algorithms didn't fare as much as MCTS did and in some layouts the minimax and alphabeta agents couldn't win whereas the MCTS agent had a better win ration on average of 10 games.

II. Technical Analysis

Monte Carlo Tree Search is a simple search algorithm which has four parts namely: Selection, Expansion, Simulation and Backpropagation.


Repeated X times
Selection
Play-out
Expansion
Backpropagation
The selection strategy is applied recursively until an unknown position is reached
One simulated game is played
One node is added to the tree
The result of this game is backpropagated in the tree

A. *Selection:*

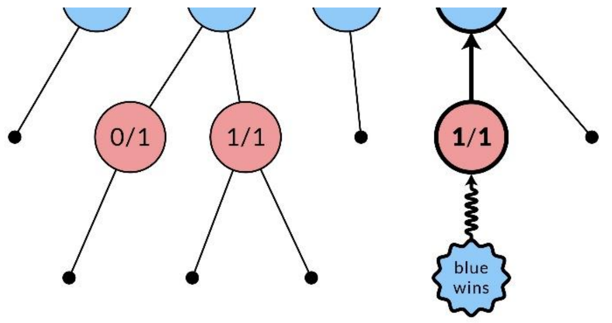
The agent while implementing the MCTS will always want to select a node which most often leads to win state and this known as exploration. This decision is made depending on the statistics stored along with the state information. A good selection balances a way between exploration and exploitation. The agent needs to explore the states which are not visited to get the knowledge of the game domain. This balance between exploration and exploitation can be done by using different Upper cutoff formulas.

B. *Expansion:*

When the game reaches a state, which is visited the first time and it doesn't have any children to explore or exploit then the agent calls the expansion function to expand the game state and get the children nodes of that state. This way the tree is expanded by one set children to a parent state at a time. These children nodes are the future moves that can be be played from the current game state.

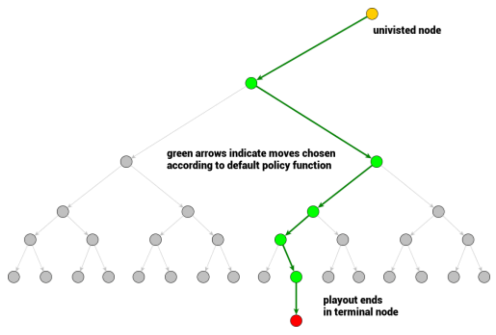
C. *Simulation (or) Playout (or) Rollout :*

After expansion simulation is a recursive approach of selection and expansion till the game is ended. In this function after an unvisited game state is expanded, the simulation plays the game by recursively selecting the expanded nodes and expanding the selected nodes till the game is won or lost. The actions to be selected in simulation is done by giving every action an equal probability. But this leads to a suboptimal solution and takes a lot of time to get to the end state. To overcome this, we might slightly incline to apply a heuristic to get better results.



D. Backpropagation:

After reaching the terminal state the results are backpropagated from the last game state to the first game state. The number of visits to a node are incremented by 1 irrespective of the win or loss. Whereas the number of wins incremented will be depending on the results. These numeric values can be anything or can be domain specific to be a little inclined to solve a particular domain.


univisted node
green arrows indicate moves chosen according to default policy function
playout ends in terminal node

T

III. Technical Approach

As discussed earlier in the analysis, Selection needs to have a upper cutoff formulae. Random selection will explore well but it doesn't exploit the most visited node or the node which guarantee a win. To balance well between exploration and exploitation we add a Upper Confidence bound function as shown in the below figure.

$$\frac{w_i}{s_i} + c\sqrt{\frac{\ln s_p}{s_i}}$$

Here in the formula:

W_i stands for the number of wins the agent got by simulating through that state.

S_i represents the total number of visits of the state in simulation process

S_p represents the total number of simulations of the present states parent node

The interesting part here is chosing the constant parameter c. I have chose the value of c to be sqrt(2) to perfectly balance the exploration and exploitation. The above UCB formula can be interpreted in a way that as the number of visits to a particular node increases the denominators of the function s_i increases significantly and dominates the numerator thereby decreasing the UCB value of the node. Hence, if a node is more often visited, then the UCB value of that node decreases and the agent will select other node which justifies the exploration part. As long as the number of visits of a node are low and number of wins achieved through that node are high, the agent will exploit that node. At the start of the game there are cases when the initial values of all the children nodes are equal and even the UCB couldn't determine the node to be selected. In such cases choosing a random function would be inadequate because the agent might end up oscillating between two or three states for a little amount of time and increasing its number of visits. Having a random function as tie-breaker for UCB also doesn't help the agent to reach the goal. Random function explores well but cannot exploit. But having a heuristic as tie-breaker will aid the exploitation and doesn't explore well. To balance exploration and exploitation in case of a tie-breaker of UCB I have used a probability function to choose a random function 40 percent of the time and choose a heuristic value 60 percent of the time to determine the children node. This probability distribution was chosen after numerous hit and trail methods which was kind of balancing the exploration and exploitation.

IV. Results

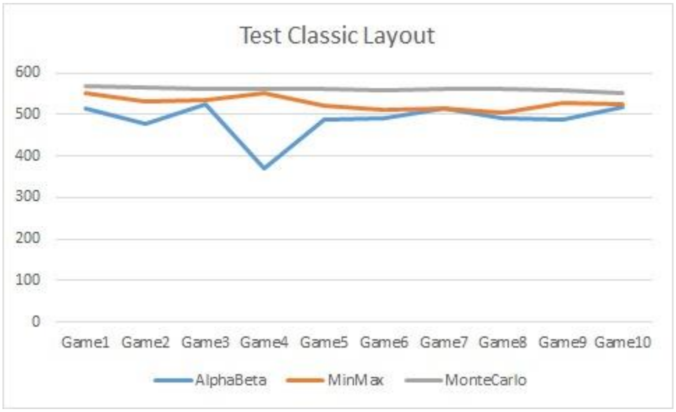


Figure 1. Comparisions on testCLassic layout

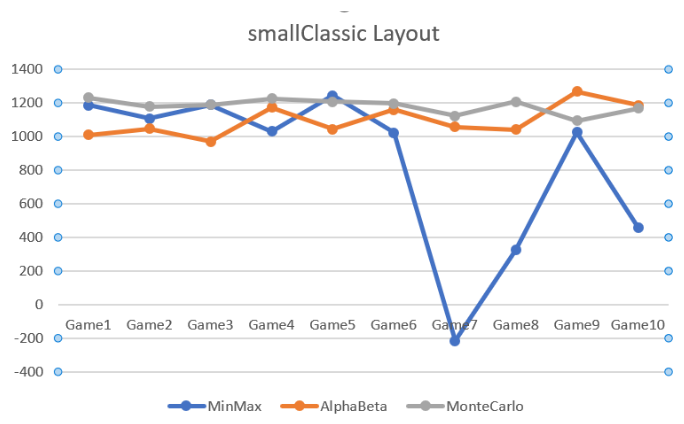


Figure 2. Comparision on smallClassic

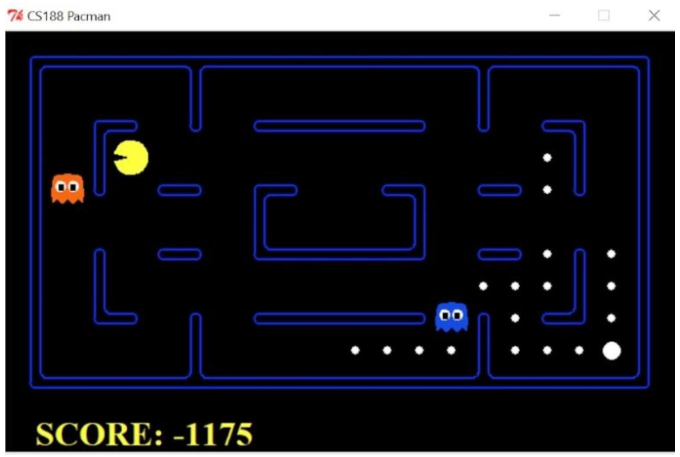


Figure 3. Minimax in mediumCLassic

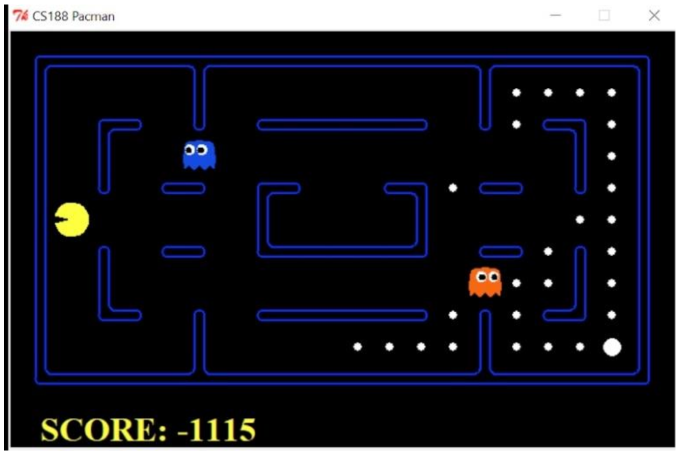
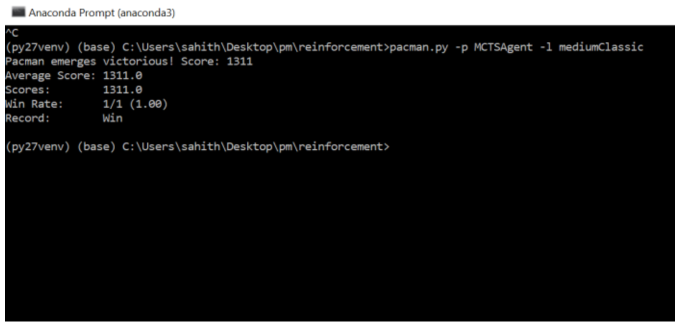


Figure 4. AplhabetaAgent on mediumClassic



Anaconda Prompt (anaconda3)

```
^C
(py27venv) (base) C:\Users\sahith\Desktop\pm\reinforcement>pacman.py -p MCTSAgent -l mediumClassic
Pacman emerges victorious! Score: 1311
Average Score: 1311.0
Scores:        1311.0
Win Rate:      1/1 (1.00)
Record:        Win

(py27venv) (base) C:\Users\sahith\Desktop\pm\reinforcement>
```

Figure 5. Result of MCTSAgent on mediumClassic

V. CONCLUSION

As minimax agent and alphabeta agent tries to solve for every node of the game state, it becomes difficult for the agent to find a solution the when layouts starts getting bigger. As we can see that minimax and alphabeta are almost comparable to monte carlo tree search algorithm when implemented in test classic which is a small layout. Minimax, alphabeta and monte carlo tree search agent wins every game of the 10 games played. But the scores are relatively less in case of minimax and alphabeta because those algorithms are implemented on an assumption of the ghost being rational but the ghost isn't rational in our real game played. Whereas, the monte carlo algorithm doesn't assume the agent to be rational. Minimax and alphabeta agent tries to survive being in a safe state instead of being killed. This is because those algorithms doesn't explore well and losses some of the states which can get to the food pellets closer. Monte carlo search algorithm exploits and explore almost every state and chooses the best and fastest path to win the game. Comparisions start to get interesting when we start to increase the size of the layout. When we implement the small classic, the minimax and alphabeta agent almost everytime is stuck in a place executing stop action everytime if it cannot find a food pellet to motivate it to move as it is a depth limited search. At some cases when the ghost nears the agent, it starts to move and if the ghost follows the agent and makes the agent go till a next food pellet then the agent will start hunting for the food pellets as it is now inside its depth limit. As long as the ghost doesn't come near the agent the agent executes stop function and the score get lower and lower. On the other hand the monte carlo tree search has the flexibility to simulate till the game ends and can search every node. So it does a lot better on smallClassic layout. Now on the mediumClassic the minimax and the alphabeta agent doesn't win the game at all. Instead they are stuck forever executing the stop action until the ghosts follow them. But mediumClassic being a relatively larger layout and ghosts having a random behaviour, it is unlikely that the ghost can follow the agent till the agent can get to the remaining food pellets on the other side of the layout. One interesting observation in mediumClassic from the monte carlo algorithm is that the pacman after visiting the power pellet state doesn't get afraid of the ghost because of the layout being large enough to explore ample number of different possibilities and finding out that after visiting power pellet state, visiting the ghost state can kill it (can fetch the agent more rewards). But to let the pacman know that the power pellet only has a limited time ability takes a lot of simulations.

VI. Contribution

1. Sahith Reddy Gaddam (100%):
   Constructing Algorithm, building code base, accumulate results, organize report.

   Signature:

   [illegible]

Acknowledgement

I would like to thank my Professor Yu Zhang for his valuable inputs and insights provided during the lectures and the doubt solving sessions which lead to this project being at the stage that it is.

References

[1] Monte-Carlo Tree Search: A New Framework for Game AI Guillaume Chaslot, Sander Bakkes, Istvan Szita and Pieter Spronck∗ Universiteit Maastricht / MICC *(references)*

[2] Real-Time Monte Carlo Tree Search in Ms Pac-Man **Published in:** IEEE Transactions on Computational Intelligence and AI in Games (Volume: 6, Issue: 3, Sept. 2014)

[3] A Survey of Monte Carlo Tree Search Methods **Published in:** IEEE Transactions on Computational Intelligence and AI in Games (Volume: 4, Issue: 1, March 2012)