Lab 4: Multi-Agent Search

Mirza Mohammad Azwad (Student ID: 200042121)

CSE 4618: Artificial Intelligence
BScEngg in Software Engineering(SWE), Department Computer Science and Engineering, Islamic University of Technology(IUT)

14th April, 2024

I Contents

II Introduction

The primary focus of this lab is on Multi-Agent search using the classic PacMan game, in simpler terms simulating a turn-based gameplay involving different agents which in this case would be **PacMan** and the **Ghosts**. The lab tasks revolve around designing evaluation functions and simulating gameplay using **MiniMax** and **ExpectiMax** for the agents' behavior. If the agent plays optimally we consider a **MiniMax**-based gameplay with the behavior being **deterministic**. On the other hand, if the opponent, i.e. the Ghost has some **stochasticity** associated with it, then we go for **ExpectiMax**. Also for Minimax gameplays, we consider another important factor about optimizing the traversal of the game tree with consideration of pruning certain branches via the use of **Alpha-Beta** pruning.

Unlike the previous lab, we shift our focus back the the Pacman environment of the earlier labs. Additionally, we are using our cse4618 conda environment which is a Python 3.6 environment required to run the provided code.

III Analysis of Problems

1 Reflex Agent

In this task, we were asked to design an evaluation function considering the state-action pairs. The evaluation function, in this case, takes the current state as well as the next possible state and returns a score that would give a measure of the goodness of that particular state. We were provided with some dummy code which provided:

- **successorGameState**: the new game configuration upon performing the action
- **newPos**: Agent's new position upon performing the action
- **newFood**: Remaining food packets upon performing the action
- **newGhostStates**: Ghost States upon performing the action
- **newScaredTimes**: The scared time if any for the ghost states obtained via **newGhostStates**.

Considering the obtained state space configurations and the next possible configuration upon performing the action, we needed to deduce an evaluation function. Simply put, an evaluation function is a numerical representation of the desirability of a particular state within the problem-solving domain and in the context of our problem, maximizing this value is what we look for. If the successor however is a win state just return a large number as this state is almost always favorable. If the successor is a lose state just return the score default score, further evaluation won't be helpful.

2 Minimax

In this problem, we are asked to create a minimax algorithm for Pacman. As per the aforementioned details, minimax essentially assumes optimal play for both agents. Interestingly, Pacman commits suicide when it can't find a possible win state to sort of keep the score as high as possible. Here Pacman is the **maximizer**, whose goal is to maximize the score with the Ghosts as the **minimizer**, whose goal is to minimize the score. Using this notion we can generate a **minimax tree**, with each level representing either **PacMan** move or **Ghost** move. Upon traversing this tree, from the root to the leaves, selecting the maximum value for maximizers and the minimum value for minimizers we can get a gameplay where both agents play optimally. Since Pacman makes the first move, the first level would be a maximizer or the root node would be a maximizer.

3 Alpha-Beta Pruning

When it comes to minimax trees, they often have less important branches, which when explored become computationally expensive. We can still obtain the same result at the root node without having to explore these less important subtrees. The idea is quite simply to check what the parent node wants, if the parent node is a maximizer, and the children are minimizers if at any point in that level, a minimizer obtains a node less than the current maximum, it avoids exploring that particular node any further, thus pruning the tree and reducing the unnecessary exploration. This method of pruning is called **alpha-beta pruning**.

4 Expectimax

In the case of **expectimax**, we remove that optimal gameplay idea. Essentially while one of the agents would play optimally and would be a maximizer node, the other agent or the opponent would be stochastic in nature, where it has some probability associated with taking a particular action. Now for **expectimax**, the "*expecti*" comes from the fact that these stochastic nodes, or the opponents/levels having stochastic nodes, generally compute the **expectation** of all its outgoing branches. This expectation is maximized by the maximizer agent.

5 Better Evaluation Function

One of the primary concerns regarding adversarial search is that sometimes, the tree is too big to actually traverse completely. This in turn makes it essential for the value at certain nodes to be sort of estimated and this is where the evaluation function steps in as the better the estimation, the better the results. Having a **good evaluation function** reduces the number of branches we have to explore, thus also making the algorithm comparatively less computationally expensive.

In this task, we were asked to design a good enough evaluation function considering only the state unlike the **Task I** which considered both the state and the action.

IV Explanation of Solutions

1 Reflex Agent

In this task, the evaluation function relies primarily on the provided values mentioned in the analysis section. The first thing I considered is that the default score would be obtained from **sucessorState** using *getScore*. We obtain the distance to the food packets by considering the positions of the food packets obtained using the *asList* method and the current position of Pacman by computing the Manhattan distance, we store the obtained distances as a list called *distanceToFood*. Similarly, we also compute the distance to the ghosts using the newGhostStates by iterating over them and using the *getPosition* method to get the ghost position and accordingly compute the Manhattan distances between the current positions and the ghost positions, storing the distances as a list called *distanceToGhost*. This default score would be modified by subtracting the minimum distance to the food packet(assuming there are food packets, we can check this using the **len** method to see the length of the *distanceToFood* list) as subtracting the minimum distance would imply the least decrease and a higher value of the resultant score such that it would move towards the minimum distance or the closest food packet. Additionally, to stop Pacman from being stuck to the same position over multiple moves, I deduct 10 from it every time the action is *Directions.STOP*. Also based on the total scared times, if the scared times are less than or equal to 0, we add the minimum distance to the ghost to the score as this ensures that if the closest ghost is nearby, it gives more priority towards moving away from the ghost as maximizing the evaluation function is the goal. But if the sum of the scared time is greater, then we deduct the minimum distance to the ghost(assuming there are ghosts, we do a similar check to that of food packets) as that would help move towards the ghost similar to the logic of the food packet, and as long as the ghost is scared, they can be eaten by pacman.

2 Minimax

To design the **minimax** algorithm, we make use of a **recursive helper method** which we assigned the identifier **value**. This method takes the *gameState*, *agentIndex*, and the *depth* as its parameters, and to access the attributes of the object from which it is being called, it also takes *self* as a parameter. Then based on the agentIndex if it's zero calling the maximizer, call the minimizer. The helper function keeps calling either of these functions as long as it is not a win condition: *isWin*, lose condition: *isLose* or it ran out of depth: *depth==0*. The depth helps determine, how many moves ahead it should observe, and win and lose conditions represent a possible win configuration and a possible lose configuration respectively. We discuss the maximizer agent but note that the minimizer agent is just symmetric or can be considered antagonistic to the maximizer agent with similar underlying logic. The minimizer and maximizer have the same arguments as value. Within the method, it initializes variables **retScore** and **retAction** to large negative values and **None** respectively, to store the best score and the resulting action found during the search. By calculating **nextAgent**, it determines whose turn will be next after the current agent, Pacman or ghost. The depth only decreases if the last agent makes the move or in other words, when an agent with agentIndex 0 arrives, indicating a full circle. The legal moves are then retrieved for the current agent and while iterating through them, each move helps generate a successor game state and evaluates its value using our recursive helper method which would call the minimizer. Using the helper function it can switch between minimizer and maximizer and based on this switching logic it can essentially create a minimax recursion tree. Upon getting the value from our helper method while backtracking, it checks if the retScore is lesser than the value obtained, if yes we have obtained a larger retScore and thus we store the corresponding action as the goal of the maximizer to maximize the utility. We eventually return the score and the action which can be used to move toward the most optimal gameplay.

3 Alpha-Beta Pruning

Alpha-Beta pruning evolves from the **minimax** implementation. As discussed in the analysis section, this pruning helps eliminate less important branches to not traverse them making the algorithm more computationally efficient. When it comes to the implementation, we modify our minimax approach by passing 2 additional arguments the alpha and beta. The alpha is initialized as a very small negative value while the beta is initialized as a very large positive value. For the minimizer, while traversing we check if the retScore obtained is smaller than alpha, and in this case, we just simply return the retScore and action without traversing any further. Doing this means that the alpha value already points to the maximizer's desired outcome, so traversing any further would not be required as the maximizer would pick its desired value anyway. A similar symmetric or antagonistic approach can be considered for the maximizer, where we compare the beta values, and if the retScore is greater than beta means that beta already found the minimizer's desired outcome so traversing that particular branch would be unnecessary. In the maximizer however, we also calculate the beta values using a simple **min** function to obtain the minimum value of beta among its children during each iteration, this beta is passed back to the helper value which would call the minimizer with this beta value. A similar thing is also observed for the minimizer. Thus, we avoid traversing unnecessary branches.

4 Expectimax

In the case of **expectimax**, the implementation logic is modified from the **minimax** implementation. The change lies in the minimizer, here instead of a minimizer we have a stochastic agent, so what this means is that we have to somehow compute the expectations. Now, for the sake of this problem, we assumed the probability of each move to be uniformly distributed so we compute the probability(p) as the reciprocal of the number of legal moves. Previously in the minimizer we compared with the retScore but here we just evaluate the retScore but not the resultant action as all of the actions are equiprobable.

In the case of **exp** function, unlike the minimizer, we assign retScore a value of 0 initially, and for each move, we update the value of retScore as retScore summed with the product of p and the value obtained from our helper function. Upon completing the iteration, we return the retScore as well as the retAction to preserve a similar structure to our original minimax implementation.

5 Better Evaluation Function

The implementation is similar to our first evaluation function. But unlike the first function, here we consider only the state and not the action. We initially define the following:

- **currentPos**: PacMan's current position
- **foodPos**: The position of the food packets
- **currentGhostStates**: The states of the ghosts as per the current configuration
- **currentScaredTimes**: The scared time of the ghosts if any obtained from **currentGhostStates**

We use the score for the currentState as the default score. Upon obtaining the score for the currentState we add the reciprocal of the sum of all the Manhattan distances to Food Packets, assuming there are food packets, otherwise, we don't add anything. This reciprocal minimum is when there are lots of food packets and the maximum is when the number of food packets dwindles. Decreasing the number of food packets by gobbling them is one of our goals. Additionally, we have to deal with the ghosts, we want to maximize our distance from the ghost. But when the ghost is scared, we want to minimize the distance as we have the option to gobble them up. Similar to the first task, we obtained our *distanceToGhost* and *distanceToFood*. We also perform similar checks for the *distanceToFood* and the *distanceToGhost* to check if there are food packets available and ghosts available respectively. First, we check if there are ghosts present. Assuming there are we check if the total scared time is positive, this would mean that the ghosts are currently scared. If the ghosts are scared we want to be closer to the ghost to eat them, so we deduct the sum distance to the ghost and the number of pellets from the score. But we also prefer having larger total scared times, so we add the total scared times to the score. Again if we had a total scared time of 0, means that the ghosts aren't scared, in this case, we can add the sum distance as our target is to be as far away from the ghost as possible as the ghosts could potentially devour Pacman. Upon computing the score in the aforementioned manner we can return the score as the result of the better evaluation function.

V Problems Faced

I struggled mostly with the first and the last task as they were about designing evaluation functions. When it comes to finding numerical representations initially, it didn't make sense, but with some tweaking, however, it started to become more comprehensible. Initially, I found it hard to wrap my head around the concept of evaluation functions needing to give higher values for more favorable conditions due to conceptual shortcomings but eventually, I understood how the maximizers or minimizers work and utilize the evaluation functions in their goals of maximizing the expected utility and minimizing the expected utility respectively.

Additionally, I couldn't figure out how to find the ghost positions and this took a considerable amount of time to figure out that ghost positions could be obtained by iterating over the *newGhostState* using *getPosition* to obtain individual positions of the ghosts.

Furthermore, I also struggled with figuring out that the *newFood* was more than just a 2D array having True(T) and False(F). Although it was used in one of the prior labs I probably forgot and this led to me having to refigure this out and then utilize the *asList* method to obtain the (x,y) positions of the food packets as a list of tuples. The tuples are the (x,y) pairs and the list contains all the tuples representing each of the food packets.

Lastly, the minimax algorithm was implemented in class and similarly, we also implemented expectimax and alpha-beta pruning in class. As these were discussed in class, I did not face significant difficulties regarding them.

VI Interesting Findings

The first of the interesting findings would be the UNIX time command which can time the execution of a Python script, before this course I don't recall ever using it and didn't know something like this existed.

Regarding the evaluation functions upon further inspection, the runtime is as follows:

Task	Time Taken	Score
Task 1(T1)	4.80	1131.3
T1 with Capsule Information(T1CI)	4.92	1131.3
Task 5(T5)	12.91	1108.4
T5 with Capsule Information(T5CI)	13.55	1108.4

All the times taken here are the average times over 10 different runs to ensure better precision. To take the capsule information I used the *getCapsules* method which revealed the position of the capsules. But what I observed is that the position of the capsules remains insignificant which contradicts my original idea that the position of the capsules and the number of capsules might help form a better evaluation function. Considering the scared times alone is enough and taking the capsules only increases the runtime but does not improve the results, as the score remains unchanged.

VII References

- **Code**: GitHub-azwadmirza
- **Overleaf**: View Template Overleaf
- **Time to Execute a Script**: Time Command Linux