Introduction

This lab is based on the idea of **informed search** and **heuristics**. One of the primary tasks was to design an admissible and consistent heuristic to carry out an informed search using **A* search** and **suboptimal search** such that the pac man can consume all the food packets while considering the walls in place. The presence of a ghost was not considered in this lab and would most likely be considered in future labs.

Some prerequisites for this lab was the idea of **heuristics**. One of the best ways to think about heuristics in my opinion is to consider children playing hide and seek. Typically in a game of hide and seek we have multiple children trying to hide, while one of them is tasked with the goal of finding them. This child initially closes his/her eyes and then tries to search for his/her friends. Instead of searching the whole area of the game. The child can simply look for potential places where the other children could hide. This is simply the intuition behind the use of heuristics. Reducing the overhead of searching every possible node by the use of some intuition or estimation.

One key factor about heuristics is their **admissibility** and **consistency**. **Admissibility** can be thought of as how sensible a heuristic is, the fact that it won't most likely lead someone or AI agent in this context in the wrong direction. Whereas **consistency** is how reliable the heuristic is, the value won't change randomly. In the context of hide and seek like above, an admissible heuristic would be looking for their friend's behind the window curtain as compared to looking for them inside a mug. **Consistency** on the other hand is making sure their estimation leads them to search at the possible hiding spots, in contrast to searching for their friends at random places, say once you look behind the curtain and another time you look inside a mug, these two estimations are far apart in their possibilities of being a logical hiding spot.

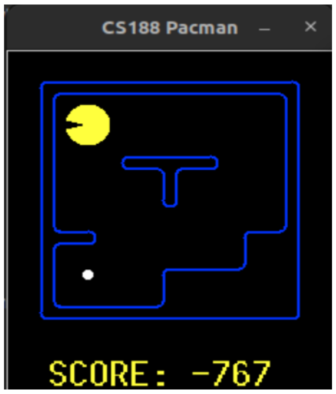
Lastly a simple difference between **trivial** and **non trivial** heuristics is essentially the difference in terms of complexity and sophistication. A **trivial** heuristic is simple and does not provide much guidance whereas a **non-trivial** heuristic is a bit more sophisticated and thus provides better guidance in moving towards the goal.

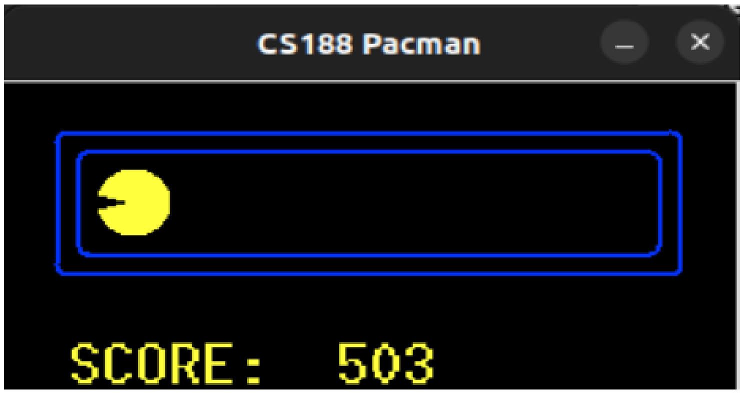
In this lab, we used our previous **conda** environment to work in the required **python 3.6** environment as required by the given files.

Analysis of the Problems

1. The **first problem** involves simulating the python pacman game downloaded as per the lab task. Upon running the mentioned commands in our task book we can see two different simulations. One in a **test maze** where the pac man can only move left and right but it moves west towards the food pellet which coincidentally is also present in the same direction and consumes it and the other in the **tiny maze** where the pacman can go in all directions. It makes use of something called a GoWestAgent which implies that the action taken by the pacman involves going in a specific direction which is the west direction in this case. The image on the left is the **tinyMaze** but since the pac man never reaches the food pellet the game never ends and the score keeps on becoming more negative which is essentially how the score changes for every move. Basically, the greater the

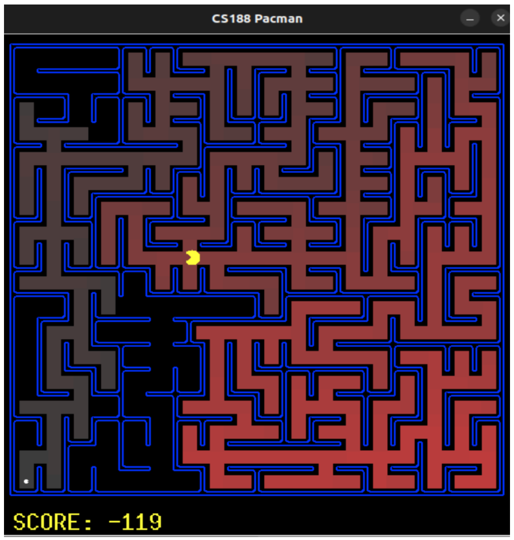
number of moves the more negative the score gets, a food pellet being consumed adds a large plus sign which can be seen when the pacman consumed the food pellet in the **testMaze**.



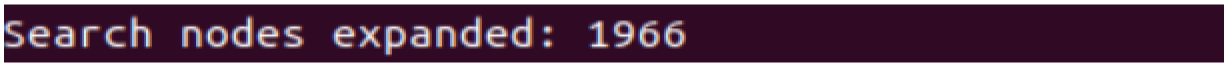


We are also shown that there is an option to see the help option to call other relevant operations for simulation

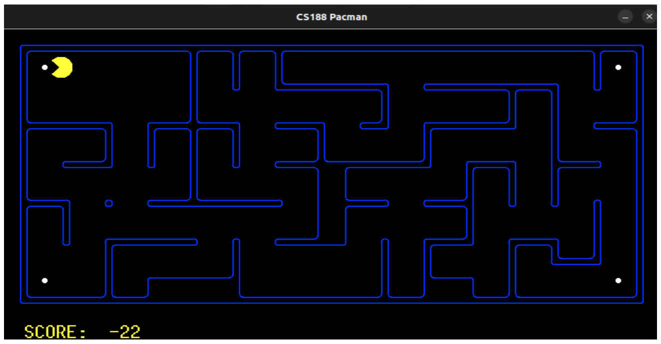
2. The **second problem** is where we implement **A* search**. Similar to the previous lab, we implemented the problem in the *search.py* file and upon implementing the file we can view the simulation results upon running the A* search algorithm for our pac man. We can do so using the commands mentioned in the task book. And upon running the simulation we can observe **pac man** moving across a large maze to reach the food pellet at a corner of the maze. For this particular simulation it made use of the Manhattan Distance Heuristic, which is essentially the sum of the absolute difference between the ordinates and the absolute difference between the abscissa.



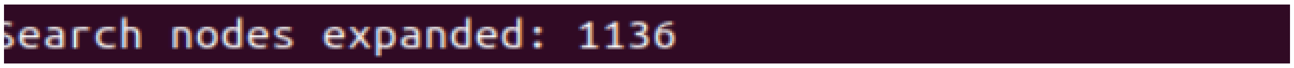
3. In the **third question**, we were asked to apply the A* search for the corners problem such that if pac man reaches all four corners, it has won the game. The solution to this problem was implemented in class. We needed to formulate a problem where the pac man has to visit possible corners of the maze and consume the food pellets placed there. Upon consuming all the food pellets in the corners, it wins the game.



```
Search nodes expanded: 1966
```

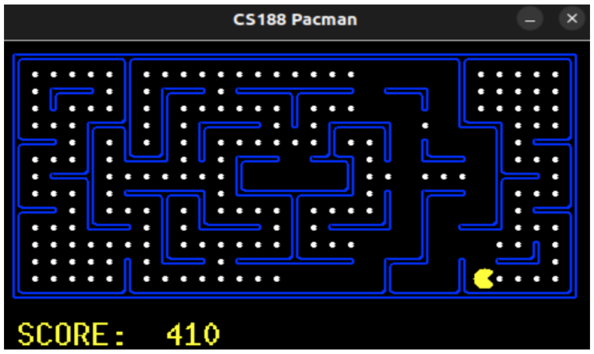


4. In this **fourth problem,** we were tasked with implementing a heuristic to go to these corners. Namely creating a heuristic that helps take pac man to the respective corners. We had to keep in mind that the heuristic had to be **admissible** and **consistent.** We can observe that upon applying the heuristic the number of nodes expanded to reach the solution drops from 1966 to 1136.



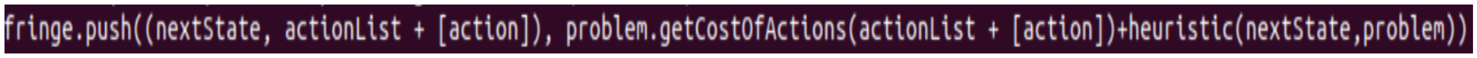
```
Search nodes expanded: 1136
```

5. In the **fifth problem**, we were tasked with implementing a **food search problem**. This problem was also implemented by our instructor but the main idea was to implement a solution or a path that pac man could follow to consume all the pellets. But our task was to implement a heuristic or a food search heuristic that allows pac man to follow the ideal path with the algorithm that produces the ideal path having to expand the least possible number of nodes. **Heuristics** in this case also had to be **admissible** and consistent and the grading policy reflects that the lesser the number of nodes expanded the more grade points one receives. We observe that the tricky search has a long start time which means that the algorithm has to expand a lot of nodes to essentially find the optimal path.
6. In the **final task**, we had to implement **Suboptimal Search**, or **greedy search** where pac man greedily eats the nearest pellet or dot. In this problem we also had to define a **goal test** to see if the position we are in is a possible goal state or not, in other words whether or not it contains a food pellet. And additionally we also had to write an algorithm to find the path to the closest dot or pellet.



Explanation of Solution

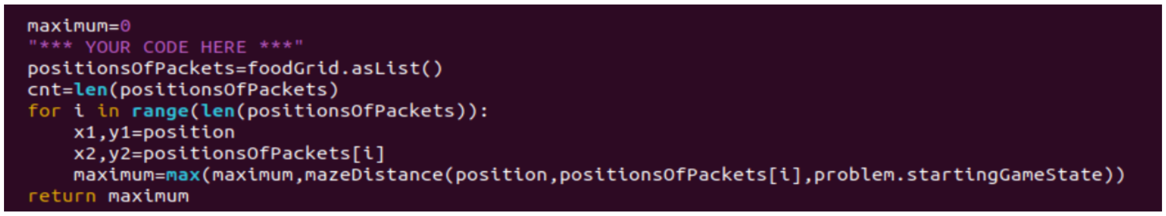
1. The first task was self-explanatory and was discussed in our analysis section. In the **second task** we had to implement the **A* search** algorithm in the *search.py* file. The other problems were developed in the *searchAgents.py* file. For the A* search we made an implementation similar to our UCS implementation using a priority queue as a fringe to get the lowest cost at the top of the priority queue regardless of how we enqueue the states. This also allows us to get the best path with an O(logN) complexity. The A* search in addition to computing the best path while considering the cost also takes into account the heuristic. This heuristic allows fewer expansion of nodes to reach the solution. We only had to make a slight modification to our UCS code to have the A* search algorithm working that involved storing a sum of the heuristic and the actual cost as part of our cost in our priority queue or min heap.



```
fringe.push((nextState, actionList + [action]), problem.getCostOfActions(actionList + [action])+heuristic(nextState,problem))
```

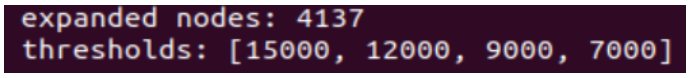
2. The **third task** essentially stores the corner information extracted from the attributes of the SearchProblem object and we define a new corners tuple as an attribute along with a visitedCorners dictionary to check for whether or not the corners were visited. In the **fourth task** we had to implement a corners heuristic that helps pacman find the corners faster. For this problem using the manhattan distance as the heuristic was sufficient to solve the problem within the mentioned constraints.

3. In the **fifth task,** we had to implement a **foodHeuristic** which was essentially using one of the implemented methods called mazeDistance that gets the actual cost by running **breadth first search** for the given problem with a defined start state and a goal state. It goes from the start state to the goal state and computes the actual cost. The actual cost is the best heuristic as a heuristic that closer to the actual cost of traversal a particular path tends to give better results. In this case we did not consider the paths to incur different costs or basically the paths had the same weight but assuming it did not we could have used **uniform cost search** in place of breadth first search. Upon using the mazeDistance method the number of nodes expanded became lesser than 7000 which helped earn the extra credit as the number of nodes expanded was precisely 4137 using mazeDistance, and the same was received using mazeDistance modified with ucs which lead me to conclude that the weights of the edges for the resulting graph is 1.. The heuristic I defined was as follows:



```
maximum=0
"*** YOUR CODE HERE ***"
positionsOfPackets=foodGrid.asList()
cnt=len(positionsOfPackets)
for i in range(len(positionsOfPackets)):
    x1,y1=position
    x2,y2=positionsOfPackets[i]
    maximum=max(maximum,mazeDistance(position,positionsOfPackets[i],problem.startingGameState))
return maximum
```

Where for every state the maximum distance amongst all the reachable food pellets is considered as a heuristic for that state, this is admissible as it is considers the farthest packet from a particular location and upon getting the top element of the priority queue the minimum weight received would point to the optimal path which tries to get to the packet with the lowest sum of the actual cost and the heuristic value, and this sum would always point to the closest element due to the actual cost being present in the sum.



```
expanded nodes: 4137
thresholds: [15000, 12000, 9000, 7000]
```

4. In the **final task**, we define the goal state by checking if it's a food pellet using the attribute food which is defined in the constructor for **AnyFoodSearchProblem**. To find the path to the closest dot or pellet, we can simply run a bfs of our problem using the **breadthFirstSearch** algorithm defined in the last lab's *search.py* file. Using the bfs we can find the optimal path considering that the weight of the edges are 1. UCS can also be used here and it also passes the test cases set in the *autograder.py*. The ClosestDotSearch agent does not always find the optimal path, as by running bfs alone, it does get the path but to the next best goal, or the closest dot. But moving towards the closest dot can lead to other overheads such as encountering a wall with no ways to move in either direction except the reverse of the direction pac man came through, this might lead to the same path being traversed multiple times which may lead to a less optimal path being traversed.

Problems Faced and Interesting Findings

In this lab I primarily struggled with finding the right heuristic for the foodSearch problem and I could not find the right heuristic during the lab time. My approach was to use the maximum Manhattan distance instead of the maximum of the mazeDistance. I also tried to use the euclidean distance, which would be a more optimistic heuristic but it still didn't pass. I realized later that the number of expanded nodes was much greater in comparison to the **mazeDistance** method.

Heuristic	**Number Of Nodes Expanded**	**Runtime(Including q1)**
Maze Distance(BFS)	4137	19s
Maze Distance(UCS)	4137	10s
Manhattan Distance	9551	5s
Euclidean Distance	10352	5s

One thing we observe here is that while Manhattan and Euclidean have lesser runtime, computationally less expensive. It still leads to the expansion of a far greater number of nodes. This is a clear indicator that a heuristic that is closer to the actual cost which is in the case of using the mazeDistance method leads to a lesser number of nodes expanded. But since **mazeDistance** makes use of bfs or ucs(modified) under the hood, it is computationally more expensive as it tries to calculate the actual cost of going from a position to a food pellet or dot.