

Simon Chaisouang

[shchaiso@uark.edu](mailto:shchaiso@uark.edu)

010878218

CSCE 5613 Artificial Intelligence

Spring 2025

Late Days: 0

## 1 - Introduction

This report builds on the search algorithms that were implemented in our last report for a Pacman game in Python. For this report, we are given five different tasks that involve leading Pacman to collect all of his food. The search agent problems that will be implemented are all contained in the searchAgent.py file. The five tasks that are implemented and expanded upon in this report are listed as follows:

* Finding All Corners
* Corners Heuristic
* Eating All Foods
* Suboptimal Search

In conjunction with the task at hand, the following constraints are applied:

* The wall cannot be hit more than 2 times.
* The wall must be hit at least 1 time.

## 2 – Implementation of Search Problems

**Finding All Corners**

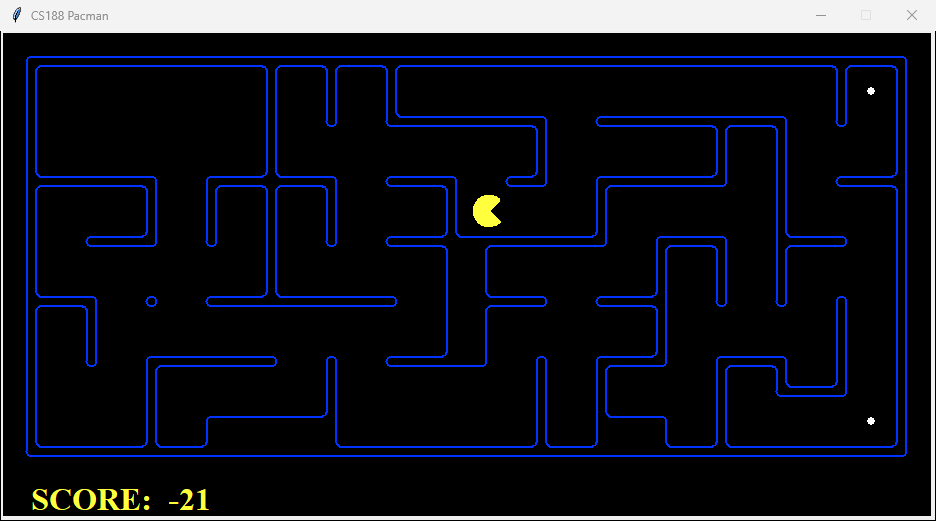
For this task, we implement four instance methods of a search problem defined as CornersProblem(). The problem is setup by initializing the starting information for a given game state such as: walls, a starting position, four corners that contain food, the cost of each action (1 per move), and a node expansion counter to keep track of how many nodes the search algorithm has expanded across.

The first instance implemented is the getStartState() method. The method returns that initial starting state of Pacman defined as (self.startingPosition, ()). The method reads in a state that is represented as a tuple where the first elements represent Pacman’s current position, and the second element represents a collection of corners that have been visited. The second element is an empty list since Pacman starts at its initial position with no corners visited.

isWall() is the second instance implemented. The method determines if a given state’s position corresponds to a wall. It extracts the (x, y) coordinates from the state and checks the corresponding value in the walls grid. If the grid at that (x, y) coordinate is marked as a wall, the method returns true; otherwise, it returns false.

The isGoal() method is the third instance implemented. The method checks if the current state meets the goal criteria, thus all four corners have been visited. A state’s second element is read in, visited corners, and checks if the length of the element is 4 meaning all 4 corners have been visited. If the element contains a length of 4 then it returns true, otherwise it returns false.

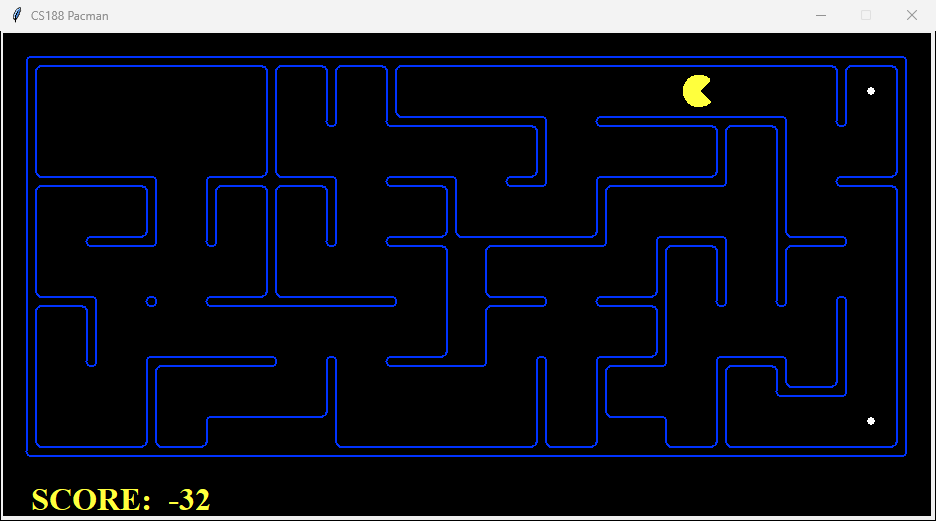
The final instance implemented is the getSuccessors() method. This method provides the next possible states for a given game state based on acceptable moves. The current state and set of visited corners are obtained from the read in state. For each possible direction (north, east, west, and south), the method loops over all possible actions. New positions are calculated from converting the action to a directional vector (dx, dy). The new positions are then verified by checking if they lie within the grid boundaries denoted by M and N. Visited corners are then updated, and if the new position is identified as a corner, but has not been visited, and not in the list, it is added to the visited corners list. A new state is then constructed as a tuple of the new position and the updated visited corners. The new state along with the action taken and a cost of 1 is added to the successors list. Lastly, the \_expanded counter is incremented by 1 and the successors list is returned.



**Figure 1.1 Sample Output of CornersProblem over Medium Maze**

**Corners Heuristic**

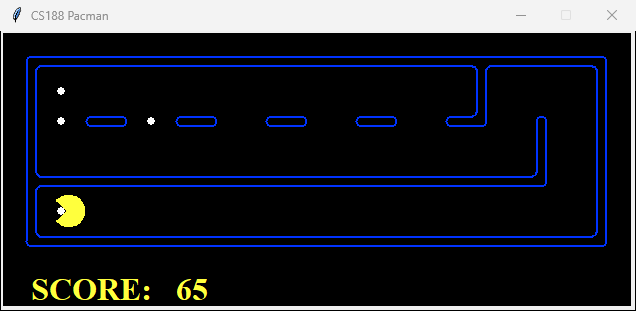
For the next task, we are tasked with implementing a cornersHeuristic() function that returns the estimated cost from the current state to the final state when using the A\* searching algorithm. The function starts by reading in a state and a problem. The stats consist of two elements; Pacman’s current position (x, y), and a list of corners that have been visited. The problem contains information about the maze, locations of all four corners, and the maze walls. The function then computes a list of corners that have not been visited yet by filtering the complete list of corners. If all corners have been visited, then it returns a value of 0 since there is no remaining cost. For each unvisited corner, the Manhattan distance from the current position is calculated. The heuristic lastly returns the maximum value among those distances.



**Figure 1.2 Sample Output of cornersHeuristic Implementation**

**Eating All Food**

For the third task; given FoodSearchProblem() where it finds a list of actions that leads Pacman to collect all his food, we are required implement a foodHeuristic() function that returns a lower bound on the actual shortest path cost to the final state. The function starts by reading in a state that contains the elements of Pacman’s current position (x, y) and a grid that contains a list of food coordinates. The heuristic is initialized as a list with [0] meaning no food is present within the list. For each food position in foodList, the Manhattan distance is calculated from Pacman’s current position to the food’s location. The distances are then appended to the heuristic list. The function then returns the value of the largest Manhattan distance from Pacman’s position to the reaming food positions. No matter what path Pacman takes, he must at least travel that maximum distance to reach the farthest food, making the value achieved a valid lower bound.

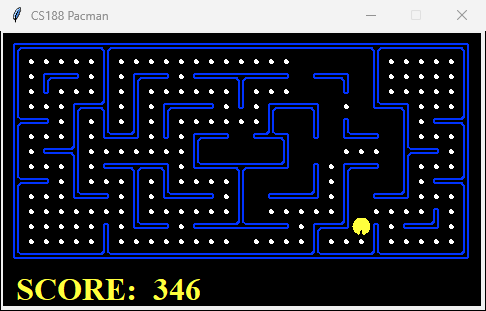


**Figure 1.3 Sample Output of foodHeuristic Implementation**

**Suboptimal Search**

There are some cases where, even with the A\* search and a good heuristic, finding the path for Pacman to collect all his food can be difficult. In such cases, it is better to find a reasonable solution rather than an optimal one. For this final task we implement a greedy algorithm that prioritizes food closest to Pacman. The findPathToClosestDot() function is implemented to lead Pacman to the closest food given a current game state.

The function retrieves the current game that contains necessary elements such as walls and search problems. The problem is instantiates as AnyFoodSearchProblem() using the current game state so the problem is set up where any state containing food is the goal. The Breadth-First Search (BFS) algorithm is called to find the shortest path to any food in a given maze. The results are then unpacked into actions and hits. numHits is updated by adding hits to track the number of times Pacman has hit the wall, and the list of moves that lead to the nearest food is returned as actions.



**Figure 1.4 Sample Output of findPathClosestDot() Imlamentation**

## 3 - Results

The following tables showcase the performance outcomes of our previous implementations:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Maze Type** | **Cost** | **Nodes Expanded** | **Score** | **Record** |
| Tiny Maze | 18 | 1245 | 1245 | Win |
| Medium Maze | 77 | 7767 | 463 | Win |

**Table 1.1 Results of CornerProblem() Implementation**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Implementation** | **Cost** | **Nodes Expannded** | **Score** | **Record** |
| cornersHeuristic | 77 | 3322 | 463 | Win |
| foodHeuristic | 34 | 23449 | 596 | Win |
| findPathToClosestDot | 348 | N/A | 2362 | Win |

**Table 1.2 Results of Task 2-4 Implementations**