Berkeley Pacman Search



Jordan James

CS 455

Embry-Riddle Aeronautical University

24, April 2019

## Introduction

This paper will analyze the results of the project given at http://ai.berkeley.edu/search.html. The main goal of this project is to implement various search algorithms to perform various tasks implemented in a Pacman maze. These tasks include finding fixed food dots using uninformed search algorithms such as breadth first search and depth first search, using a heuristic to visit each of the four corners in the maze in the least amount of moves, and eating all of the dots in as few moves as possible. This project is built using Python 2, with many python files being provided. The provided files build a shell of the Pacman game, and provide test mazes for implementing and testing the algorithms. The project also comes an auto-grader file, to check the correctness of the implemented solutions.

The first steps in this project are focused around implementing uninformed search algorithms. The first algorithm that needs to be implemented is a depth first search algorithm. Depth first search follows a very basic principle. Starting from the root node, this algorithm immediately expands each new successor it receives. This can lead to solutions being found quickly, but may not be least cost solutions. This algorithm is implemented in the Pacman maze using the stack data structure. Breadth first search is similar to depth first search but differs in that states are discovered in the order they are received. This can lead to many nodes being visited, but often finds the least cost solution. To implement this in the Pacman maze, a queue is used.

After implementing uninformed searches, the next task is to implement informed searches. The two desired searches to implement are a search using a cost function, and the A\* search algorithm. The cost function search is similar to breadth first search, except it takes into account the cost of the path. For this project, two cost functions are already implemented. One of these rewards paths that have food with a lower cost, while the other penalizes dangerous paths that have ghosts in the area. This can be easily implemented using a priority queue. A\* uses heuristics to determine which paths to take. Two heuristics, Manhattan Distance and Euclidean Distance, are provided for use in this project. Very similarly to the uniform cost search, the A\* search algorithm can be completed using a priority queue and just adding in a heuristic to analyze.

The third set of tasks is to implement two functions to traverse to each of the four corners of the maze. The first function is meant to do this using any of the basic search algorithms, while the second is meant to utilize a heuristic to find the shortest possible path to traverse all four corners. The key missing component of the first function is expanding the tree with all successor nodes. Once completed, the function just needs to keep track of how many corners have been visited. The second function can be completed easily through the use of the implemented Manhattan Distance heuristic.

The final set of tasks for this project is implementing two functions for eating all food dots in the maze. Implementing the first function is simple as food dots are given in a list, so the only necessary implementation is using the A\* heuristic implemented earlier to find food. The second function is similarly implemented where it always tracks the closest food location.

**Contents**

Introduction ii

Approach 1

Problem 1 1

Problem 2 2

Problem 3 2

Problem 4 3

Problem 5 4

Problem 6 5

Problem 7 6

Problem 8 7

Results 7

Problem 1 8

Problem 2 9

Problem 3 10

Problem 4 12

Problem 5 13

Problem 6 13

Problem 7 14

Problem 8 15

Conclusion 15

Appendix A: Lessons Learned 16

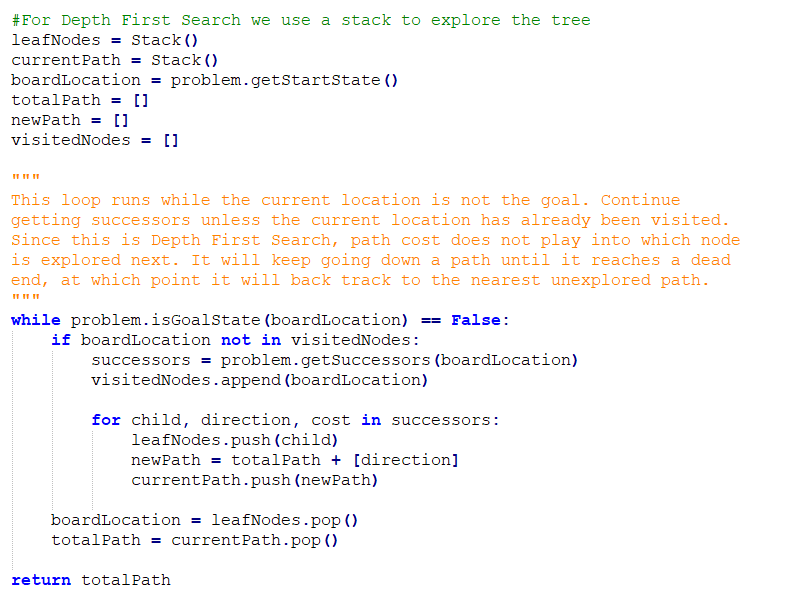
References 17

## Approach

Many of the functions that need to be implemented in this project can be done using methods already provided in the project. The files “searchAgent.py” and “util.py” both contained useful information used in the implementation of the search algorithms. The “searchAgent.py” file contained information on getting successor nodes as well as useable heuristics for the latter problems. The “util.py” file provided information on the stack, queue, and priority queue data structures used in solving the first few problems.

Problem 1

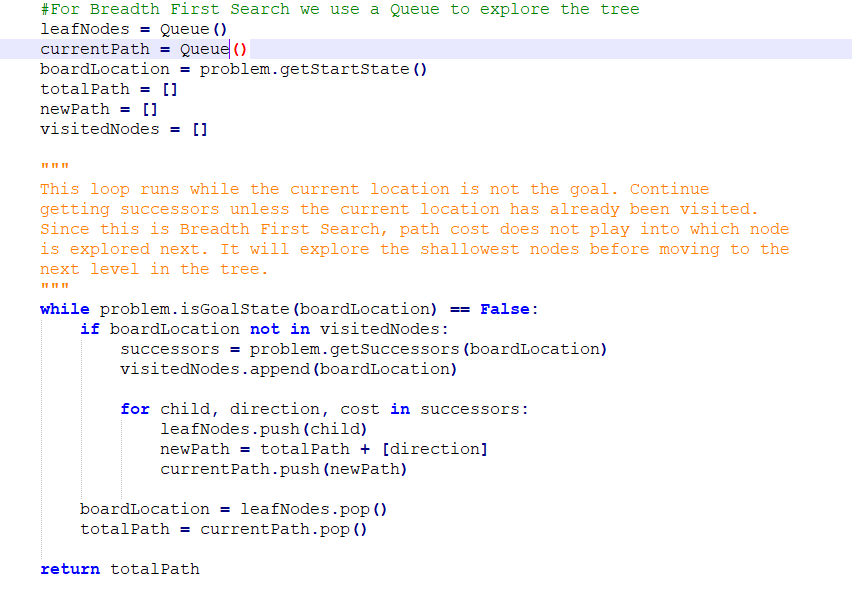
The goal of this problem is to implement the depth first search algorithm in the *depthFirstSearch* function in “search.py” to help guide Pacman to a food dot located on the map. To implement this, a loop is used to track if the goal state has been met. Inside of the loop, successor squares are pushed into a stack. Locations that have already been visited are kept track of to avoid repeat visits. The final path returned is the first solution path found by the algorithm.



**Figure 1.** Code for DFS algorithm.

Problem 2

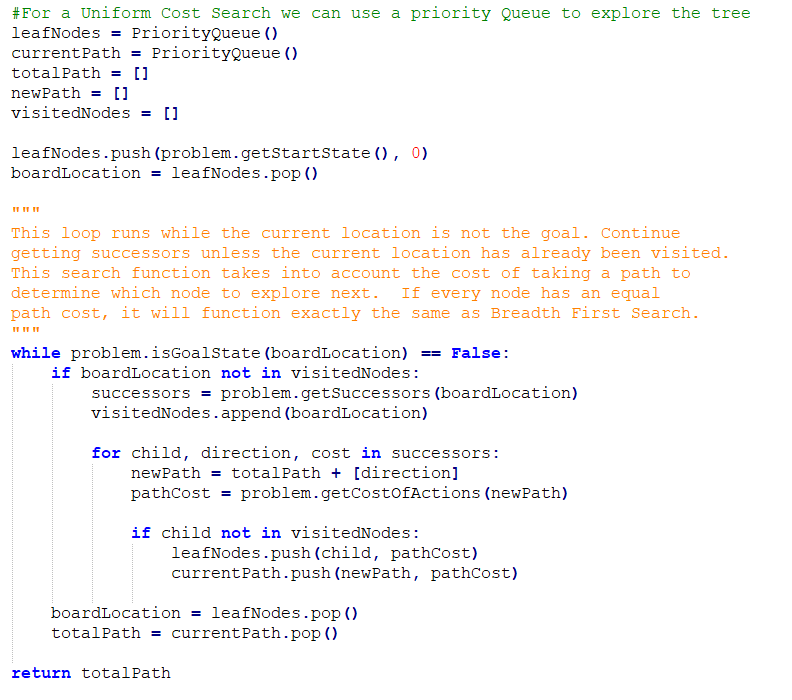
Problem two asks to implement the breadth first search algorithm in the *breadthFirstSearch* function in “search.py” to guide Pacman to a fixed food dot. The solution to this problem is essentially the exact same as the solution to Problem 1 above. The only difference is that instead of using a stack, a queue is used. The queue allows each child node to be evaluated before moving on to the next level in the tree.



**Figure 2.** Code for BFS algorithm.

Problem 3

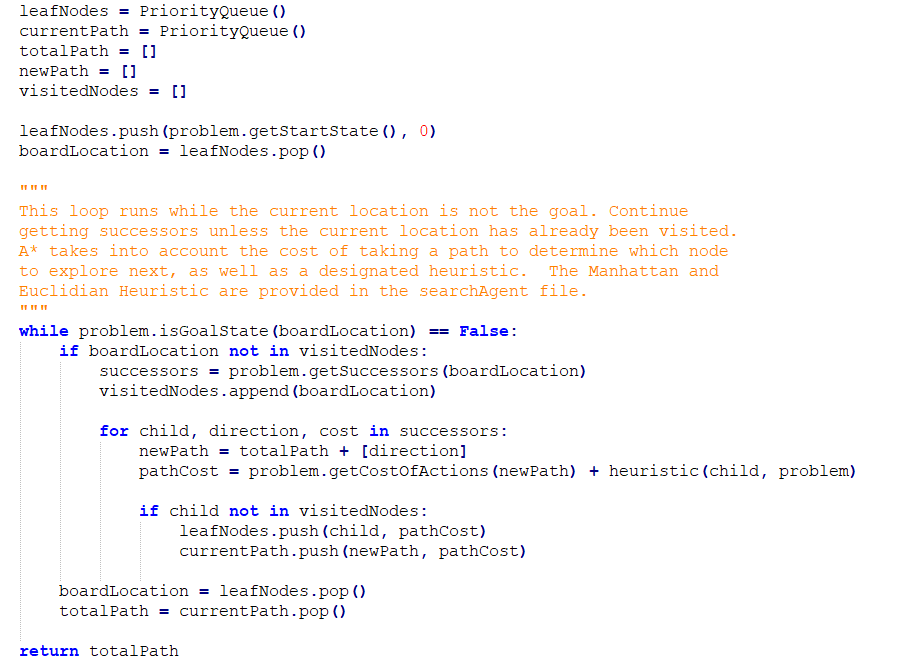
Problem three requires an implementation of the uniform cost search algorithm in the *uniformCostSearch* function in “search.py”. This function is designed to use two cost functions implemented in the *stayEastSearchAgent* and *stayWestSearchAgent* methods in “searchAgent.py” to choose the path with the lowest cost. The solution for this problem is similar to the previous two, but uses a priority queue as its data structure. This allows the cost function to be included into the successor decision.



**Figure 3.** Code for uniform cost first search algorithm.

Problem 4

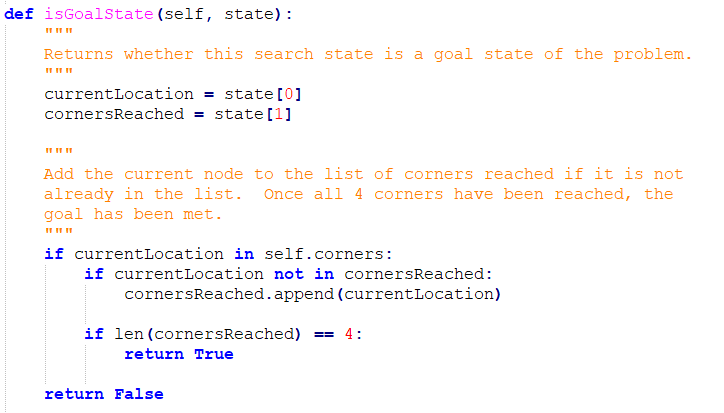
Problem four asks for an implementation of the A\* search algorithm in the *aStarSearch* function in the “search.py” file. This algorithm uses a heuristic to determine which successors to evaluate. Two heuristics, Manhattan Distance and Euclidean Distance, are given for use in the “searchAgent.py” file. The implementation for this method is similar to the implementation for the uniform cost search method, but for this one a heuristic is added in.



**Figure 4.** Code for A\* search algorithm.

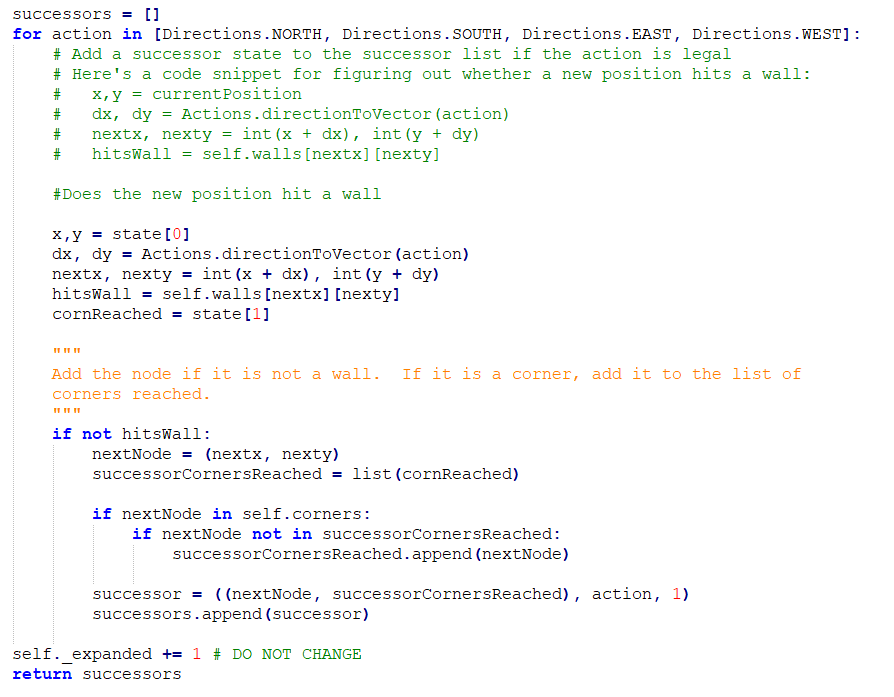
Problem 5

Problem five builds off of problem two in order to find all four corners in the maze. This problem requires editing of multiple functions in the *CornersProblem* class in “searchAgents.py”. The two functions requiring change are *isGoalState* and *getSuccessors*. The *IsGoalState* method can be completed by a simple check if all four corners have been reached.



**Figure 5.** Code for *isGoalState* function in the *CornersProblem* class.

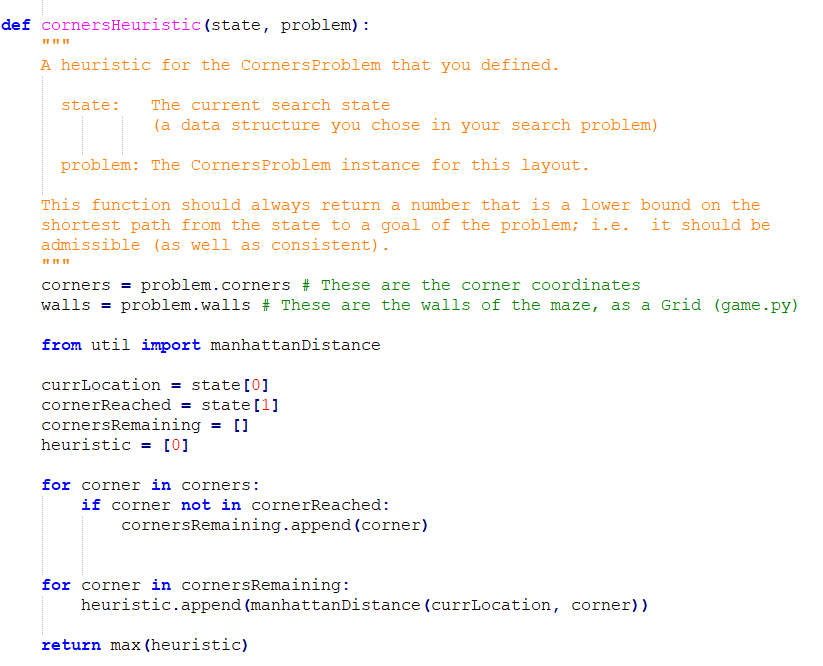
The first part of *getSuccessors* must be checking whether the new successor is a wall. If so, that node must be discarded since it is not a legal move. Next, the function must check if the node is a corner of the map.



**Figure 6.** Code for *getSuccessors* function in *CornersProblem* class.

Problem 6

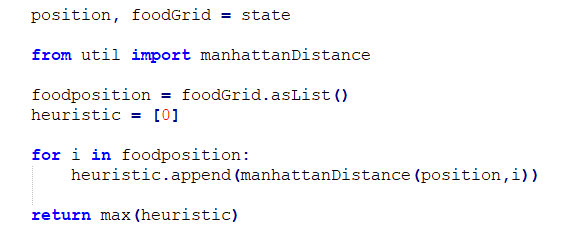
Problem six is similar to problem five, but requires a heuristic to be used to reach all four corners in the fewest possible moves. The heuristic used for this problem is the Manhattan Distance heuristic.



**Figure 7.** Code for *cornersHeuristic* function.

Problem 7

The task for problem seven is to implement the *foodHeuristic* function in “searchAgents.py” to eat all food dots on the map. The heuristic used to solve this is again the Manhattan Distance heuristic.



**Figure 8.**  Code for the *foodHeuristic* function.

Problem 8

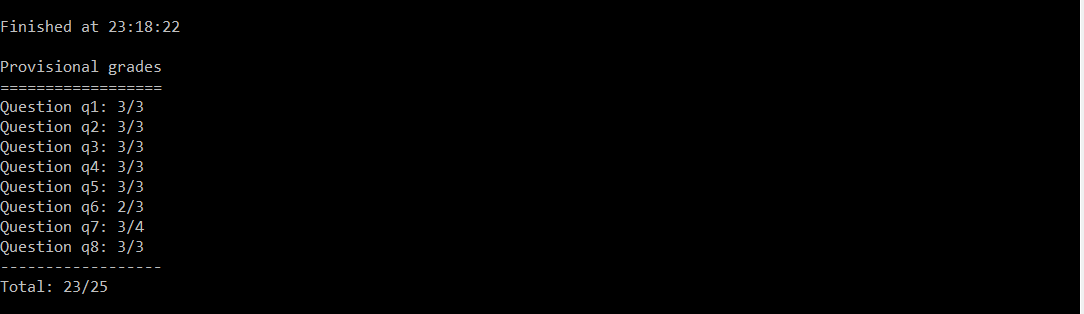
Problem eight asks to implement the *findPathToClosestDot* function in “searchAgents.py”. Using the given *AnyFoodSearchProblem* class, the closest food dot can be found by simply using A\* search.

**Table 1.** Files needed to run the project.

|  |  |
| --- | --- |
| **Edited Files** | |
| search.py | This file stores all of the search algorithms |
| searchAgent.py | This file stores all of the search-based agents |
| **Other Files** | |
| pacman.py | The main file that runs the Pacman games |
| game.py | Logic behind the Pacman game |
| util.py | Data structures for implementing search algorithms |
| graphicsDisplay.py | Graphics for Pacman |
| graphicsUtils.py | Support for Pacman graphics |
| textDisplay.py | ASCII graphics for Pacman |
| ghostAgents.py | Agents to control ghosts |
| keyboardAgents.py | Keyboard interfaces to control Pacman |
| layout.py | Reads layout files and stores their content |
| autograder.py | Project auto grader |
| testParser.py | Parses auto grader test and solution files |
| testClasses.py | General auto grading test classes |
| searchTestClasses.py | Pacman Search auto grading test classes |
| test\_cases/ | Directory containing test cases |

## Results

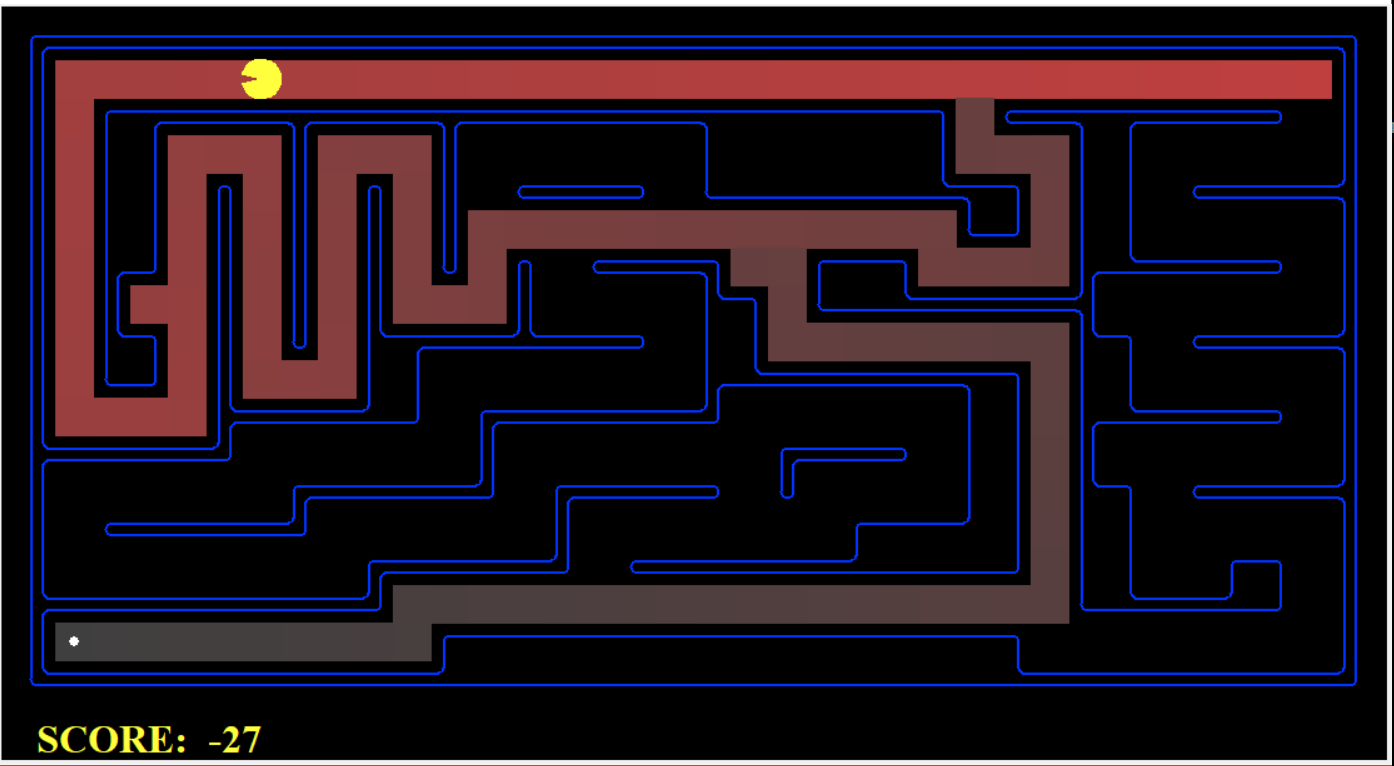
Overall, this project can be considered a success. The program runs and successfully solves each of the eight listed problems. Using the provided auto grader provided in “autograder.py”, the code receives a score of 23/25. Problem six and Problem seven each received one point off because a solution that expands fewer nodes could be implemented.

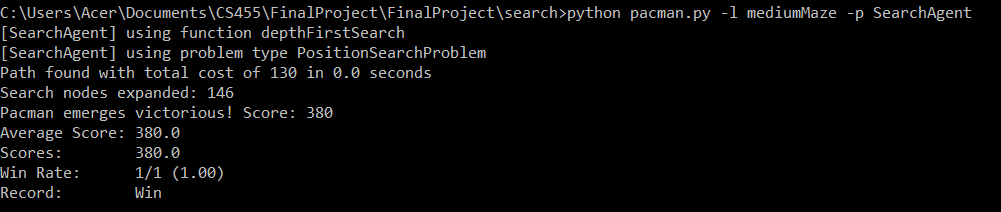


Problem 1

Depth first search resulted in a very high solution path cost. This is one of the major disadvantages of depth first search. This algorithm can find a solution faster due to traversing deeper in the tree earlier on, but often times the first solution found will not be a least cost solution.

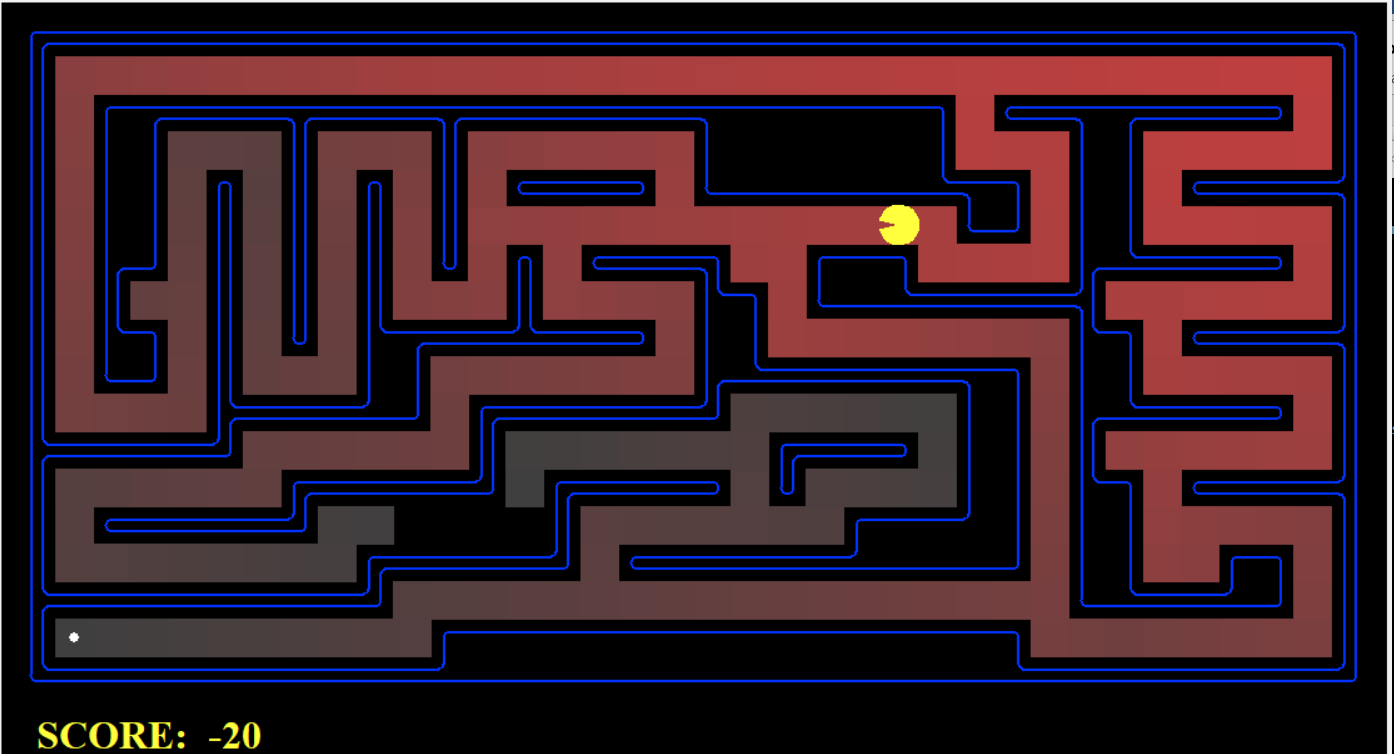
The graphic below shows a visual representation of the paths explored by Pacman. The colored paths represent all nodes explored in the tree. The brighter the color, the earlier it was explored. When the program is run, Pacman traverses the final solution path. As you can see, Pacman did not take the most efficient path to the food dot using this search algorithm. The expected result for this example is a final path cost of 130, so this problem is a success.

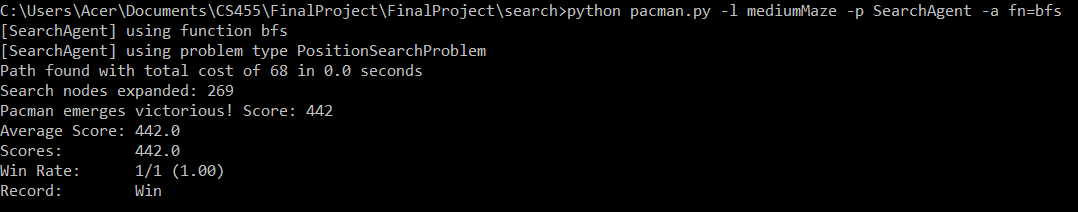




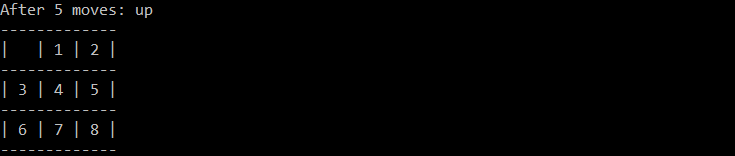
Problem 2

Breadth first search expanded many more nodes than depth first search, but was rewarded with a lower cost solution. As you can see in the visual, most of the path is colored, indicating that the node was explored in some path (may not be the solution path). For small problems such as this, breadth first search will almost always outperform depth first search, because time is not a factor. Both search algorithms found the solution before 1 second passed. For larger problems where thousands of nodes need to be explored, breadth first search can become extremely time consuming. This is when depth first search can possibly have an advantage.



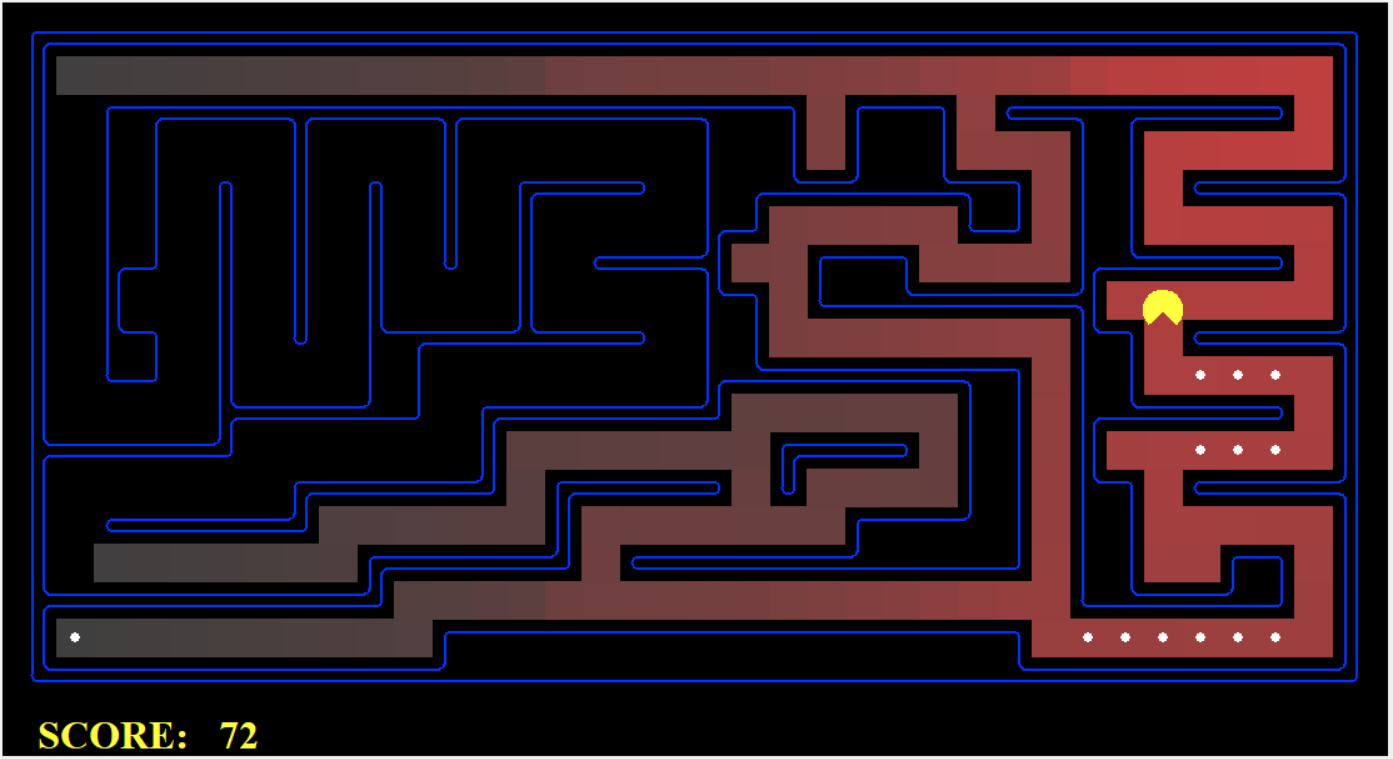


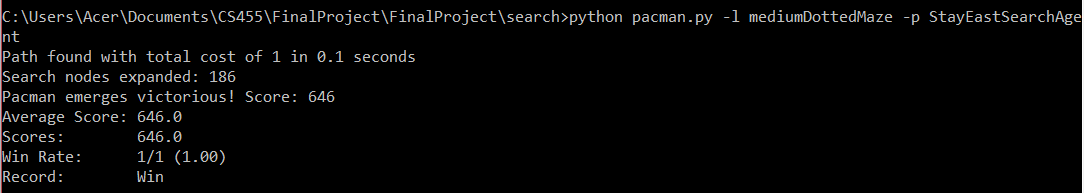


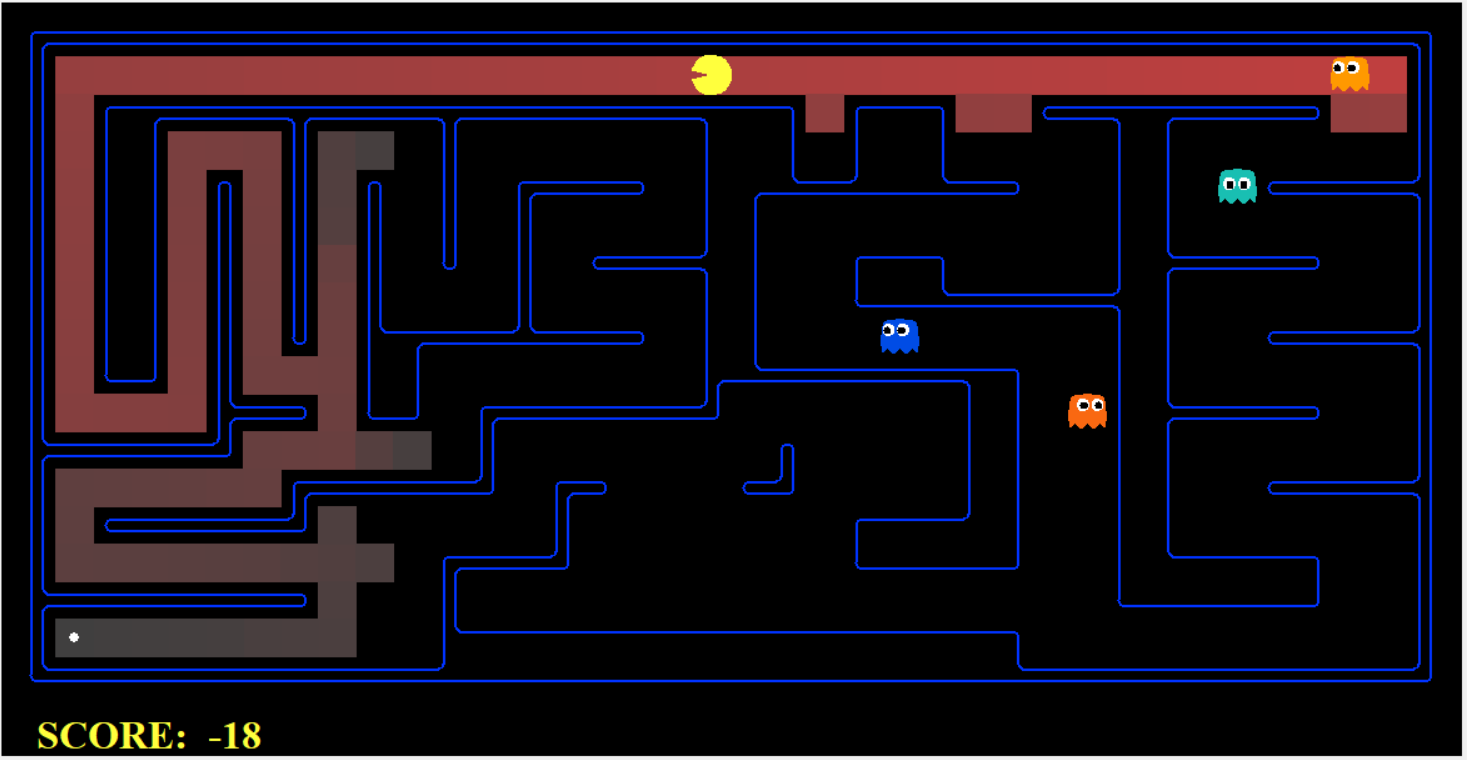


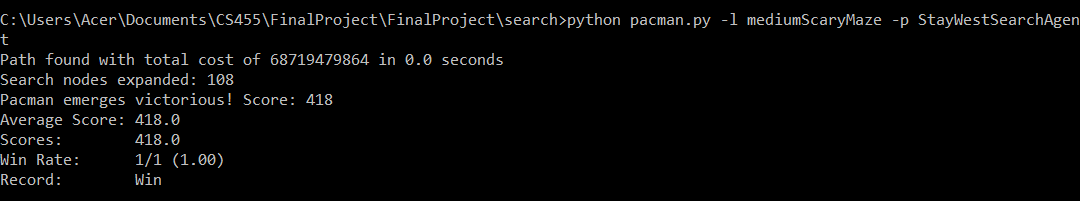
Problem 3

The results from this problem were a bit confusing, despite getting the correct outcomes. The two cost functions that were given seem to be plain functions that simply divide or multiply the current cost with each step. Running the command “python pacman.py -l mediumDottedMaze -p StayEastSearchAgent” gives a very small final cost as stated it should. Conversely “python pacman.py -l mediumScaryMaze -p StayWestSearchAgent” gives a very large final cost as it should. The cost functions that were given seem like they could be much better.



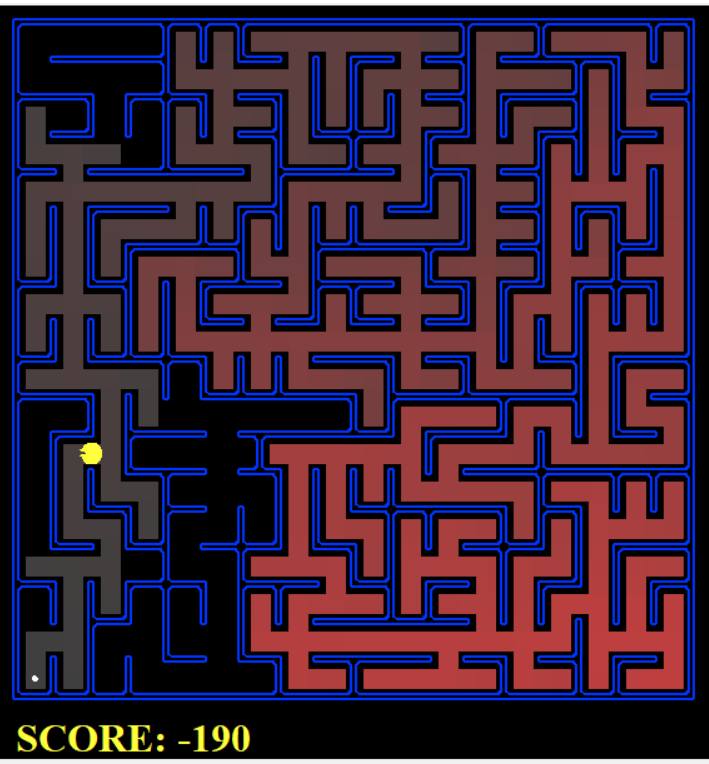


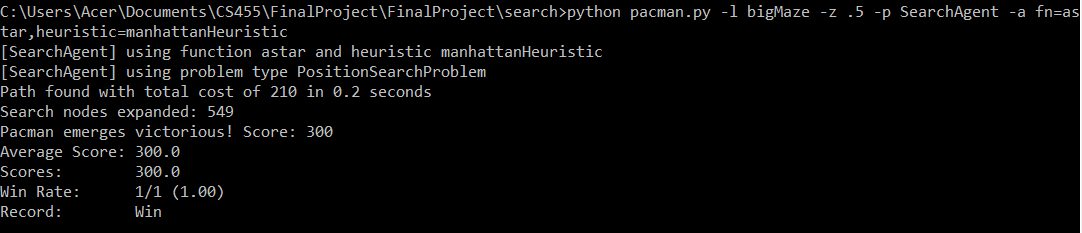




Problem 4

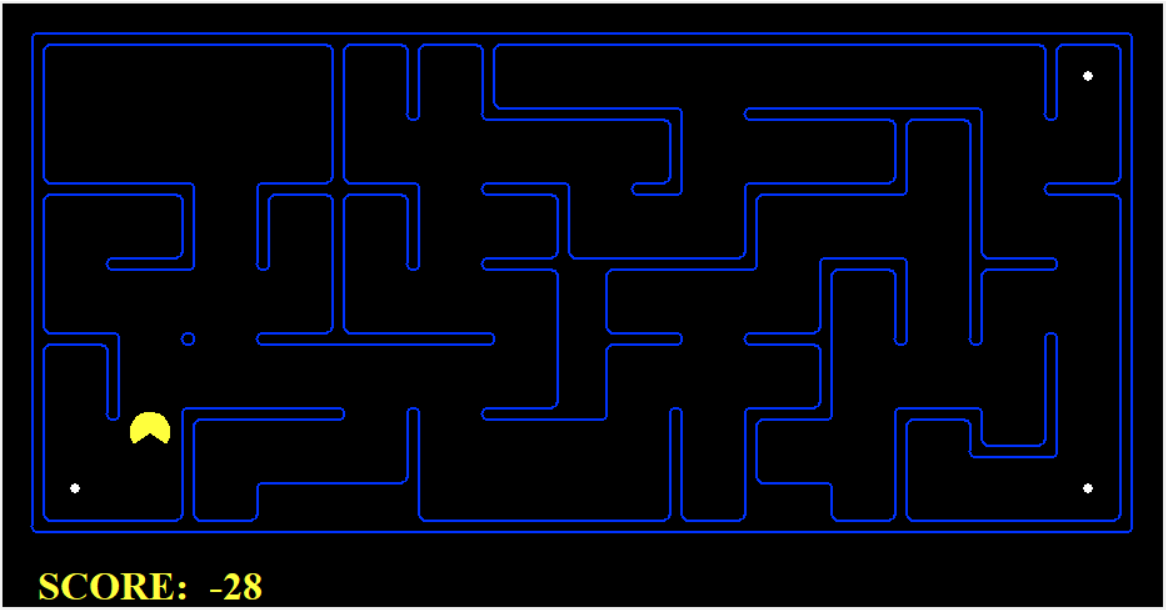
The A\* algorithm performed slightly better than the uniform cost search. Using the big maze to compare the two, A\* found a solution after expanding 549 nodes. Uniform cost search required 620 nodes to be expanded to find the solution.

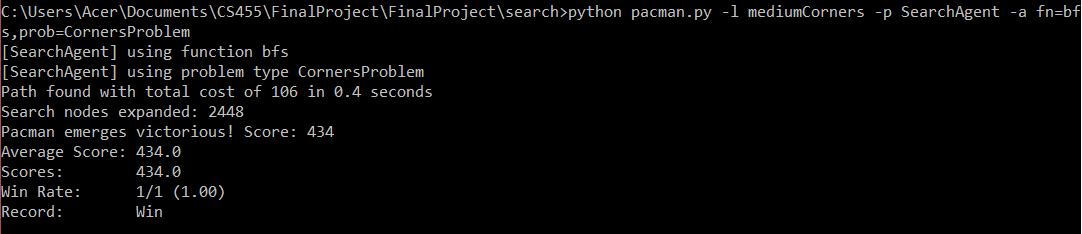




Problem 5

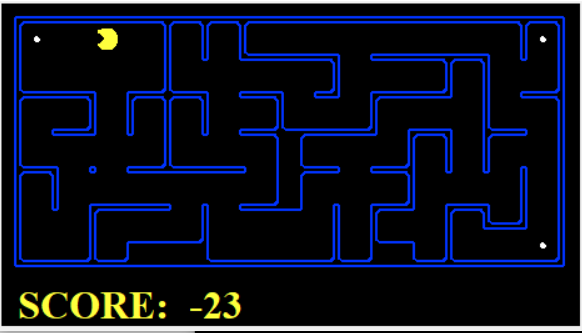
For this problem, my implementation of the corners problem is correct, but slightly less efficient than the optimal method. The optimal method can reach the four corners in under two thousand nodes expanded. The implemented method requires 2,448 nodes to be expanded. While this does not cause an issue for this problem, larger scale problems may see a performance hit because of this.

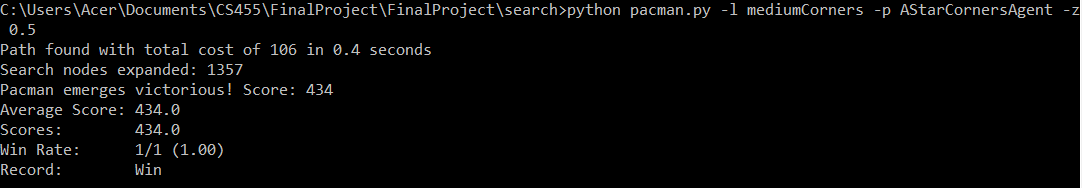




Problem 6

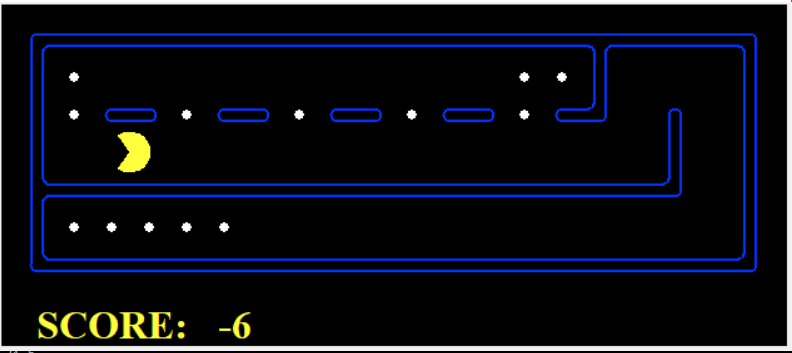
The results of this problem are identical to the results from the previous problem. The implemented solution is correct, but a more optimal solution exists. The implemented solution expands 1,357 nodes, whereas the optimal solution expands under 1,200 nodes. Again, for a problem of this size, this difference does not affect anything. Larger systems however could potentially be impacted by this minor difference.

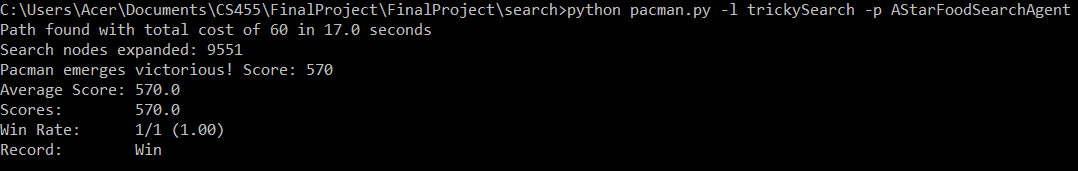




Problem 7

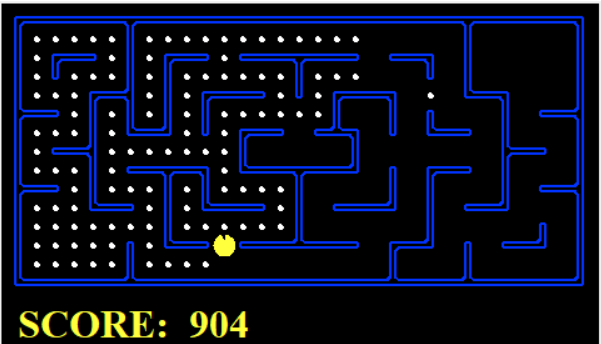
The results for this problem are again identical to the previous problem. The optimal solution can solve this problem in under 7000 nodes expanded, while the implemented one requires 9,551.

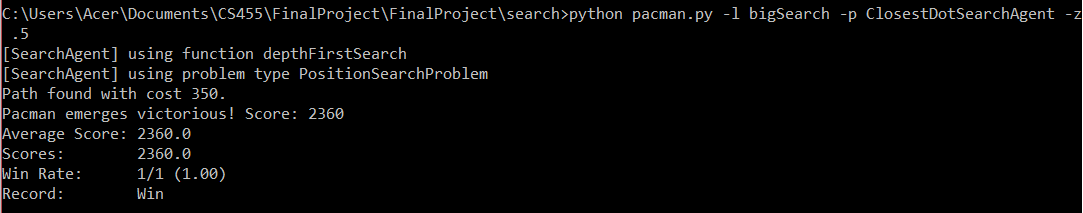




Problem 8

This problem uses an algorithm of eating the closest dot to Pacman. While this in theory is a good algorithm, it often leads to stray dots being left on the map as shown in the graphic below. This can lead to multiple nodes being repeat visited multiple times.





## Conclusion

If I were to continue working on this project, I would love to implement a cost function to use on the base Pacman game. My first idea for a cost function would be to increase the cost of paths that have ghosts within a certain distance of them.

## Appendix A: Lessons Learned

This project really helped me learn about various applications that search algorithms. It also helped me become comfortable with coding in Python. The given code was written extremely well. I learned a lot from simply browsing through the given code. My favorite part of the project was seeing how the breadth first search algorithm could solve the random eight puzzle problem so easily. As a kid, I hated the eight-puzzle problem because I could never solve it.

## References

No reference used.