Nicholas Allaire, A10639753
Son Tang, A11370127
Marcel Aguiar, A10904034

In this programming assignment, we come up with algorithms that determine which paths a Pacman agent should go through in order to reach a desired goal efficiently.

(1)The first problem is to find a dot on the map by using a Depth First Search (DFS) algorithm. We implemented this algorithm using the Stack (LIFO) data structure found in util.py. The problem with this approach is that it takes a very long time for the algorithm to find the desired location with a big-O of $O(b^m)$ where b is the number of branches and m is the maximum depth.

(2) The second problem is similar to the first, except that is utilizes Breadth First Search (BFS) algorithm to navigate to the goal state. BFS requires the use of a Queue data structure, which was provided for us in the util.py file. A queue uses the concept of first in, first out (FIFO) to allow us to always be searching the closest nodes. We maximize the efficiency of this algorithm by not expanding already visited nodes, giving BFS a big-O of $O(b^d)$ where b is the number of branches and d is the depth of the shallowest solution.

The biggest problem with depth first search is that the algorithm is not complete. It will return a path the moment it finds it rather than finding the most optimal solution. To prove this, we run both of them on a medium sized pacman game with the functions:

python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
python pacman.py -l mediumMaze -p SearchAgent -a fn=dfs

The breadth first search algorithm finds a path with a total cost of 68 while the depth first search algorithm returns a path with a total cost of 130. Although DFS may save space and may return a solution more quickly than BFS in some situations where it goes straight to the solution, it is not complete and does not return the optimal solution.

(3) For problems 1 and 2, there was never a decision that had to be made by Pacman since every path cost was equal to 1. But this is not always a realistic scenario, which brings us to problem 3. In problem 3, we take into account path values, i.e. a path with a ghost will have a higher path cost than a path without a ghost. To implement this, we used a uniform cost search that requires a priority queue, which was provided to us in the util.py file. A priority queue differs from a standard queue in that it prioritizes the nodes pushed onto the queue so that, in our case, the lowest cost node is popped off first. The path cost is determined when a node gets expanded, it adds the current node cost to the path cost to each successor node. With this approach, the big-O results in $(b^{l + C/e})$, where b is the branching factor and l is the depth limit. Uniform cost search is also complete as it provides us with the optimal path with a total cost of 68 (the same as BFS) when we run:

python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs

(4) We then implemented the A* search using the uniform cost search as a base. Like uniform cost search and BFS, A* search returns the optimal path with a cost of 68 on the same function with a null heuristic. A* search allows us to determine which nodes are more valuable than others by using heuristics and thus returning the optimal solution more quickly than the uniform cost search. We tested this with the given manhattanHeuristic which adds a value, or heuristic, to each node depending on its manhattan distance from the goal. This will make the search algorithm search through the paths that are closest to the goal first.

This way we can find the optimal solution slightly faster than uniform cost search. The results of the four search algorithms are shown in Table 1.

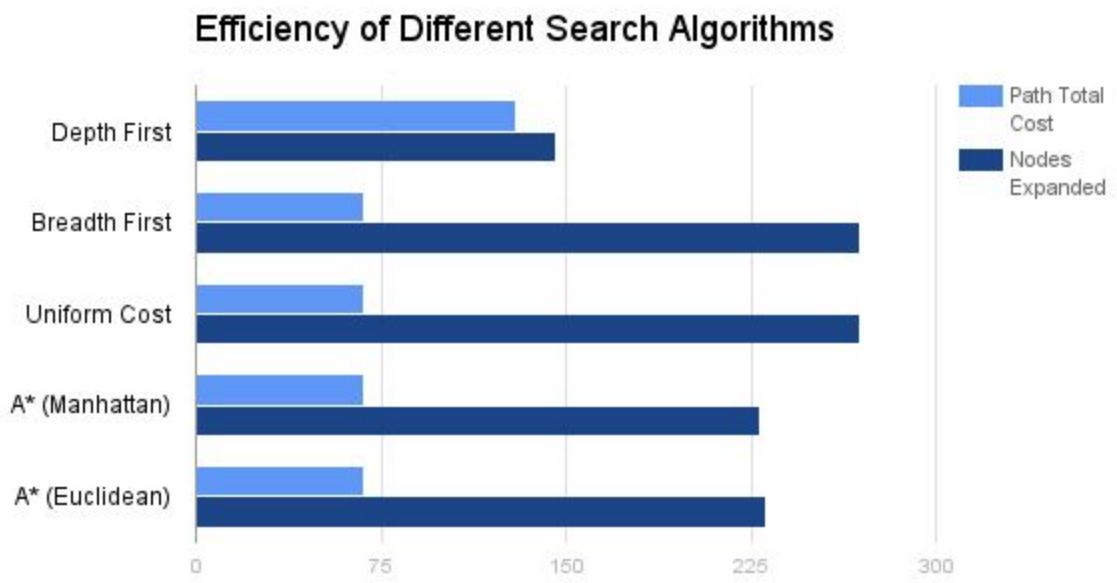

Table 1. Efficiency of Different Search Algorithms. The four search algorithms: Depth First Search, Breadth First Search, Uniform Cost Search, and A* Search (with manhattan and euclidean heuristics) were run on mediumMaze with uniform cost values. The total cost of the path returned by the algorithm and the number of nodes expanded to reach the goal were recorded.

(5) A* Search, which brings in heuristic functions, allows us to expand our search algorithms to more problems. In problem 5, the goal is to find the shortest possible path to reach all four corners using breadth first search. To simplify this problem, we created our own state representation to encode whether or not all fours corners have been reached. Our state representation was (x, y, cbl, ctl, cbr, ctr), where x and y are the current Pacman coordinates, cbl stands for corner bottom left, ctl stands for corner top left, cbr stands for corner bottom right, and ctr stands for corner top right where they are all boolean variables that are set to true when that corner has been reached. When we finished implementing the conditions, the pacman would go through all four corners to complete the goal with the trivial heuristic.

(6) Our next problem then is to implement our own non-trivial, consistent heuristic that will reduce the number of nodes expanded, and thus the time it takes to reach the goal. In problem 6, we solved this problem by implementing a heuristic that determines the euclidean distance from a node to the nearest corner that had not been reached yet. When all four corners have been visited, the heuristic would be 0, which is the goal. This heuristic makes the pacman search in a direction of the nearest corner first, and go to the next corner from there. This heuristic is admissible and consistent because the euclidean distance cannot be greater than the distance that it will take the pacman to reach the goal because at best the distance will be equal on a straight line. This heuristic also does not take walls into account. However, this heuristic addresses the situations when a corner has been visited or does not have a food because it does not search a corner that is marked as true in the tuple. On the autograder, this heuristic expanded a total of

1564 nodes. We also tried a similar heuristic that determines the manhattan distance to the nearest corner, but it is less efficient as it expanded a total of 1682 nodes.

(7) Next we want to build on the A* search in problem 4 and use a non-trivial, consistent heuristic to guide Pacman to every food location in as few steps as possible. With the command: python pacman.py -l trickySearch -p AStarFoodSearchAgent, our current heuristic, which uses the food grid as a list and pops the next food location, and then calculates the euclidean distance from pacmans current position to the popped food location. This heuristic successfully eats all of the food by expanding 13045 nodes and is consistent and admissible by the same reasons stated in problem 6.

(8) Finally, our last problem consists of creating an agent that greedily eats the closest food. To accomplish this, we searched all of the food locations in a for loop and found the nearest food location to Pacmans current location. When running the command: python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5, we also achieved a path cost of 350.

However, the ClosestDotSearchAgent does not always find the shortest path through the maze because there might be a food that is the same distance as another food, but so it is ignored. It then goes on a path (because it looks for the closest food) away from the initial uneaten food that leads far away from it. In this case, the path is not the shortest because the pacman has to go back a large distance to finish off that lingering food. This case was seen when we ran:

python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5

Nick: In this PA, I found a group late and by the time I joined Son had already wrote the code for BFS and DFS. After that I helped write and debug the code for the UCS and A* definitions in search.py. I also helped work on the implementation for getStartState, isGoalState, getSuccessors, and init in the CornersProblem class. I also helped discuss the strategy and pseudocode for tackling the cornersHeuristic definition but wasn't available to help code it. I contributed to a lot of the write up and commenting of the code.

Marcel: Spent a large amount of time getting Python to work on Windows. After being slightly behind, I discussed with my team members the reasoning on how we implemented 1-4 (all very similar, with changes in data structures used and implementation of a method to remember the path back through parents). Discussed heuristic choices for Problem 5-8 and why they weren't expanding as few nodes as they should have. Brainstormed most meaningful data comparisons for graphs that best demonstrated the time and space efficacy of each search algorithm.

Son: Started the code and coordinated with the team to get the project going. Main writer of the search functions and search agents. Figured out and implemented the current heuristic functions for the corners problems and eating all the dots. Responsible for most of the code implementation and testing. Contributed to the write up, commenting, and made the graph.