Introduction

For this project, I was tasked with analyzing four search algorithms and their effectiveness in arriving to an optimal solution for different mazes in the pacman world. I will begin by explaining the functionality of the different algorithms based on their implementation in *search.py*. I will then analyze how these different algorithms perform when tasked with the problem of finding and eating a single food dot in a given maze. My analysis will comprise of testing the algorithms on several mazes that I design myself and comparing their performances. I will do this by plotting the number of expanded nodes, the cost of the path, and the time taken to execute. I will then study the problem of efficiently eating every food dot in a given maze. I will be making use of several search agents provided to us and analyzing their effectiveness in producing an optimal solution on mazes of different size and complexity. This will highlight the aspects of each algorithm that makes them suitable or not suitable for the task at hand.

Understanding Search Algorithms

Generic Search

This general graph search has the basic structure of a search algorithm because expands unvisited nodes at every step until a solution is found, or until there are no more nodes left to expand. It keeps track of nodes that have already been expanded in a "visited" array as graphs can contain cycles. This generalized algorithm can be easily used to apply Depth-First, Breadth-First, Dijkstra's, and A* Search. What makes these four search algorithms unique, is their method of evaluating the next node to explore.

Depth-First Search

Depth-First Search traverses a graph by expanding the "deepest" of the current neighboring nodes in the graph. This method uses a "Last in First Out" queue, otherwise known as a Stack. This means that the node which is most recently generated is the one chosen for expansion. Depth-First Search is an exhaustive search and will therefore always produce a solution if one exists. However, Depth-First Search does not necessarily produce the optimal solution, as it does not always produce the least cost path.

Breadth-First Search

Breadth-First Search traverses a graph starting at the source node. It follows by expanding all of the neighbors of the source node and then expanding all of the neighbors of those nodes and so on. In other words, at a given step (distance from the source), all nodes are expanded before moving to the next set of nodes in the graph. Breadth-First Search uses a "First in First Out" queue, generally referred to as a traditional Queue. This search method will

always produce the “least step” path, or the path that requires the least number of moves to produce a solution. It makes sense then that the solution produced by Breadth-First Search is only optimal when all “steps” have equal cost. The downside is that often times, Breadth-First Search expands the largest number of nodes before reaching a solution (when compared to DFS, Dijkstra, and A*). BFS like DFS is an exhaustive search and therefore will always produce a solution if one exists.

Dijkstra/Uninformed Cost Search

Dijkstra’s Algorithm is similar to Breadth-First Search in the sense that it traverses the graph by expanding nodes based on their distance from the source. For a node to be chosen for expansion in Dijkstra’s Algorithm, it must have the lowest cost of any nodes being considered. We use a Priority Queue data type to ensure that the nodes are stored in order of lowest cost, so that we can pop the appropriate node at every level. Dijkstra’s Algorithm uses the cost function “f(n) = g(n)”, which only considers the cost to get from the source to each node. This algorithm finds the shortest path from source to every other node in the graph.

A* Search

A* Search evaluates a node using a more complex cost function than that of Dijkstra’s, namely “f(n) = g(n) + h(n)”. It is comprised of the cost accumulated thus far “g(n)” and the heuristic cost “h(n)” of reaching the goal state. This search method uses a greedy strategy to choose which node to expand next based on the value of the cost function mentioned above. Note that A* is considered an informed search because it accounts for a heuristic cost. If we did not consider a heuristic cost, that is h(n) = 0, then A* would default to Dijkstra’s.

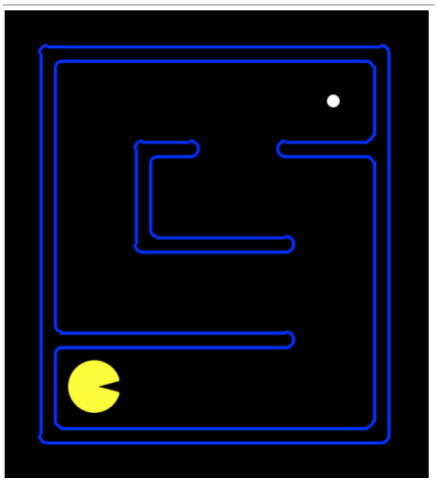
Eating a Single Food Dot

State Space Representation and Explanation

In states for the search problem where pacman only has to reach a final position given an initial position we only have to represent pacman’s current position. Then we have x * y possible states where x and y are the length and width dimensions of the maze. In other words, a given (x, y) represents an agent position. Therefore, the size of the state space is x * y. The initial state can be defined by (x_i, y_i) and the final position (goal state) as (x_f, y_f), where pacman has reached the food dot. The operators here are defined by the different positions the agent can face, namely North, South, East and West.

Maze Design

myMaze1



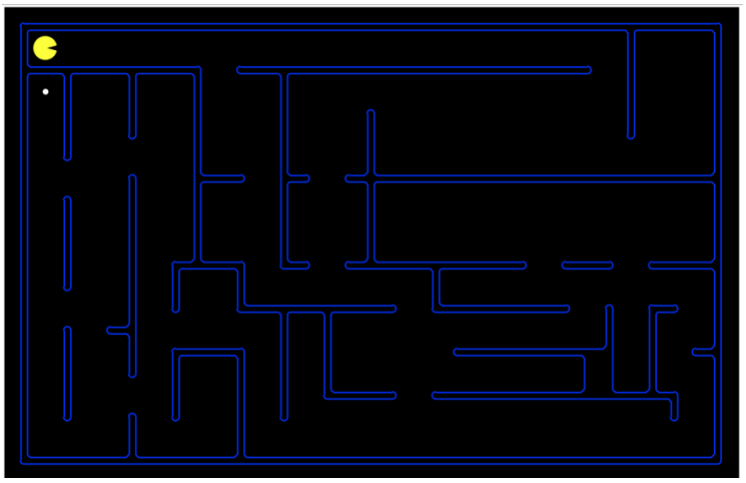
I initially intended for DFS to return the optimal solution, however this was not the case. Because DFS uses a stack datatype, it always chooses the most recently generated node for expansion and that is why DFS expands a fewer number of nodes than BFS in this graph. It does not, however, promise the fewest number of expanded nodes or the least cost path of all four search algorithms. In this maze, A* (Manhattan), A*(Euclidean) and BFS return the least cost path, but only A*(Manhattan) produces the optimal solution. As is shown in the plots below A*(Euclidean) expands one more node than A*(Manhattan) because of the heuristic cost considered when taking the next step. The Manhattan heuristic is better at estimating the heuristic cost of reaching the goal state because pacman can only move in one of four directions at a time.

myMaze2

This was the second maze I created, and it is particularly interesting because DFS expands the fewest number of nodes among all four search algorithms but produces a non-optimal path of a greatest cost of all four search algorithms. This is because the target node is quite far from the source node, so DFS is better suited to discover a path between them before any of the other algorithms. A* Search is by far the best algorithm to use here because it

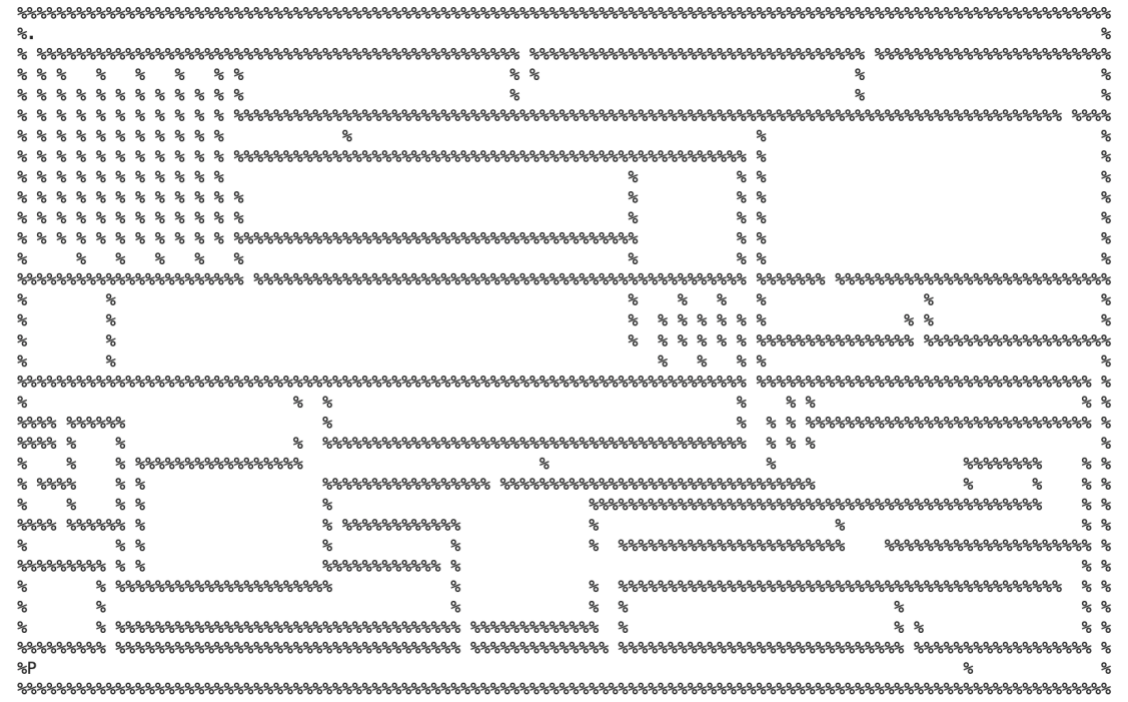
promises the optimal solution. It is the algorithm with the second fewest expanded nodes and also returns the lowest cost path.

myMaze3



In this maze, Depth-First Search again expands the fewest number of nodes of all four search algorithms but returns a path of the greatest cost of all four search algorithms showing once again that it is not optimal in finding the least-cost path. A*(Manhattan) outperforms Depth-First Search because Depth-First Search considers the large (more than 1 node wide) empty spaces between walls as one path and doesn't care about the cost of an action, whereas A* uses its heuristic cost to promise a the most optimal path even if it means expanding more nodes. At each step, A*(Manhattan) Search removes the node with the lowest cost is from the Priority Queue, the cost values of its neighbors are updated, and those neighbors are added to the queue. The Manhattan heuristic is better suited for mazes when compared to the Euclidean heuristic because you can only move in four directions.

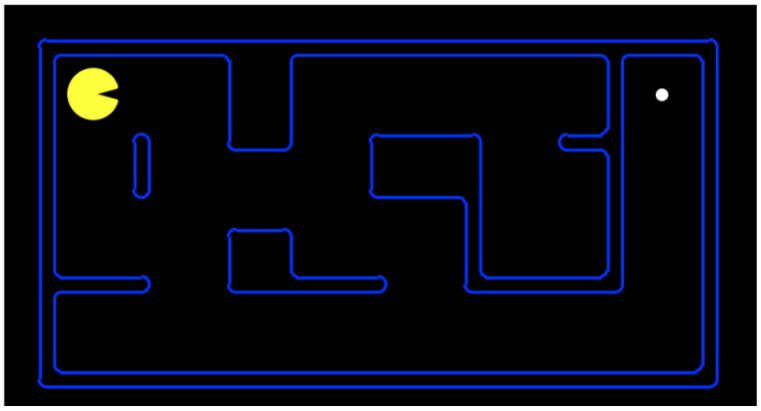
myMaze4



This is my fourth maze. As it is too large to capture in the pacman world, I have included the layout file for reference. This maze performs similarly to my first maze in the sense that Depth-First Search neither promises the fewest nodes expanded, nor the least cost path. Breadth-First Search explores the greatest number of nodes, as can be expected, and A*(Manhattan) again produces the optimal solution by guaranteeing both the least cost path and the lowest number of explored nodes. Depth-First Search does not explore the fewest nodes in this maze (but comes close) because of the size of the maze, the large spaces between walls, and the fact that the path the algorithm takes to the goal is much longer than required.

Additional Mazes

myMaze5: DFS outperforms A*(Manhattan)



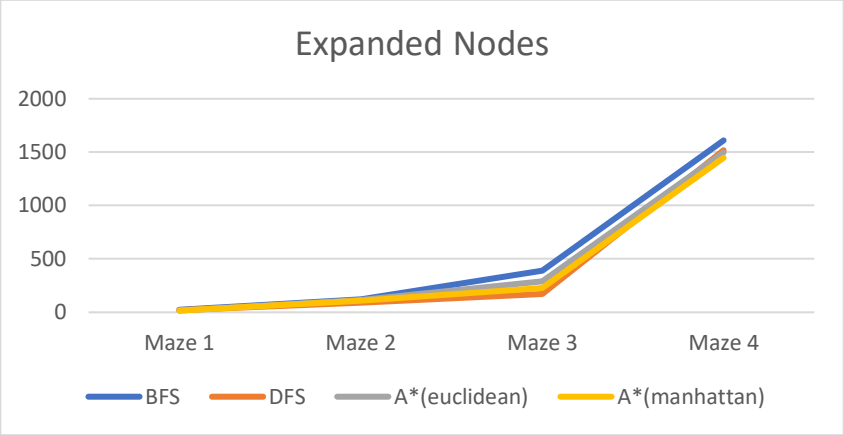
DFS: 35 nodes expanded, optimal path of cost 22
A* Manhattan: 40 nodes expanded, optimal path of cost 22

This maze is interesting because here, DFS expands a fewest number of nodes of all four search algorithms AND produces the least cost path. In this case, DFS returns the optimal solution because the path it explores first connect directly with the destination. With A*(Manhattan) there are many points where there are several possible actions to be taken that put pacman closer to the food dot, and therefore this algorithm must expand more nodes to find the least cost. In other words, A* with the Manhattan heuristic explores nodes at each level based on which action puts pacman closer to the goal state. Depth-First Search ignores these actions and follows one path to the source, a path which happens to be the optimal one.

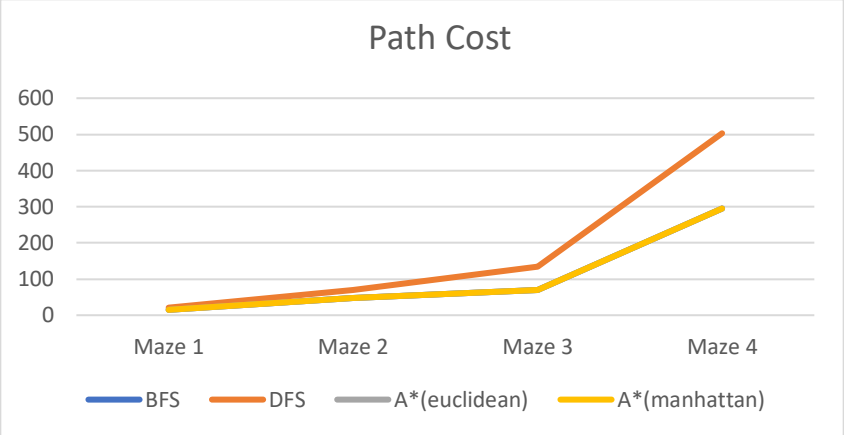
Maze: BFS outperforms A*(Manhattan)

I don't think that a maze in which Bread-First Search expands fewer nodes than A*(Manhattan) exists. I can construct mazes where the two search algorithms promise the optimal solution, but not one where Breadth-First Search out-performs A*(Manhattan). This occurs because the A* search algorithm is a specialized case of BFS that uses a priority queue that greedily selects the least-cost action from nodes at the frontier. Therefore, A*(Manhattan) will always perform the same if not better than Breadth-First. In addition, the Manhattan Heuristic proves to be admissible because of the structure of the pacman mazes. Both promise a least-cost path, but A*(Manhattan) will always be favorable over Breath-First Search.

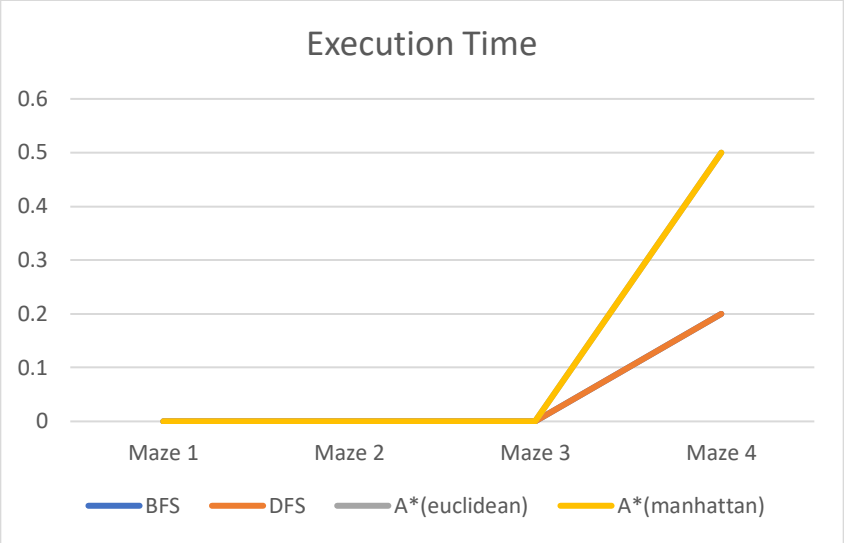
Plots



	BFS	DFS	A*(Euclidean)	A*(Manhattan)
Maze 1	24	21	17	16
Maze 2	120	91	116	111
Maze 3	392	171	293	227
Maze 4	1610	1518	1497	1447



	BFS	DFS	A*(Manhattan)	A*(Euclidean)
Maze 1	15	21	15	15
Maze 2	47	69	47	47
Maze 3	70	134	70	70
Maze 4	295	503	295	295



	BFS	DFS	A*(Euclidean)	A*(Manhattan)
Maze 1	0.0	0.0	0.0	0.0
Maze 2	0.0	0.0	0.0	0.0
Maze 3	0.0	0.0	0.0	0.0
Maze 4	0.2	0.2	0.5	0.5

Analysis

These three graphs demonstrate how Breadth-First Search, Depth-First Search, A*(Euclidean) and A*(Manhattan) perform on the four different mazes I designed. They consider the number of expanded nodes, the cost of the path taken and the execution time of each algorithm.

One trivial observation is that the number of expanded nodes increases as the size of the mazes increase. Another observation is that depending on the distance between the food item and pacman, the algorithms perform differently. Breadth-First Search is clearly the least suitable for these maze searches because although it promises the least-cost path, its method of considering all neighbors first usually results in the highest number of expanded nodes. It is evident that Depth-First Search is more suitable for games and puzzle problems because when we make a decision using this algorithm, we explore all paths through that decision. If this decision leads to a win situation, we stop. Generally, it seems that Depth-First Search works well when the solution lies away from the source, whereas Breadth-First Search is better for searching for solutions that are closer to the given source.

Another thing that can be gathered from these plots is that the A* search algorithm performs the best on almost all graphs, with the exception of graphs like myMaze5. Because at every step A* determines which action has the least cost by considering both the cost of expanding the next node and the estimated cost of reaching the goal state from that node, it is the best suited algorithm for this maze problem. The Manhattan heuristic works better than the Euclidean heuristic. The two-dimensional structure of the maze combined with the fact that pacman can only move in four directions (NSEW) favors the Manhattan heuristic because it

produces a more accurate estimate for the cost of reaching the goal state from the next possible step.

Eating All the Food Dots

State Space Representation

In the state where Pacman has to reach and eat all the food dots, we have to represent pacman's position and if there is or is not a dot in each of the positions where there could be one. This can be represented using Booleans (yes or no) for each of these positions. This means that there are 2^n possible states for the food dots where n is the number of food dots. Therefore, the size of the state space is $x * y * 2^n$. The initial state is defined by the initial position (x_i, y_i) where pacman starts (denoted by P in the layout file) and the Booleans of the food dots are initialized to true. The final state is one in which all the Booleans representing the food dots are false.

Agents

A* Closest Food Manhattan Distance

This Agent used the A* search algorithm and for the heuristic cost, the Manhattan distance between two points. This can basically be defined as the shortest horizontal and vertical distance between two points. The distance is calculated by summing the absolute value of the difference between the two points.

A* Closest Food Maze Distance

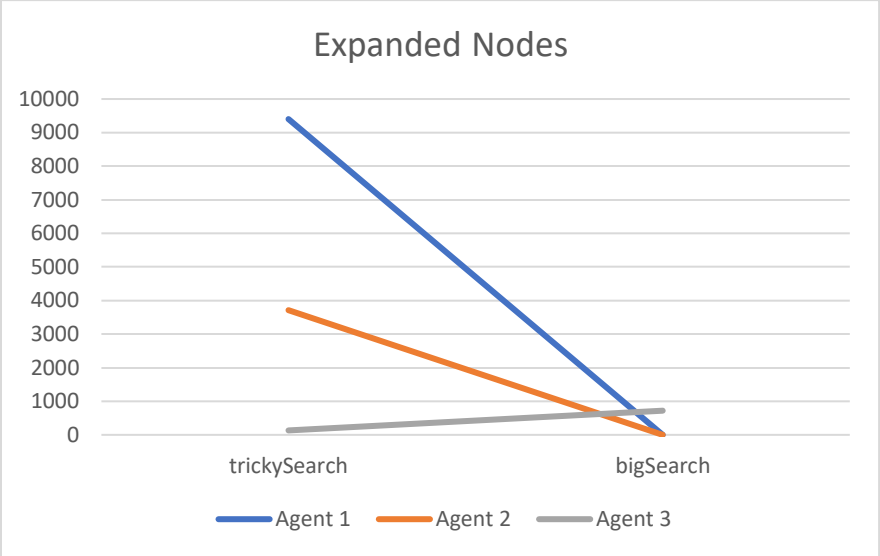
This Agent also used the A* search algorithm. Here, the heuristic cost considered the maze distance. It is defined as the length of a Breadth-First Search from the current location to the given food dot while also considering the initial game state. It can be calculated by running the position search problem from pacman's current position to the food item/goal state

Closest Dot Search

This Agent uses Breadth-First search to search for one food dot at a time by combining several position search problems. Once the closest food item is discovered, pacman takes that path. From each subsequent state, pacman checks if there are any food dots left in the maze and if so continues this search.

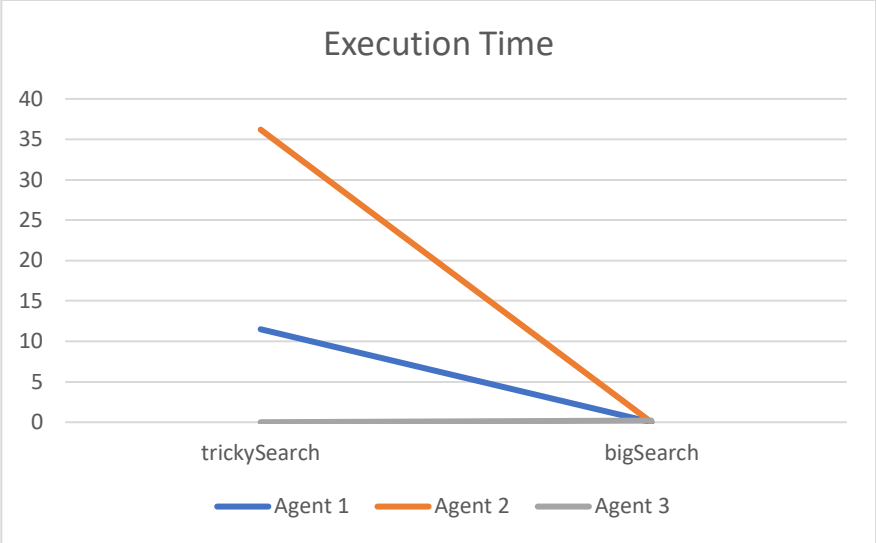
Plots

Note: I could not produce results for either of the A* search agents on the bigSearch maze (I even left my computer running overnight), so I explain a reason for this happening in my analysis.



	Agent 1	Agent 2	Agent 3
trickySearch	9402	3712	132
bigSearch	N/A	N/A	719

	Agent 1	Agent 2	Agent 3
trickySearch	60	60	68
bigSearch	N/A	N/A	350



	Agent 1	Agent 2	Agent 3
trickySearch	11.5	36.2	0.0
bigSearch	N/A	N/A	0.2

Analysis

This problem is clearly far more interesting than that of eating a single food dot. In this problem we encounter a search space that is exponentially large because even though the maze might be small, each possible subset of food dots represents a different state in the search space. This makes the A* algorithm a weak choice for the problem at hand regardless of the heuristic chosen.

It is evident when looking at the comparison of expanded nodes among the three agents that the A* Manhattan agent expands by far and away the highest number of nodes to solve the trickySearch maze. In addition, even though the A* (maze distance) only expanded a third of the nodes that A*(Manhattan) did, it was still on the order of 28 times as many as our Breadth-First Search agent. As far as the difference between the two A* search agents, it can be explained by noting that the A*(maze distance) agent uses a Breadth-First Search to evaluate the "lowest cost action" to be taken which is more efficient for this problem.

When looking at the path costs we see the tradeoff between space complexity and the ability to truly produce a least-cost path. The Breadth-First Search agent yields a path of cost 68 while both A* search agents produce a path of cost 60.

The most interesting of the three plots is the one that compares the execution times of the three agents because these results led me to question whether or not the time complexity was a factor in my code not executing in reasonable time. The Breadth-First Search agent was the breadwinner, executing exponentially faster than the next fastest agent, and was actually able to run on bigSearch in a mere 0.2 seconds.

The trickySearch maze to be tested on in this part of the assignment is sparse, meaning most of the maze was empty with only a few food dots. So, we could speed things up drastically by only considering the food. Instead of finding the shortest path in a maze of n nodes, we only

care about a subset of those nodes. This basically allowed us to solve a problem in which the maze we considered was comprised of only that subset of nodes.

The way this is done is by finding the shortest distance between every pair of food dots and also taking into account the distance from pacman's current location to the food dot. Then the shortest path on the graph is computed. This approach leaves us with a hypothesis space of n! in which the agents try to find an optimal sequence of actions.

The bigSearch maze contained 220 food dots. This means that the size of our hypothesis space is 220! and it is therefore evident why Agents 1 and 2, which rely on the A* search algorithm, were unable to come to a solution. Instead, Agent 3 is the most optimal. It does not necessarily promise the least-cost path or the fewest number of expanded nodes but will execute in significantly less time, especially as the maze becomes more complex and we introduce more food dots. This tradeoff between space and time complexity and the ability to produce the least-cost path makes the "best" algorithm for the Agent 3 (if not only for the fact that it is the only one that worked for me).

Concluding Remarks

Eating a Single Food Dot

After running the four search algorithms on my five maze designs and evaluating their performance I think it is safe to say that A* (Manhattan) search was the best algorithm for the problem at hand. Breadth-First Search was least suited for this problem when compared to Depth-First Search, A*(Manhattan) and A*(Euclidean) because of its traversal method of exploring every node at a given depth before moving on. In addition, A* search is basically a modified Breadth-First Search that is considered "informed" because it uses a priority queue that evaluates the best action based on an intricate cost function. The Manhattan Heuristic proved to be the optimal heuristic for the job essentially because pacman can only move in four directions.

Eating All the Food Dots

Unfortunately, I could not produce results for the A* search agents on the bigSearch maze despite leaving my computer running overnight and for the better part of the next morning. This led to some difficulty at first as I thought it could be something other than the time complexity of the algorithms causing the execution to take as long as it did. It was not until I further analyzed the code and found that the size of hypothesis space was n! that I gave up running my code and sought to explain why I thought they would not run on such a large, filled maze. The Breadth-First Search agent however was able to produce results and although they were not optimal, they were promised in exponentially less time than the A* agents and therefore proved optimal for this problem.

Difficulties

- Determining why each of the search algorithms traversed the mazes as they did.
- Understanding why DFS was optimal for some mazes.
- Running the A* search agents on the bigSearch maze until I realized why it took as long as it did.

Challenges

- Designing the additional mazes to produce specific results (Where DFS was optimal over A*).
- Understanding why the time taken to execute the A* search agents grew exponentially with maze size and complexity.

Benefits

- I gained an understanding of how the four search algorithms can apply to game and puzzle problems and where they performed best
- I now have a clearer picture of how informed searches can be used in Artificial Intelligence
- I better understand the datatypes used by the search algorithms and their code implementation.
- I learned about some of the variables that should be considered when evaluating the performance of an algorithm.
- Translating code into concepts that can be extrapolated upon.