Artificial Intelligence Spring 2020

Final Project: Pac-man

Liam Creedon 100437408

Introduction

In this project, we were tasked with the analysis and comprehension of the code provided for the Pac-man game developed by UC Berkeley. The code we are most interested in is the movement of Pac-man himself, most specifically, the implementations used for him to search for various objectives. In order to do this, I will first analyze and describe the search algorithms as implemented in *search.py*. Then, I will look at two different search objectives: one that finds a single food item, and another that looks to collect multiple food items.

In searching for a single food item, I will show how this process is in essence a maze search, as there is sole objective. I will analyze and describe the performance of 4 different searches on novel mazes that I have designed. These new mazes are designed to show and compare the behavior of the algorithms, as I will also detail plots that compare the performance of each. In searching for multiple food objects, I will conduct a similar process in which I will analyze and describe the functionality and performance of the different algorithms, as well as comparing their performance using plots.

Understanding search algorithms

The given implementation in the *search.py* allows for a customizable *searchAgent*, permitting the mazes to be searched using different algorithms. The generic, 'dummy' methods in *search.py* simply define the type of data structure to be used in order to execute the given algorithm. Universally, the structures for each algorithm must be compatible with the *pop* and *push* functions, otherwise the given *generalGraphSearch* would yield errors. So, the meaning of the *structure* parameter in the general function, is used to determine which algorithm is being executed.

generalGraphSearch

This function provides the general code to search a given *problem* and will traverse through the maze differently depending on the type of *structure* it is utilizing. The different traversals arise through whatever will be *popped* from the *structure* at the

beginning of each step. The implementation begins when the current location is taken from the *path popped* from the *structure*. For each tile the user visits, a list of the *successors*, that is the valid tiles that can be visited from the current location, has their path generated, which is how to arrive at this tile from the initial state. Depending on the *structure* being used, the possible *successors* to visit, along with their corresponding *path*, will be *pushed* onto the *structure* in a particular order. This will then loop back up to where the *structure* is being *popped*. What allows this implementation to work is that each *structure* provided by the different search algorithms will accordingly make choices depending on its data structure.

Depth-first search – stack

By using a stack as the structure to represent DFS's traversal, we can ensure that the choices that bring Pac-man to the deepest part of the maze will be selected first in the traversal. This is because the stack is a LIFO structure, which will *pop* off the most recently generated successor nodes first. Once a node is exhausted, it is removed from the stack.

Breadth-first search – queue

By using a queue as the structure to represent BFS's traversal, we can ensure that the choices that bring Pac-man to the widest parts of the maze will be selected first. This is because the queue is a FIFO structure, which will *pop* off the generated successor nodes in the same order they were *pushed*. Once each level of nodes is exhausted, the subsequent level is visited.

Dijkstra's – priority queue

By using a priority queue sorted by *cost* to represent a maze search using Dijkstra's algorithm, we can guarantee that in this FIFO structure, the actions requiring the lowest *cost* will be *popped* first. This ordering is determined by backwards cost as defined in the dummy method. Goal states are checked after the node is selected, not when it is generated, as to ensure the lowest cost path and avoid non-optimal choices.

A* -- priority queue

By using a priority queue sorted by combined *cost* and *heuristic* to represent a maze search using A*, we can guarantee that in this FIFO structure, the actions requiring the lowest combined *cost* and *heuristic*, or *f(x)* where $f(x) = g(x) + h(x)$, will be *popped* first, that is the combined cost of reaching the next node, as well as the cost

of reaching the goal from the current node. This ordering is determined by backwards cost of *f(x)* as defined in the dummy method.

Finding a fixed food dot

PositionSearchProblem

The main difference between completing a maze search and a position search lies in defining what the goal state for the problem is. In this class's definition, we see this accomplished in the initialization of the *PositionSearchProblem* object. The default goal state is provided in the parameters as the location (1, 1) in the maze, or the bottom left corner. To modify this, we check for food items in the *gameState* object that has been passed in the parameters by calling its *getNumFood( )* and checking that there is exactly one food item in the given maze. Then, the location of the food item is gotten, and set to *self.goal*, or the goal state of the problem. If there is not exactly one food item in the *gameState*, then the goal state will be the location (1, 1) as passed in by the parameters.

State space:

Determined by the size of the gameState passed to this object, the state space will consist of the grid in which Pac-man can operate, where each possible location is represented with an (x, y) coordinate. So, there will be *x * y* possible states for Pac-man. Then we have to consider our goal state which is the single food item located somewhere within our maze. As the search will end upon finding the food item, this piece of information does not expand our state space. Therefore, the size of the state space is *xy*.

Initial state:

(x_i, y_i) where Pac-man begins, indicated by a 'P' in the layout file.

Goal state:

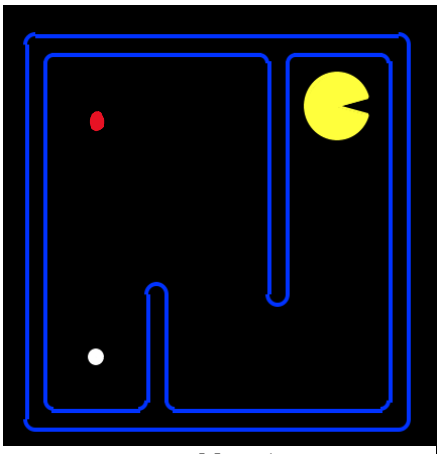
(x_f, y_f) where the food item is, indicated by a '.' in the layout file.

Operators:

The indicated direction N, S, E, W (valid moves defined by *getSuccessors( )*). If the goal state is reached, the food item will disappear, and the search problem is completed.

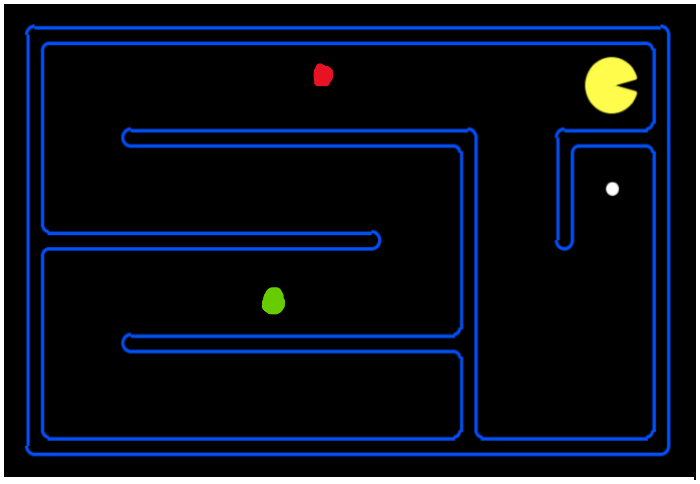
Analysis

Mazes:



myMaze 1

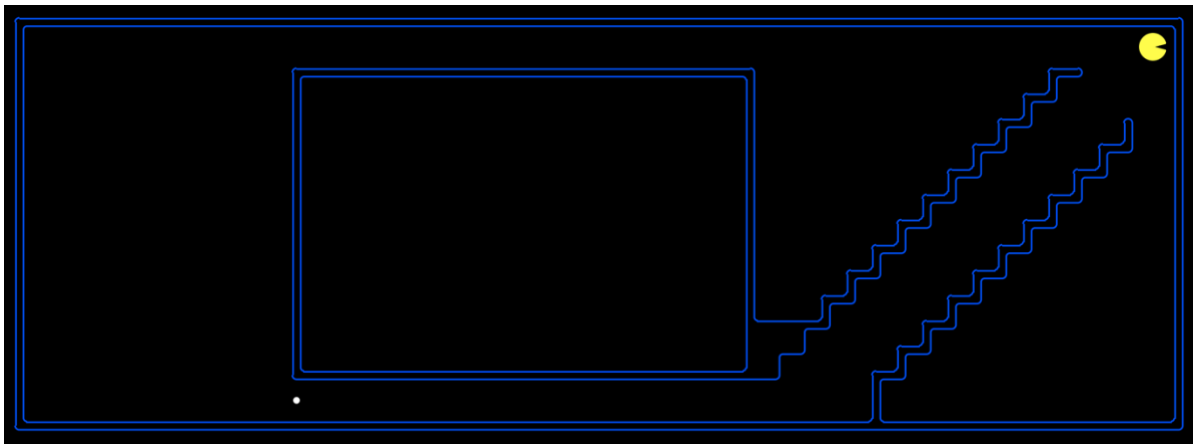
I am lumping my first two mazes together in order to show specifically some of the basic differences between a DFS and a BFS. Both of these mazes are very small in terms of size, and do not offer many decision-making points, as the path to the goal state is rather straightforward. This small size with simple decisions should allow both A* algorithms to find the goal state very easily. The differences between BFS and DFS that I want to highlight with these mazes should be seen when decisions are made at the red dots. In both mazes, I expect the final path (and thus the path cost) for both algorithms to be the same as there is a very clear path to the food item in both. The difference should be in the number of nodes that each search expands. In the first maze, a BFS will explore the nodes above the red dot, while a DFS will ignore the nodes above and continue west to the deepest part of the maze, where the goal lies. In the second maze, the goal state is very close to the initial state. At the red dot a DFS will continue west, exploring the deepest nodes of the path, and should not return to the red dot until it has expanded all nodes up to the green dot. A BFS will expand the node south of the red dot first, but also expand nodes west of the red dot. But a BFS won't ever reach the green dot, as it will only expand as many nodes west as it does south, which is halted once the goal state is reached.



myMaze 2

In the third maze that I designed, I begin to increase the size of the maze in order to try and see differences in execution time. I also want to try to show a few other behaviors of the searches in terms of decision making. With this maze, I predict that a DFS will not find a path with particularly low cost due to the large amount of open space that exists in the deeper (left side) of the maze. As for the other three searches, I predict that each will eventually find a similar path by traversing down the diagonal, but that the difference will lie in the number of nodes each algorithm needs to expand in order to find the food. Based on the layout, I predict that a BFS will open a lot of nodes (probably including the open space the DFS traversed through), A* Euclidean will not

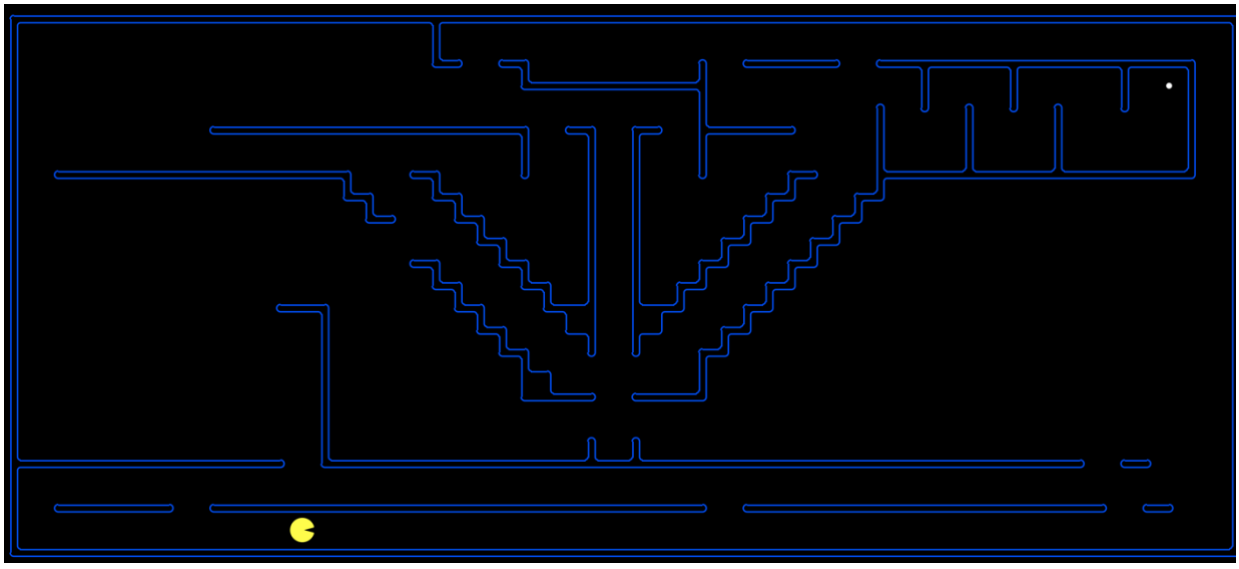
expand any further west of the food item but will expand almost all of the nodes east of the food, and I am aiming to trick the A* Manhattan into first traversing south, where it will get stuck and then discover the diagonal path. Overall, I want to show how a larger



myMaze 1

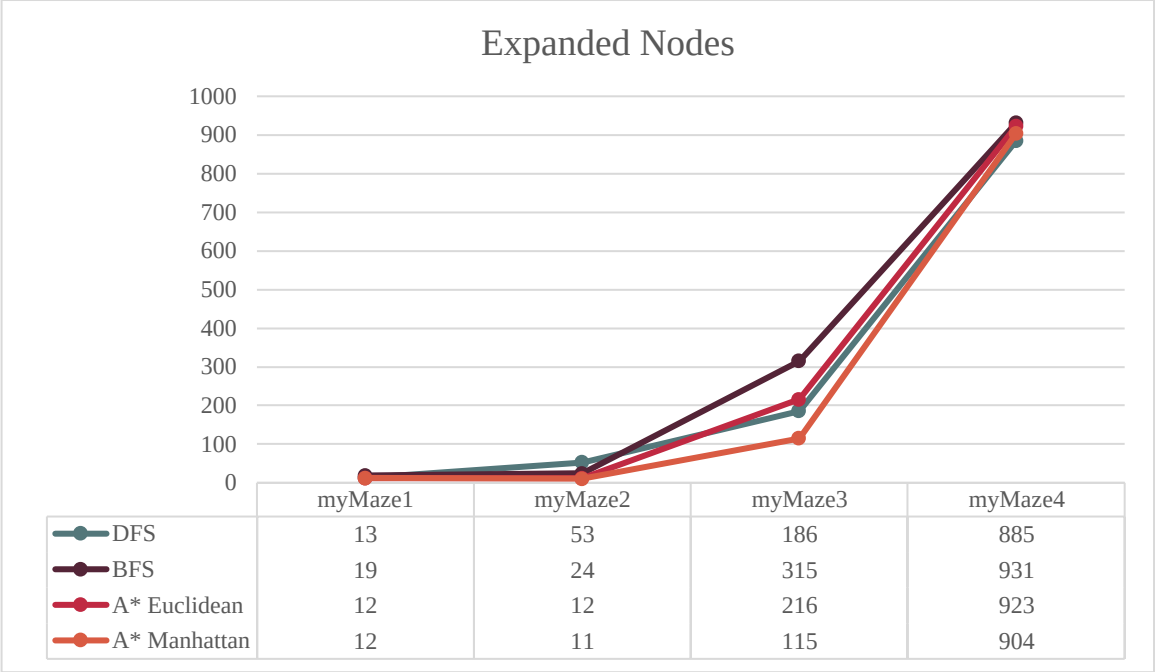
map, and more open space begin to affect the performance of each search. *(A brief note for maze 3, I blocked out the big space in the middle of the maze in order that the DFS did not expand an enormous number of nodes as to make the plots more readable in terms of scale).*

I continue to increase the size and complexity of the maze with my fourth design. My goal here was to really just give a lot of decisions to make and see what happens. The main difference I am expecting to see is in the execution time between the algorithms due to the size and number of decisions that have to be made. I expect all of the algorithms to expand a high number of nodes for the same reason. Similar to the other mazes, I expect that the eventual path that is found should be pretty similar among definitely both A* searches and possibly the BFS, but I predict that DFS will not find a low cost path as there are so many nodes to expand in the open space.



myMaze 4

Plots:



	myMaze1	myMaze2	myMaze3	myMaze4
DFS	13	53	186	885
BFS	19	24	315	931
A* Euclidean	12	12	216	923
A* Manhattan	12	11	115	904

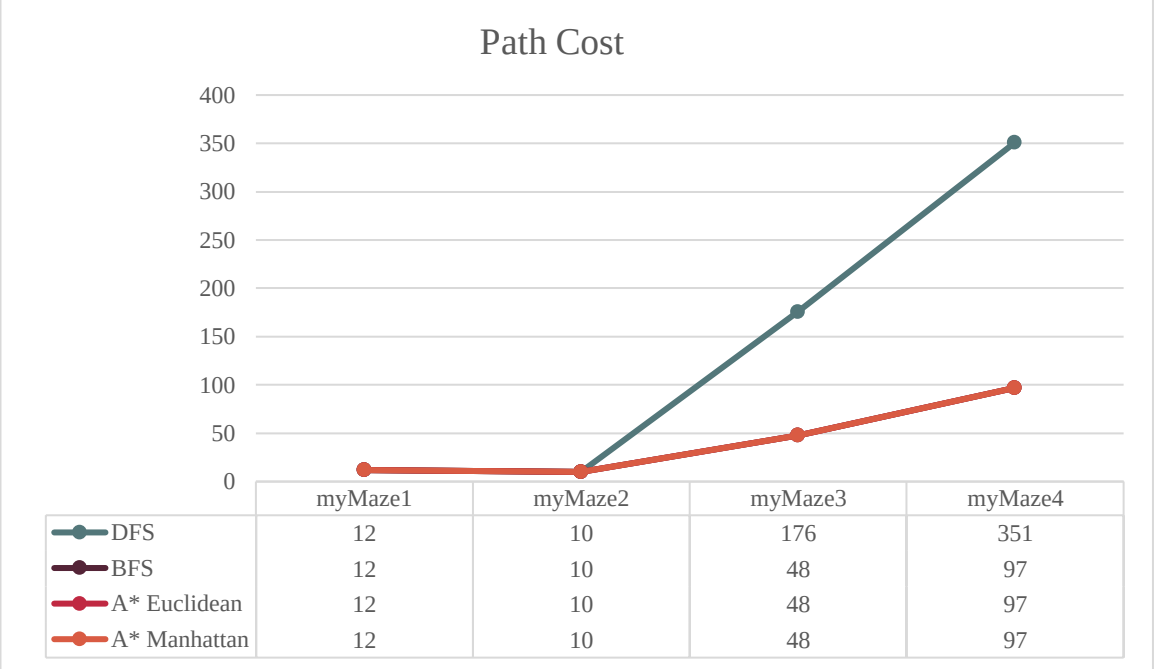
From this plot, we can see immediately the direct correlation between maze size and expanded nodes in general, however this is trivial and really should just be a logical intuition. What we are interested in lies in the difference from algorithm to algorithm.

For the first two mazes, in which I expected to see differences between DFS and BFS, we do. In the first maze, DFS expands fewer nodes than BFS because the food item is in the deepest part of the maze. DFS will find this path in its first traversal, whereas BFS will expand the entire board before discovering the food. In the second maze, DFS expands significantly more nodes than any of the other algorithms because of how deep the initial path goes. None of the other algorithms will traverse this deep because they find the solution before so. As the food item is very near the initial state, A* certainly performed well, as did BFS just not to the same extent. A* ignores the branch that traverses away from the food, and BFS will only expand as many nodes in that direction as it will in the south direction.

In maze 3 I saw similar behavior to what I predicted. While I wanted to show the large number of nodes DFS begins to expand, this search actually expanded fewer nodes than BFS and A*E. While I believe that on larger maps DFS will normally expand the most nodes, I think that the location of the goal state (in a deep part of the maze and along the first branch DFS will expand), in addition to the fact that there are more nodes to expand on the right side of the maze (which DFS never looks at) makes it so DFS doesn't actually expand that many nodes for this maze. The difference in nodes expanded for the other 3 searches show some key properties of the algorithms. For the

A* searches, we see some of the specific behavior of each heuristic. Euclidean begins to expand nodes along the top of the maze but does not expand any nodes west of the goal (as this would be taking the agent farther from the goal). Manhattan searches the bottom right corner of the maze first, as I intended, and cannot find a solution there. Rather than looking along the top of the maze like Euclidean, the agent steps down the diagonal to the goal. Looking at the two heuristics side by side, Euclidean prefers the 'hypotenuse' of the triangle formed in the bottom right, and Manhattan prefers the two sides of the triangle. This is why Manhattan searches in this corner first.

For the last maze, I was surprised again by DFS expanding fewer nodes than the other searches, this time outperforming all of them. I believe this can be attributed to the fact that DFS had fewer decisions to make as it explored mostly open space. The complexity of this maze made it so BFS and A* had to expand many nodes to arrive at the goal. Similar to my prediction, each algorithm expands a high number of nodes because of the complexity of the maze.



	myMaze1	myMaze2	myMaze3	myMaze4
DFS	12	10	176	351
BFS	12	10	48	97
A* Euclidean	12	10	48	97
A* Manhattan	12	10	48	97

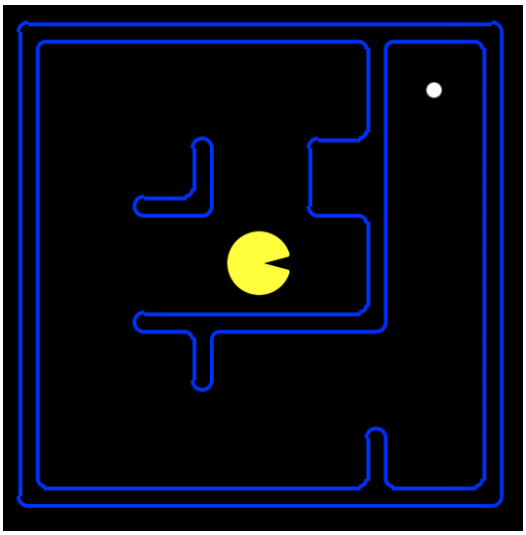
When looking at the path cost for each algorithm, we do not see too much variability. Apart from DFS, the other searches always find a path with the same cost. For the first two mazes DFS performs the same because of the simplicity of the maze, where there is not much (if any) open space for the DFS to traverse all the way through, so the path it eventually finds will be the same as the other searches. We see the differences in the higher complexity mazes when DFS is traversing through lots of open space in order to find the goal. So, for the last two mazes, while DFS actually expanded fewer nodes than the other searches, it suffers in optimality.

BFS and A* will always find the optimal path because in the case of BFS, all of the nodes have the same cost, and for A* both the Euclidean and Manhattan heuristics are admissible.

	myMaze1	myMaze2	myMaze3	myMaze4
DFS	0	0	0	0.1
BFS	0	0	0	0.1
A* Euclidean	0	0	0	0.3
A* Manhattan	0	0	0	0.2

The first three maze designs were not complex or large enough to measure differences in execution time. For the fourth maze we do see differences though. DFS and BFS both have $O(b^d)$ time complexity, so they will perform identically. Both A* algorithms have a higher execution time on maze 4 which can be attributed to the higher number of nodes this maze requires to expand. Each successor node must be placed in the priority queue with the proper priority, and the cost heuristic is calculated for each node too. A* took longer because there are no calculations involved in DFS and BFS node generation.

Other mazes:



DFS nodes < A M*

This is the first additional maze I designed in which DFS finds the optimal path and expands fewer nodes than A* with a Manhattan heuristic.

DFS: 22 nodes expanded, 18 path cost

A Manhattan: 29 nodes expanded, 18 path cost*

This occurs because of the location of the food dot relative to the initial state. The Manhattan heuristic will want to find the fastest, most direct path to the goal, so it expands the nodes to the north and east of the initial state as to reach the goal; however, the

agent gets stuck and then discovers the path. A DFS will move west from the initial state, and then south once the wall is reached. From that point, both algorithms will expand the same number of nodes, so the difference is found in the Manhattan heuristic's preference towards nodes that move closer to the food item.

As for a maze in which A* with Manhattan heuristic will expand more nodes than BFS, this is not possible. A* is just a more specialized case of BFS, so A* will only ever expand as many nodes as BFS, never more.

Eating all of the food dots

FoodSearchProblem

For this search problem, instead of searching for one food dot, the problem will search for multiple goal states. The implementation difference in the given code is in the way in which the goal states are communicated to the search. In the PositionSearchProblem, the initialization handles determining the goal state and it is stored as a class variable, namely *self.goal*. In this search, goal states are handled through each state of the successor generation. As each state of this problem is the tuple of *(Pac-manPosition, foodGrid)*, where *foodGrid* is a Boolean grid indicating the location of food items on the map, when the game state is passed into generating successors, it updates the grid as well. This is achieved by accessing the location that Pac-man is moving into on the grid and updating its value to false. The final goal, being that there are no more goal states to visit, is reached once the item count in *foodGrid* is 0, or no values are true.

State space:

Determined by the size of the gameState passed to this object, the state space will consist of the grid in which Pac-man can operate, where each possible location is represented with an (x, y) coordinate. So, there will be $x * y$ possible states for Pac-man. Then we have to consider our goal states which are the n food items located somewhere within our maze. We represent this using a Boolean grid, so each coordinate will be assigned a true or false value, so there will be 2^n states for the food. Therefore, the size of the state space is $xy * 2^n$

Initial state:

(x_i, y_i) where Pac-man begins, indicated by a 'P' in the layout file, and the true values for the food items in the grid.

```
Goal state:

      Food grid is populated entirely by false values.

Operators:

      The indicated direction N, S, E, W (valid moves
      defined by getSuccessors( )). Food grid values
      updated to false if Pac-man visits a space with a
      true value.
```

Analysis

Agents:

A Closest Food Manhattan Distance*

A* utilizes a backwards sorted priority queue by the heuristic *f(x)=g(x)+h(x)* where g(x) is the cost of the move, and h(x) is the tiebreaker. The h(x) using Manhattan distance can be best defined as the shortest horizontal and vertical distance between two point, in our case Pac-man's position and a food item. The distance is calculated by summing the absolute value of the difference between the two points.

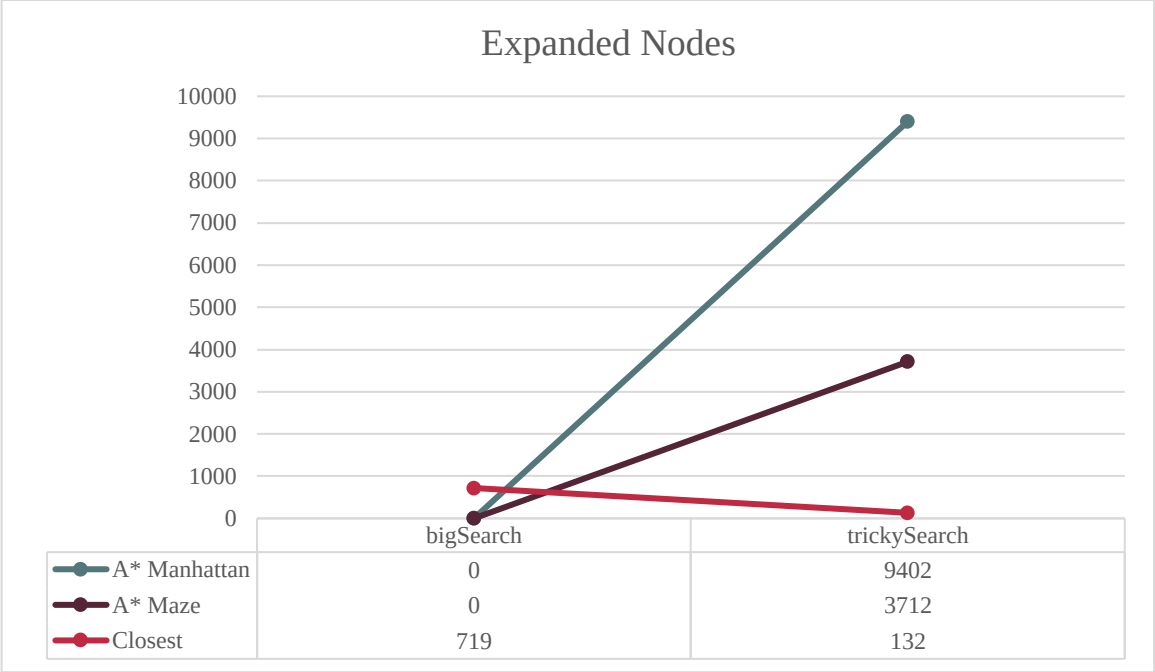
A Closest Food Maze Distance*

A* utilizes a backwards sorted priority queue by the heuristic *f(x)=g(x)+h(x)* where g(x) is the cost of the move, and h(x) is the tiebreaker. The h(x) using the maze distance heuristic is defined as the length of a BFS from the current location to the given food item while also considering the initial game state. This heuristic is calculated by running a position search problem using the coordinates of the current position as the initial state, and the food item as the goal state.

Closest Dot Search

This search is very straight forward in terms of implementation. Using a BFS, this method searches for one food at a time by sequencing together multiple PositionSearchProblems. So, from the initial state, a BFS is run (which means a FIFO queue is our data structure) until the closest food item is found. Once this item is found, Pac-man will take the path. Now, from this new state, the search checks if the food count is greater than 0, if it is the search continues. This process repeats until there are no goal states left to reach.

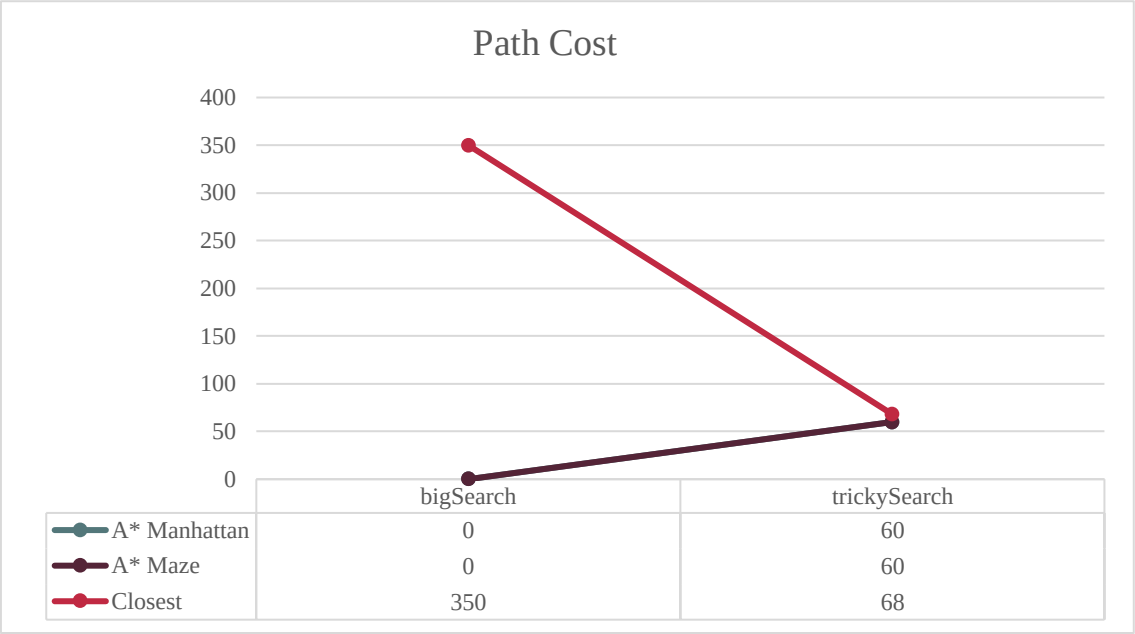
Plots:



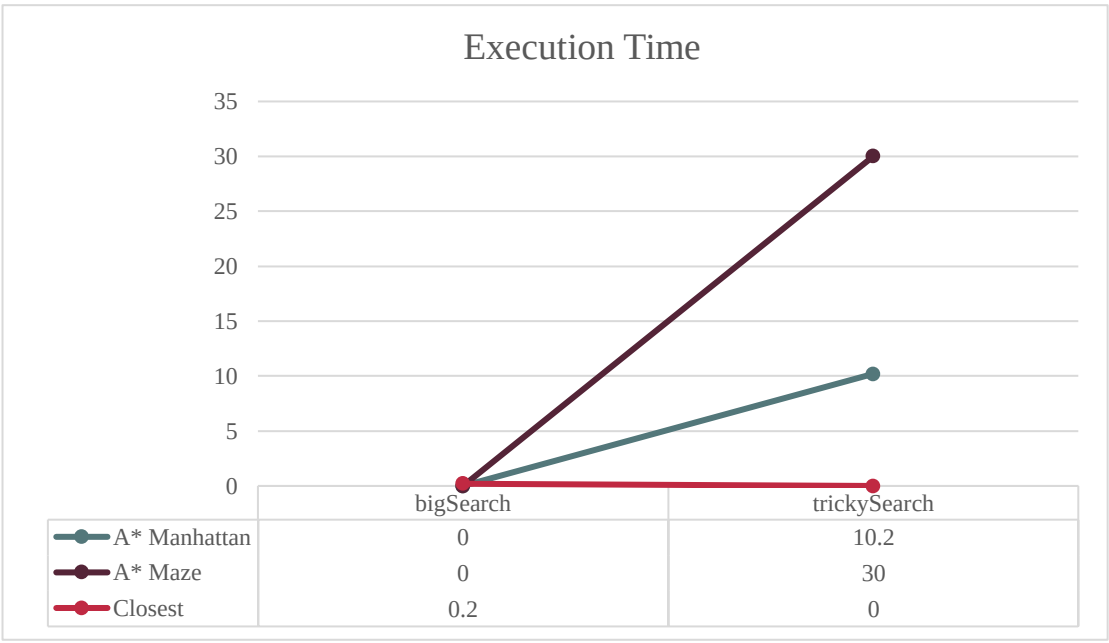
From the data I was able to collect, we are able to take away a few items. First, I want to address the failure of both A* algorithms to execute on the bigSearch maze. I let both of these algorithms run over night and neither were able to complete their execution. The reason this happens is the size and complexity of this maze. As there are 220 food items in bigSearch, the A* algorithms have to search for an optimal solution in a hypothesis space of 220! which essentially guarantees that this search will never terminate. From this I can roughly estimate values that would have been achieved, in the sense that my estimates will be immense for expanded nodes and time complexity. As for the path cost, maybe A* does eventually find a more optimal path, but the space and time complexity do not even begin to justify A* as more effective for this problem. Therefore, I will more closely analyze the algorithms' performances on trickySearch.

From this first graph we can see the large differences in number of nodes expanded. I want to emphasize the relatively small state space for trickySearch in comparison to the number of nodes expanded. Operable space for Pac-man is only 18 spaces wide by 5 spaces deep, yet A* Manhattan expands over 9000 nodes in this search. This can be attributed to the decisions that have to be made, as having more than one goal severely effects the performance of A* as the number of heuristic calculations that need to be made rise greatly. However, we see the A* using the maze distance heuristic expands just over 1/3 of the number of nodes when compared to Manhattan. This is due to the maze distance being a simpler heuristic that Manhattan distance. The maze distance as I describe above operates off of a BFS and will calculate the heuristic based on the lowest cost (closest) food items. Our third algorithm is very similar to this,

and we see such a low number of expanded nodes because the closest food search is just the first iteration of an A* with maze distance as the heuristic. The A* keeps expanding nodes though because it needs to determine which path will yield the lowest cost.



Here we see that the tradeoff for space complexity of the A* algorithms is that they will find the optimal path. While the difference is not all that much between A* and closest food, it is to be noted like I mentioned that the trickySearch maze is not that large, so already there is not much variability in the possible paths. While the execution did not finish, we can also definitively say that A* *would* have found the optimal path because the heuristics are admissible.



In line with the number of nodes expanded, we see that the A* algorithms take significantly more time to operate than the closest food search as well. Again, the fact that the closest food search is in essence a sequence of position searches using a BFS

ensures that there are no heuristic calculations that need to be made, simply generating successors into the queue, which takes minimal time. I believe the time difference is seen between the two A* algorithms because Manhattan distance will be a more efficient heuristic than maze distance. Part of this is due to the code for calculating each, where the maze distance has to perform checks for walls and call other functions (including running a BFS), while Manhattan simply calculates the heuristic and returns.

Conclusion & comments

In total, we have analyzed the performance of 3 different algorithms on mazes of varying sizes, complexity, and with different goals to accomplish. While I don't think that based on our results we can say there is necessarily a *best* algorithm to use for mazes, we can say that there are conditions in which some will certainly outperform others. DFS will perform well on mazes in which the goal state is far away, but because it is uninformed and traverses so deep this search will suffer in optimality and will seldom be the best choice. BFS will perform well on mazes where the goal is nearby and will also always find a solution if it exists. BFS does tend to expand more nodes than necessary, but this again can be credited to the fact that it is an uninformed search. While we did not analyze the performance of UCS, this is just a special BFS which will reach similar results, just usually more efficient across the board. A*, the only informed search we observed most certainly was the best choice for our smaller mazes with single goals (obviously heuristic depending). However, this search suffers on mazes with more than one goal and makes me question whether the optimality tradeoff is worth the inefficiency in terms of time and expanded nodes.

- Difficulties
    - Determining which searches were provided to us as a way of detecting poor performance (i.e. bigSearch with A* or a BFS more optimal than A*)
    - Tracing code through the different method calls that jumped from class to class
- Challenges
    - Designing a maze with the goal of producing specific behavior (trial and error)
    - Getting searches to *not* show good behavior (typically when the good performance arose by accident when trying to show different behavior)
    - Understanding *why* searches were failing to perform
- Benefits
    - Code reading abilities greatly improved
    - Benchmarking and comparisons for algorithm performance
    - Translation of concepts to implementation