**QS1.1:**

**Data structure:** Stack. Because DFS’s strategy expand a deepest node in the frontier first and stop when the goal is picked to expand. So, the last one added to the frontier intuitively is a deepest node.

**QS1.2:**

* No, the exploration order is not what I would expected. Because it is not the optimal rote.
* Yes, Pacman actually go to all the explored squares on his way to the goal

**QS2.1:**

**Data structure:** Queue. Because BFS’s strategy is that the node in frontier should be expanded and the sallowest node is expanded first. It’s like first in first out.

**QS3.1:**

Uniform-Cost Search is to find the lowest-cost path between the nodes representing the start and the goal states. It expands nodes according to their path costs form the root node. Uniform cost search could solve any graph/tree where the optimal cost is in demand. A uniform-cost search algorithm is implemented by the priority queue. It gives maximum priority to the lowest cumulative cost. Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.

So, we create a function that takes in weighted edges of the GRAPH which outputs the directed weighted graph. This function needs to take in the directed weighted graph and starting vertex which output the shortest path from starting vertex to every other node in the graph and its predecessor dictionary.

**Q4.1:**

**Null heuristic:**

The null-move heuristic is designed to guess cutoffs with less effort than would otherwise be required, whilst retaining a reasonable level of accuracy. It returns 0 for any node.

**Manhattan distance heuristic:**

Given a particular state, consider every non-empty tile. Calculate the Manhattan distance between the current position of the tile and the goal position of the tile. Add this value for all the non-empty tiles together, and we have the heuristic value. Let’s calculate the Manhattan distance heuristic value for the initial state on the left. Calculate this value yourself. Then, keep reading for the answer.

NullHeuristic is no guarantee a optimal solution.

The Manhattan distance heuristic is better than NullHeuristic is because that the Manhattan distance heuristic is admissible since it considers each tile independently (while in fact tiles interfere with each other). And it's optimistic.

The priority of an item is given by it's cost plus the heuristic for the end state of the action.

**QS5.1:**

The approach was to store the state as a tuple of Pacman’s position and the remaining corners to be visited.

**QS5.2:**

The problem is solved when Pacman’s position is equal to the only remaining unvisited corner. The successors are computed by filtering illegal locations and then constructing the next state by filtering the unvisited corners to make sure none of them are Pacman’s current position.

**QS6.1:**

In this heuristic, we use ManhattanHeuristic. Manhattan distance heuristic for a PositionSearchProblem by calculating the Manhattan distance between the source and goal.

**Strong points:** pacman allowed to move only in four directions only (right, left, top, bottom)

**Weak points:** However, once you allow diagonal or any-angle movement, manhattan distance becomes nonadmissible  because it overestimates diagonal costs, which necessarily means it's not consistent.

It is admissible because it is always . And it is consistency. Because for every node n and every successor n' of n generated by any action a: h(n) ≤ c(n,a,n') + h(n')

**QS7.1:**

In this heuristic, we use euclideanHeuristic. Euclidean distance heuristic for a PositionSearchProblem by calculating the Euclidean distance between the source and goal.

**Strong points:** pacman allowed to move in any directions. Since Euclidean distance is shorter than Manhattan, you will still get shortest paths.

**Weak points:** May have trouble with using A\* directly because the cost function g will not match the heuristic function h.

It is admissible because it is always . And it is consistency. Because for every node n and every successor n' of n generated by any action a: h(n) ≤ c(n,a,n') + h(n')

**QS8.1:**

We solve this problem by Bread First Search algorithm.

Strong points:

* Its biggest benefit is that it is relatively straightforward to use and comprehend.
* Tends to find paths with fewer steps than other algorithms, such as Depth First Search.
* Can be easily parallelized, which means it can take advantage of multiple processors to speed up the search.
* Guaranteed to find the shortest path from the starting point to the goal.

Weak points:

* It can be slow since it expands all the nodes at each level before continuing on to the next
* It can be quite memory demanding because it needs to keep track of every node in the search tree.
* It occasionally finds less-than-optimal solutions because it doesn't fully investigate all potential paths through the search tree.