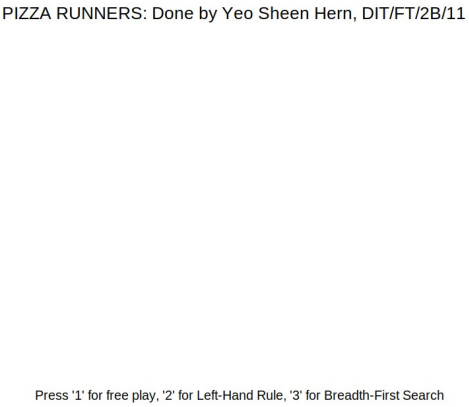
DSAA CA1 Report

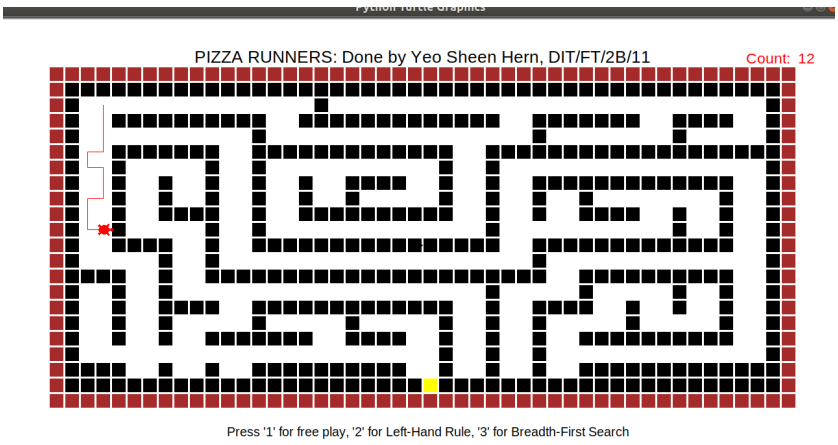
Part (a) Description of usage of app:

After launching:



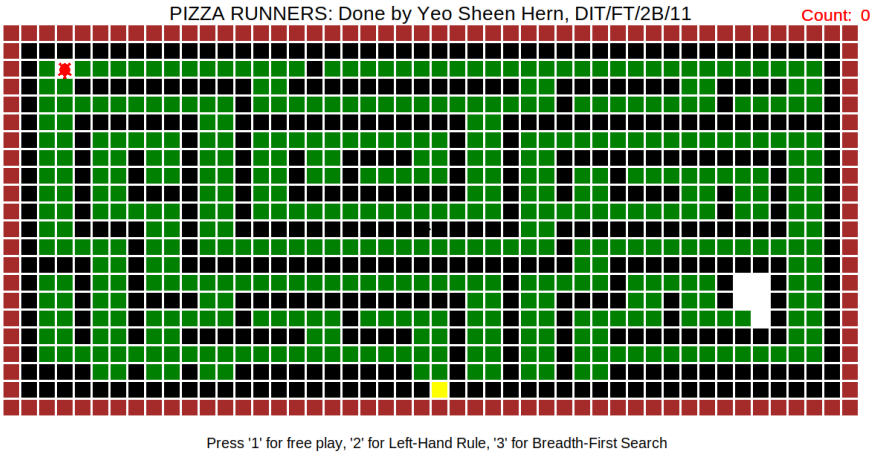
Press on keyboard ‘1’, ‘2’, or ‘3’ to select Game mode or Algorithm Visualisation.

Free Play:



Use Arrow Keys to move around. Press ‘1’ to restart, ‘2’/’3’ to switch to algorithm visualization.

Algorithm Visualizations:



Press ‘1’ to switch to Free Play, ‘2’/’3’ to switch algorithms or restart visualization.

Part (b) Algorithms:

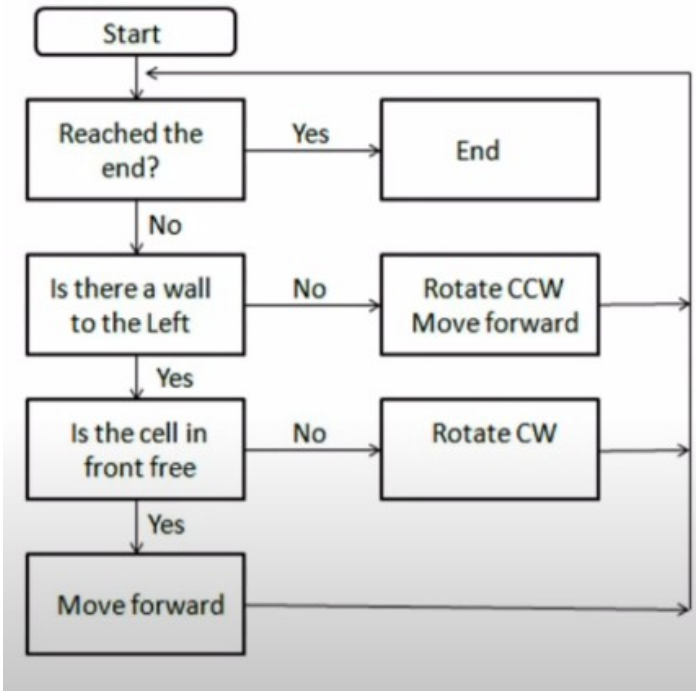
Left-Hand Rule:

The left-hand rule is one of the simplest algorithms to implement due to its simple rule set.

1. If there no wall to the left, rotate counter-clockwise and move forward

2. If there is a wall to the left and the cell in front is free, move forward.

3. If there is a wall to the left but the cell in front is not free, rotate counter-clockwise



We just repeatedly check this set of rules until the drone finds its destination.

The drone is guaranteed to find its destination using this algorithm!

However, there are various limitations and constraints with this algorithm. One of which is that the drone almost never finds the most efficient path to its destination. In our example, the left-hand rule took over 400 steps to reach its destination, while our other algorithm reached its destination in just steps.

Another constraint is that, in the event that there are no walls around the starting position of the drone, the drone may end up traveling in an infinite loop. This was one of the challenges that I faced while implementing this algorithm. I had to make sure that the drone found a wall, before starting its main left-hand rule loop to prevent a case where the drone would just walk in circles. Finally, even when we found a wall, there was also a possibility that if its destination was not connected to any walls, we would not find the destination!

In terms of the data structures and complexity of this algorithm, this algorithm is very computationally easy to run because of its simplicity. We are only checking a simple rule set every step the drone takes before making its next move. However, it is thus not very efficient in finding a short path to its destination. As for the data structures used, we stored the walls of our map in a simple 2D Array for the algorithm to simply check on.

Breadth First Search

Breadth-First Search is one of the best path-finding algorithms to find the shortest path. This is because of how its searches every single possible path and covers every route and cell in the map to determine which path is the shortest to take. However, the trade-off for this efficient pathing is that it is much more computationally expensive.

In this algorithm, we use various data structures to store the visited, frontier and backtrace path.

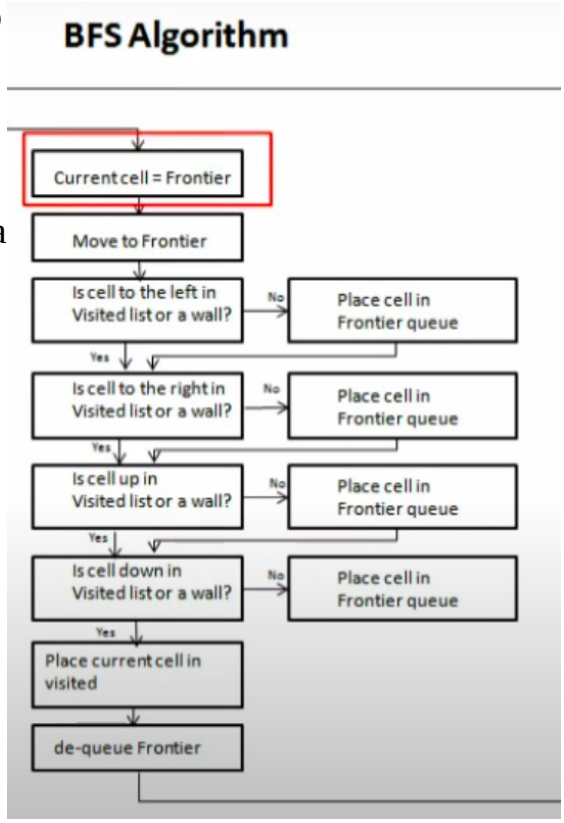
1) Make current cell first cell in frontier queue.

2) Check all adjacent walls, if not visited and not a wall, place wall in frontier queue. Also put the current cell as the value to the solution dictionary and the frontier cell as the key to facilitate backtracking.

3) Place current cell in visited

4) De-queue frontier

We repeat this until there are no more cells in the frontier queue, and backtrack to find solution.



This algorithm is amazing, as it always manages to find the shortest path to the drone's destination. However, its biggest limitation is that it is extremely computationally expensive to search every cell and route in the map to find the shortest path for the drone to take, and thus will work much better in small maps, where there are less cells and paths for the algorithm to cover.

In terms of the data structures used in this algorithm, we used a deque for the frontier queue, a set for the visited, and a dictionary for the solution backtracking. The deque for the frontier queue is ideal as we do not need the functionalities of a list to add elements to the middle of an array. By usinga deque instead, we make it much more efficient to store and access the values we want without having to traverse through various nodes like a list. As for the set, it ensures that there are no duplicates and also allows elemts to be traversed in the way there were entered. Finally, the dictionary for the solution backtracking is ideal as we can store the cell in which we found the next cell from as we do our breadth-first search and easily recursively find the shortest path to the start from the destination.]

The complexity of this algorithm is not very ideal as we have to search through all possible cells and paths, making it relatively complex. Thankfully, we only have to loop through everything once, and by using the suitable data structures like we did, we are able to ensure that the algorithm is not toocomplex.