# **Task 1**

### **Algorithm**

Since there are relationships between each point with their neighboring points (horizontally, vertically, diagonally as well as points that can be reached with a straight line without passing any other points), we will take a graph approach for this problem.

Since we want to visit every single possible combination of points, we will take a depth-first-search approach using a recursion implementation. The algorithm will be written in Java.

### **Implementation Details**

### Assumption: The screen grid is 3 x 3 and the alphabets are from A - I.

Each point is mapped to an integer, that can be represented in row and columns using the equation x = j \* 3 + i, where i is the row and j is the column. For example: ‘E’ is 5 which is row 1 column 2, 5 = 1 \* 3 + 2.From now onwards, we will refer the point to the number representation (5 instead of ‘E’).

Data structure needed:

graph - Map<Integer, Set<Integer>> to store the nodes.

posPoint - Map<Integer, Integer> to store the first to third point that we have to traverse in order.

visited - a set to keep track of visited nodes.

res - an array of strings in number representation.

finalRes - an array of strings in alphabet representation.

### Graph Generation

We will use a Hashmap where the key is the corresponding point, and the value is the list of neighbors.

While manually writing the graph is possible for a small grid and the neighbors are fixed, we will write a function that creates the graph. For each point, we create a list of neighboring points.

1. Graph traversal

We will create a recursive function dfs() that takes the point, the current position of the desired points traveled, and the current path from the beginning.

If we have already reached the final desired point for the ordering, we will add the path to the result, then we return. This is our base case.

Else, we go through every single point. We will only recursively call dfs if it matches any of the 2 conditions: it is an unvisited neighbor, or it's an unvisited non-neighbor separated by a visited neighbor.

How do we check the latter? The non-neighbors are always separated by a midway neighbor. The midway point is always a neighbor because it is always reachable either diagonally, vertically or horizontally directly. Hence, we will do the check by checking the sum of the two points, and then divide by 2 and check if that point has been visited.

We will also check for the case where we have only reached the first point, but the next point is the 3rd point. In this case we will only ignore.

Once we get all the results, we will convert the number representation of points back to string representation.

### **Result/Performance**

Space complexity: Without a fixed grid size, the space depends on the size of the graph: O(n\*n), where n is the number of nodes. This is because nodes can be neighbors with each other. Since each node has a fixed number of neighbors, and the number of nodes are fixed, it will be a constant space usage.

Time complexity: The runtime depends on the recursive dfs function. Since we have to test each possible combination of paths (only the first node is fixed), 1 \* (n-1) \* (n-2)... which simplifies to O(n^n). However, since the number of nodes are fixed, it will be a constant runtime.

# **Task 2**

## Part 1

### **Observation**

Upon inspection, the initial generation, mutation, and crossover function seems fine, as well as the fitness evaluation. With more reading, the observation is that I should manipulate the population size, the mutation rate and the crossover rate so that the algorithm will not overlook the optimum solution.

### **Approach**

I experiment with bigger population size, smaller mutation rate and bigger crossover rate. I finally settled with a population size of 250, a mutation rate of 0.2, and a 0.9 crossover rate to get a minimum cost of RM117.++.

## Part 2

### **Observation**

The population size, mutation rate and crossover rate will retain as is from part 1.

The initial generation and the fitness evaluation needs to be changed.

The initial generation of each individual does not allow any destination to *not* have cars - that is, each customer must be satisfied. There are also unlimited cars in part 1, but we can only have 2 cars. Hence we change the number of cars, and also introduce -1 into the individuals that indicate that we will not satisfy the demands of this customer.

For example, for a test case of 4 customers and 2 cars, and an individual initially [ 0, 1, 2, 5, 3, 4, 6], we introduced -1 randomly to be [0, -1, 2, 5, 3, 4, 6]. This means that the car with a car id of 5 will skip the first customer (at index 1).

The fitness evaluation also has to be changed to accommodate the -1. Currently, the fitness evaluation assumes that each customer will be served, so it calculates the distance from this point to the next point without checking. Hence, we have to change it so that we find the next point to be reached (or terminate the distance calculation if this point is our final destination).

Since the demands now is a variable that has to be accounted, we will change the fitness evaluation equation that rewards more demands and less cost ( 100 \* total\_demands / total\_cost).

### **Approach**

In the init\_population function, once the individual is totally generated, we will range over it and change the element to a -1 (only if it is a customer id) randomly.

In the fitness evaluation function, for both a car point and a customer point, we will make sure that the next point is not a -1. If it is, then we will find the next non -1 point by traversing the point until we hit the final element or a car point. If it is already a final destination, then the distance is 0.

Here will introduce a bug in the crossover that will only generate properly if the entire array is non -1 because it is initialized to an array of None. Hence, we change the crossover function to initialize the C1 and C2 array to -1s instead to fix the bug.